

SUDOKU GAME PROJECT



TABLE OF CONTENTS

ABSTRACT	6
CHAPTER 1	7
INTRODUCTION	7
1.1 Introduction	7
1.2 Motivation	8
1.3 Problem Statement	9
1.4 Challenges	10
1.5 Project Scope	10
CHAPTER 2	12
LITERATURE SURVEY	12
2.1 Early Development of Sudoku	12
2.2 Digital Transformation	13
2.3 Advancements and Innovations	13
2.4 Community and Social Features	14
2.5 Recent Trends and Future Directions	15
CHAPTER 3	16
METHODOLOGY	16
3.1 Planning	16
3.2 Design	17
3.3 Development	18
3.4 Testing	19
CHAPTER 4	20
IMPLEMENTATION	20
4.1 Importing Libraries	20
4.1.1 Pygame Library	20
4.1.2 OS Library	21
4.1.3 random Library	21
4.2 Game Window	23
Components of the Game Window	23
4.3 Grid Class	24
Responsibilities of the Grid Class	24
4.4 Grid Numbers	27
Key Functions	27
Grid Generation Process	29
4.5 Draw Numbers	29
Setting Up the Font	30

Drawing the Numbers-----	30
Managing Cell Values-----	31
Drawing Text on the Grid-----	32
4.6 Remove Numbers-----	33
Purpose of Remove Numbers-----	33
Implementation of Remove Numbers-----	33
4.7 Pre-occupied Cells-----	35
Purpose of Pre-Occupied Cells-----	36
Implementation of Pre-Occupied Cells-----	36
4.8 Mouse Click-----	37
Purpose of Mouse Click Detection-----	38
Implementation of Mouse Click Detection-----	38
4.9 Number Selection-----	40
SelectNumber Class-----	41
Properties of SelectNumber Class-----	41
Drawing the Buttons-----	42
4.10 Button Hover-----	44
4.11 Button Click-----	46
4.12 Number Coloring-----	48
4.13 Check Grids-----	50
4.14 Draw Text-----	51
4.15 Restart Game-----	52
4.16 Code-----	55
CHAPTER 5-----	60
RESULTS-----	60
5.1 Game window-----	60
5.2 Solving the game-----	61
5.3 Restart of the game-----	62
CHAPTER 6-----	63
CONCLUSION-----	63
CHAPTER 6-----	64
REFERENCES-----	64

ABSTRACT

The Sudoku Game Project is designed to create a digital version of the popular logic-based puzzle game, Sudoku. This project aims to develop an interactive and user-friendly platform that allows players to engage with Sudoku puzzles of varying difficulty levels. The application will provide features such as puzzle generation and solution validation to enhance the user experience. The primary objectives include improving problem-solving skills, promoting cognitive development, and offering an entertaining and educational tool for users of all ages. The project will be implemented using modern programming languages and frameworks, ensuring cross-platform compatibility and accessibility. This abstract outlines the project's goals, significance, and the anticipated outcomes, providing a comprehensive overview of the Sudoku Game Project.

CHAPTER 1

INTRODUCTION

1.1 Introduction

The Sudoku Game Project is a comprehensive initiative aimed at bringing the widely cherished puzzle game, Sudoku, to the digital realm. Sudoku, which originated in Japan, has become a global phenomenon due to its simple yet challenging nature. The game consists of a 9x9 grid divided into nine 3x3 subgrids, where the objective is to fill the grid with numbers from 1 to 9. Each number must appear exactly once in each row, column, and subgrid. This combination of simplicity and complexity makes Sudoku an ideal candidate for a digital transformation.

The Sudoku Game Project holds significant potential in terms of both entertainment and education. Sudoku is not just a game; it is a powerful tool for mental exercise that can help improve various cognitive skills. By bringing Sudoku to a digital platform, the project aims to make these benefits more accessible to people of all ages and backgrounds. The digital format also allows for the inclusion of additional features and enhancements that are not possible with traditional paper-based Sudoku puzzles.

The Sudoku Game Project represents an exciting opportunity to merge traditional puzzle gaming with contemporary digital technologies. By creating an interactive and versatile Sudoku application, the project aims to offer users a compelling and

beneficial experience that combines entertainment with cognitive development. Through thoughtful design and implementation, the Sudoku Game Project aspires to set a new standard for digital puzzle games, making Sudoku more accessible, engaging, and educational than ever before.

1.2 Motivation

The Sudoku Game Project is driven by several key motivational factors, each contributing to the desire to bring this classic puzzle game to a digital platform. These motivations encompass the game's widespread popularity, its educational benefits, the opportunities for innovation in digital gaming, and the potential for enhancing user experiences.

One of the most compelling motivations for developing a digital Sudoku game is the educational value it offers. Sudoku is more than just a game; it is a powerful tool for cognitive development. Playing Sudoku enhances critical thinking, problem-solving skills, and concentration. It helps players develop pattern recognition and logical reasoning abilities. By digitizing Sudoku, I aim to make these cognitive benefits more accessible to a broader audience, including students, educators, and lifelong learners. The game's structured challenges can serve as an effective educational resource, promoting mental agility and academic growth.

Sudoku has garnered a massive global following since its rise to prominence in the late 20th century. Its simple rules and challenging puzzles have made it a favorite pastime for millions of people. The universal appeal of Sudoku lies in its ability to captivate players with its logical structure and progressively difficult challenges. By creating a digital version, we aim to tap into this widespread popularity and

provide a convenient, accessible way for people to enjoy Sudoku anytime, anywhere.

1.3 Problem Statement

The Sudoku Game Project addresses several key challenges associated with the traditional, paper-based format of Sudoku puzzles. Despite the game's popularity, there are notable limitations and inefficiencies that can hinder the overall user experience and restrict access to the cognitive and educational benefits of Sudoku.

Traditional Sudoku puzzles are typically available in books, newspapers, or printed sheets. This format restricts when and where players can engage with the game, limiting their ability to enjoy Sudoku during moments of free time or while on the go. There is a need for a more accessible solution

Printed Sudoku puzzles are static and offer a limited selection. Once a puzzle is completed, players need to acquire new books or printouts to continue playing. This approach is not only inconvenient but also restricts the variety and customization of puzzles. There is a demand for a solution that can generate a virtually infinite number of unique puzzles and allow customization of difficulty levels to cater to individual preferences and skill levels.

The reliance on printed materials for traditional Sudoku puzzles contributes to paper waste, which has an environmental impact. A digital solution can help reduce this waste by providing an eco-friendly alternative that eliminates the need for paper and ink.

1.4 Challenges

1.4.1 Puzzle Generation: Developing an algorithm capable of generating an infinite number of unique Sudoku puzzles with varying levels of difficulty is a complex task. The algorithm must ensure that each puzzle is solvable and adheres to Sudoku's rules.

1.4.2 Solution Validation: Implementing a robust solution validation mechanism is essential. The system must accurately detect and respond to incorrect entries and incomplete solutions in real-time. This requires sophisticated logic to ensure that the feedback provided to users is both accurate and helpful.

1.4.3 Performance Optimization: Ensuring that the game runs smoothly without lag or crashes, especially during complex operations like puzzle generation and validation, is essential. Performance optimization involves efficient coding practices and thorough testing to identify and mitigate potential issues.

1.4.4 Balancing Challenge and Learning: Designing puzzles that strike the right balance between being challenging and educational is critical. The game must cater to various skill levels, providing a learning curve that helps users improve while keeping them engaged and entertained.

1.5 Project Scope

The Sudoku Game Project encompasses the development of a comprehensive digital platform for playing Sudoku. This project aims to create a robust, user-friendly application that offers a rich and engaging experience for Sudoku

enthusiasts of all skill levels. The scope of the project includes the following key areas:

1. Puzzle Generation and Difficulty Levels

- a. **Algorithm Development:** Design and implement an advanced algorithm capable of generating a virtually infinite number of unique Sudoku puzzles. The algorithm will ensure each puzzle is solvable and adheres to Sudoku rules.
- b. **Difficulty Levels:** Provide multiple difficulty levels ranging from easy to expert to cater to players of different skill levels. The difficulty levels will be adjustable, allowing players to choose their preferred level of challenge.

2. User Interface and User Experience

- a. **Responsive Design:** Develop a clean, intuitive, and visually appealing user interface that is optimized for different devices, including desktops, tablets, and smartphones.
- b. **Ease of Use:** Ensure the interface is user-friendly, with easy navigation and clear instructions, making it accessible to players of all ages and technical abilities.
- c. **Accessibility Features:** Incorporate accessibility features such as adjustable font sizes, high-contrast color schemes, and screen reader compatibility to make the game inclusive for all users.

CHAPTER 2

LITERATURE SURVEY

The literature on Sudoku game projects encompasses a range of historical developments and innovations in the field. This survey provides an overview of key milestones and related projects that have shaped the evolution of Sudoku games, both in traditional and digital formats.

2.1 Early Development of Sudoku

1. Origins and Early Development

- **Maki Kaji's Contribution (1984):** Sudoku, known initially as "Number Place," was popularized in Japan by Maki Kaji, who was instrumental in its rise. The puzzle format, which involves a 9x9 grid with the goal of placing numbers so that each row, column, and subgrid contains all digits from 1 to 9, was first introduced by Howard Garns in the United States in 1979. However, it was Kaji's efforts that brought it widespread popularity in Japan and later internationally.

2. The Role of Publishing

- **Japanese Puzzle Books and Newspapers:** During the 1980s, Sudoku gained traction through Japanese puzzle books and newspapers. Publications like the "Nikoli" magazine were pivotal in disseminating Sudoku puzzles and establishing its popularity among puzzle enthusiasts. This early adoption in print media set the stage for future digital adaptations.

2.2 Digital Transformation

1. Early Digital Sudoku

- **First Digital Sudoku (1990s):** The transition of Sudoku to digital platforms began in the 1990s, with early implementations appearing on early computer systems and handheld devices. These initial digital versions maintained the classic rules and format of Sudoku while introducing basic digital features.

2. Web-Based Sudoku (2000s)

- **Sudoku.com and Online Platforms:** The early 2000s saw the emergence of web-based Sudoku platforms, such as Sudoku.com, which provided users with an online environment to play Sudoku. These platforms popularized the game further and introduced features like puzzle generators and difficulty settings, enhancing accessibility and user engagement.

2.3 Advancements and Innovations

1. Mobile and App-Based Sudoku

- **Sudoku on Smartphones (2007 Onwards):** With the advent of smartphones and app stores, Sudoku experienced significant growth in the mobile gaming market. Notable apps like "Sudoku by Brainium Studios" and "Sudoku.com" offered enhanced features, including various difficulty levels, hint systems, and user statistics. These apps leveraged the capabilities of mobile devices to provide a more interactive and personalized gaming experience.

2. Algorithmic and AI Enhancements

- **Advanced Puzzle Generation (2010s):** The development of sophisticated algorithms for Sudoku puzzle generation and solving has been a major advancement. Researchers and developers have focused on creating algorithms that not only generate puzzles but also analyze their difficulty. Techniques like constraint satisfaction problems (CSP) and backtracking algorithms have been employed to create a diverse range of puzzles and solutions.

3. Educational and Cognitive Tools

- **Sudoku as an Educational Tool:** The recognition of Sudoku's cognitive benefits has led to its integration into educational tools and apps. Research by cognitive scientists has explored how Sudoku enhances problem-solving skills, logical thinking, and concentration. Projects such as "Sudoku for the Brain" have been developed to leverage Sudoku for cognitive training and educational purposes.

2.4 Community and Social Features

1. Social Integration and Gamification

- **Online Leaderboards and Multiplayer Modes:** The integration of social features and gamification elements has become a significant trend in digital Sudoku games. Platforms like "Sudoku.com" and apps developed by companies such as "AppyNation" have introduced leaderboards, multiplayer modes, and social sharing options. These features aim to foster community engagement and competitive play.

2. User-Generated Content

- **Custom Puzzles and Community Contributions:** The ability for users to create and share their own Sudoku puzzles has been facilitated by digital platforms. Websites like "Puzzlemaker" and apps like "Sudoku Ultimate" allow users to design custom puzzles and share them with the community. This user-generated content contributes to the ongoing evolution of Sudoku games.

2.5 Recent Trends and Future Directions

1. Integration with Emerging Technologies

- **AI and Machine Learning:** Recent projects are exploring the use of artificial intelligence and machine learning to enhance Sudoku games. AI-driven features can provide more sophisticated hints, adaptive difficulty levels, and personalized user experiences. Research in this area focuses on using machine learning algorithms to analyze user behavior and optimize gameplay.

2. Virtual and Augmented Reality

- **Innovative Interfaces:** The exploration of virtual and augmented reality (VR/AR) technologies offers new possibilities for Sudoku games. Projects are experimenting with VR environments where users can interact with Sudoku puzzles in immersive 3D spaces. AR applications aim to integrate Sudoku puzzles into real-world environments, providing an innovative and engaging experience.

CHAPTER 3

METHODOLOGY

The methodology for the Sudoku Game Project developed using Python and Pygame is designed to ensure a structured and efficient approach to creating a fully functional and engaging Sudoku game. The game includes features that enhance user interaction, provide real-time feedback, and offer a seamless restart option. The methodology encompasses the stages of planning, design, development, testing, and deployment. This system is broken down into three main phases:

- 3.1 Planning
- 3.2 Design
- 3.3 Deployment
- 3.4 Testing

3.1 Planning

3.1.1 Requirements Gathering

- **User Needs Assessment:** Conducted initial research to identify key features required by users, such as real-time feedback for correct and incorrect inputs, a congratulatory message upon successful completion, and an easy restart option.
- **Scope Definition:** Defined the scope to include basic Sudoku gameplay, color-coded feedback for user inputs, a congratulatory end-game message, and a restart function activated by the spacebar.

3.1.2 Objectives

- **Primary Goals:** Develop a user-friendly Sudoku game that provides immediate feedback on user inputs and ensures an engaging experience through visual cues and end-game rewards.
- **Secondary Goals:** Ensure the game is easily restartable and maintains a consistent performance across various platforms.

3.1.3 Timeline and Milestones

- Created a detailed timeline with milestones for design, development, integration, testing, and deployment phases.

3.2 Design

3.2.1 User Interface (UI) Design

- **Wireframing and Prototyping:** Designed wireframes and prototypes focusing on a simple, intuitive layout that facilitates easy interaction.
- **Color Coding:** Planned the color scheme for feedback, where correct inputs turn green and incorrect inputs turn red.

3.2.2 Game Mechanics and Features

- **Puzzle Generation:** Designed algorithms for generating valid Sudoku puzzles.
- **Input Feedback:** Planned the implementation of real-time feedback for user inputs, using green for correct numbers and red for incorrect ones.

3.2.3 Technical Architecture

- **Python and Pygame:** Choose Python for its simplicity and Pygame for its game development capabilities. Defined the architecture to include modules for game logic, user interface, and event handling.

3.3 Development

3.3.1 Front-End Development

- **UI Implementation:** Developed the user interface using Pygame, ensuring a clean and responsive design. Implemented interactive elements such as clickable cells and buttons.
- **Color-Coded Feedback:** Implemented the color-coding logic to change cell colors based on user inputs, providing immediate visual feedback.

3.3.2 Back-End Development

- **Game Logic:** Developed the core game logic, including puzzle generation, validation of user inputs, and checking for puzzle completion.
- **End-Game Features:** Implemented the logic to display a congratulatory message upon successful completion of the puzzle and to restart the game when the spacebar is pressed.

3.3.3 Integration

- **Combining Front-End and Back-End:** Integrated the user interface with the game logic, ensuring smooth interaction between input handling and visual feedback.

3.4 Testing

3.4.1 Functional Testing

- **Unit Testing:** Conducted unit tests on individual components to ensure correct functionality, focusing on puzzle generation, input validation, and feedback mechanisms.
- **Integration Testing:** Performed integration testing to ensure the UI elements work seamlessly with the game logic and feedback system.

3.4.2 Usability Testing

- **User Feedback:** Conducted usability testing with real users to gather feedback on the game's intuitiveness, visual feedback, and overall experience.
- **Refinements:** Made iterative improvements based on user feedback to enhance usability and engagement.

3.4.3 Performance Testing

- **Load Testing:** Ensured the game performs well under various conditions and remains responsive.
- **Cross-Platform Testing:** Tested the game on different platforms to ensure consistent performance.

The methodology for the Sudoku Game Project ensures a comprehensive and structured approach to developing a functional and engaging Sudoku game using Python and Pygame. By following this methodology, the project aims to deliver a high-quality gaming experience with real-time feedback, intuitive design, and seamless restart functionality, meeting the needs and expectations of its users.

CHAPTER 4

IMPLEMENTATION

4.1 Importing Libraries

In the development of the Sudoku Game Project using Python, several libraries were utilized to implement the game's features and functionalities. Below is a detailed explanation of each library used:

4.1.1 Pygame Library

Pygame is a cross-platform set of Python modules designed for writing video games. It includes computer graphics and sound libraries. For this project, Pygame was essential for creating the game window, handling user inputs, rendering text and graphics, and managing the game loop.

- **Game Window Creation:** Pygame was used to initialize and manage the game window, where all visual elements of the Sudoku game are displayed. This includes setting up the window size, title, and background color.
- **User Input Handling:** Pygame handles keyboard and mouse inputs, enabling users to interact with the game by selecting cells, entering numbers, and pressing the spacebar to restart the game.
- **Graphics Rendering:** The library's drawing functions were employed to render the Sudoku grid, numbers, and color-coded feedback (green for correct inputs, red for incorrect ones). This includes drawing lines, rectangles, and text on the game window.

- **Game Loop Management:** Pygame manages the main game loop, which updates the game state and renders the graphics at a consistent frame rate. This ensures smooth and responsive gameplay.

4.1.2 OS Library

The `os` module provides a way of using operating system-dependent functionality like reading or writing to the file system. It was used in this project to handle file and directory operations.

- **File Handling:** The `os` library was utilized to manage file paths and ensure compatibility across different operating systems. This is important for loading game assets, such as fonts and images, from the correct directories.
- **Resource Management:** It facilitated the loading of game assets by providing functions to navigate the file system and access the required resources, ensuring that the game can locate and load these resources efficiently.

4.1.3 random Library

The `random` module is used to generate pseudo-random numbers for various distributions. In the Sudoku game, it was crucial for generating the puzzle grid.

- **Puzzle Generation:** The `random` library was used to generate the initial Sudoku puzzle by placing numbers randomly in the grid while ensuring the puzzle remains solvable. This random placement helps in creating unique puzzles for each new game.
- **Difficulty Adjustment:** Random functions were also used to determine the number and placement of pre-filled numbers in the grid, allowing for the creation of puzzles with varying difficulty levels.

4.1.4 deepcopy Library

The deepcopy function from the copy module is used to create a deep copy of an object, meaning that all levels of nested objects are copied recursively. This was particularly useful for managing the game state.

- Game State Management: deepcopy was used to create copies of the Sudoku grid during puzzle generation and solving processes. This allowed the game to maintain multiple states of the grid without interference, ensuring that changes to one state did not affect others.
- Puzzle Solving: When implementing the puzzle-solving logic, deepcopy was essential for exploring different possible number placements without altering the original grid, enabling backtracking and ensuring the puzzle's solvability.

```
from random import sample  
from selection import SelectNumber  
from copy import deepcopy
```

```
import pygame  
import os  
from grid import Grid
```

Fig 4.1 All libraries Imported

Each of these libraries played a crucial role in the development of the Sudoku Game Project. Pygame provided the essential tools for creating the graphical interface and handling user interactions. The os module ensured smooth file management and resource loading. The random library facilitated the generation of unique Sudoku puzzles, and deep copy was instrumental in managing the game

state. By leveraging these libraries, the project achieved a robust, interactive, and user-friendly Sudoku game experience.

4.2 Game Window

The game window is the primary interface where players interact with the Sudoku game. It is designed to be user-friendly and visually appealing, providing a seamless and enjoyable experience. The game window is created and managed using the Pygame library, which offers a wide range of functions for graphics rendering, user input handling, and event management.

Components of the Game Window

4.2.1 Creating the Game Window

The Pygame library is used to initialize the game window with a resolution of 1200 x 900 pixels. This size is chosen to comfortably display the Sudoku grid and additional UI elements, providing ample space for user interaction.

4.2.2 Setting the Title

The window title is set to "SUDOKU" using Pygame's `set_caption` function. This gives the game window a clear and identifiable name, enhancing the user experience by providing context and branding.

4.2.3 Window Positioning

The position of the game window on the screen is set using the `os.environ` module. By setting the environment variable "`SDL_VIDEO_WINDOW_POS`" to "`400, 100`", the window appears at coordinates (400, 100) on the user's screen.

This ensures the game window is positioned conveniently and consistently every time it is launched.

```
# set the window position relative to the screen upper left corner
os.environ["SDL_VIDEO_WINDOW_POS"] = "%d,%d" % (400, 100)

# create the window surface and set the window caption
surface = pygame.display.set_mode((1200, 900))
pygame.display.set_caption("SUDOKU")
```

4.3 Grid Class

The Grid Class is the core component of the Sudoku Game Project, responsible for both the game logic and the visual representation of the Sudoku grid. This class encapsulates all the functionality needed to create, render, and interact with the Sudoku grid, making the game logic clean, modular, and easy to manage. Here's an in-depth look at the Grid Class and its primary functions:

Responsibilities of the Grid Class

4.3.1 Grid Initialization

- **Setup:** The Grid Class initializes the Sudoku grid, setting up the initial state with pre-filled numbers and empty cells for player input. This setup involves either loading a predefined puzzle or generating a new one.
- **Attributes:** Key attributes include the size of the grid (9x9), the initial puzzle state, the current state reflecting user inputs, and a copy of the grid to track changes made by the player. Additionally, the size of each cell is set to 100 pixels for a clear and spacious layout.

```
class Grid:
    def __init__(self):
        self.cell_size = 100
        self.line_coordinates = create_line_coordinates(self.cell_size)
```

4.3.2 Creating Line Coordinates

- **Function `create_line_coordinates`**: This function generates the x, y coordinates needed for drawing the grid lines. By using two loops, it iterates over a range to calculate the positions for the lines. The outer loop ranges from 1 to 9 for the y-coordinates, and the inner loop ranges from 1 to 10 for the x-coordinates. This setup ensures that all the necessary coordinates for drawing the grid lines are computed accurately. A temporary list, `temp`, is used within the Drawing the Grid Lines

```
def create_line_coordinates(cell_size: int) -> list[list[tuple]]:
    #creates the x,y coordinates for drawing the grid lines.
    points = []
    for y in range(1,9):
        #horizontal lines
        temp = []
        temp.append((0, y * cell_size)) # x,y points [(0, 100), (0, 200), .....]
        temp.append((900, y * cell_size)) # x,y points [(900, 100), (900, 200), ....]
        points.append(temp)

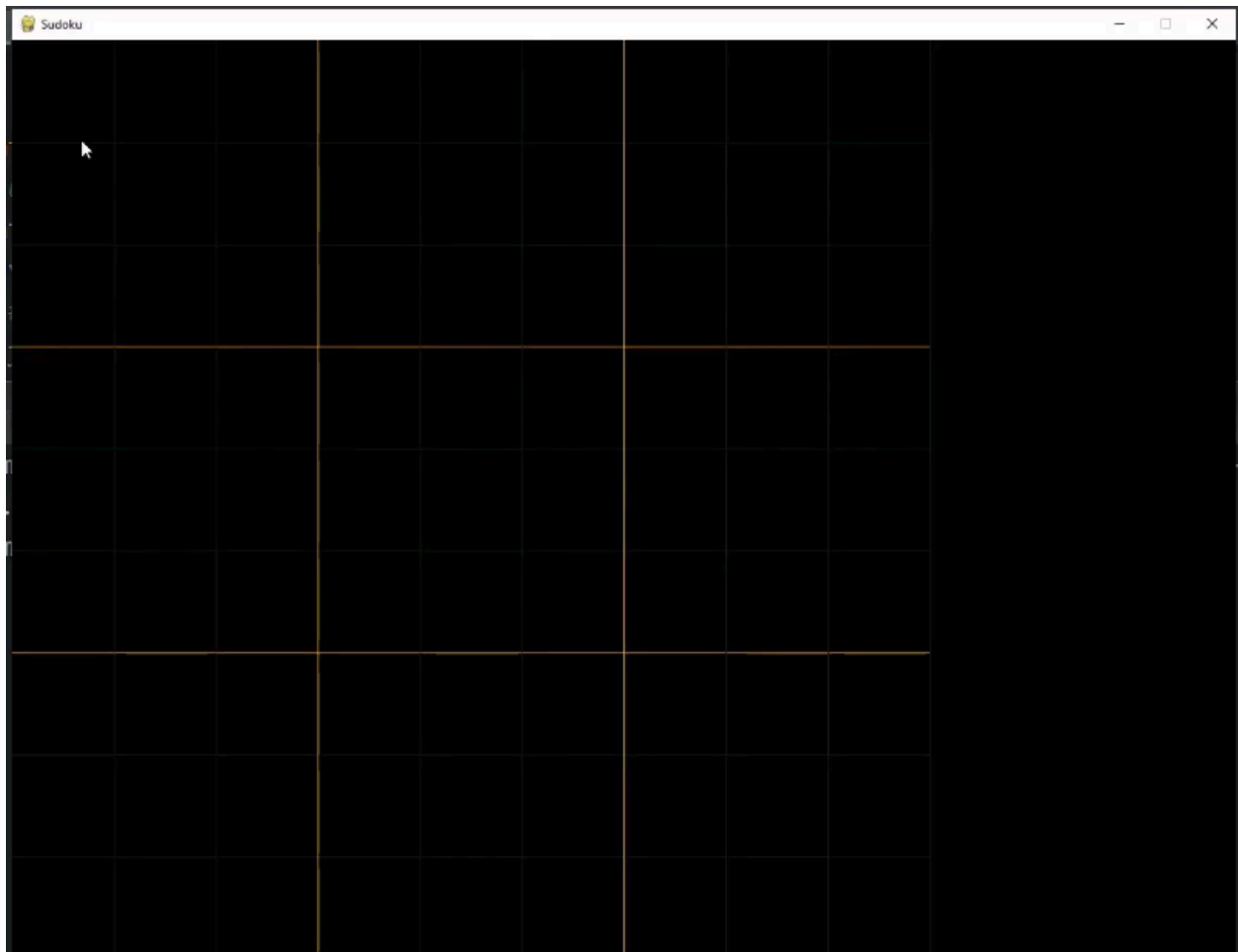
    for x in range(1,10):
        # vertical lines - from 1 to 10, to close the grid on the right side
        temp = []
        temp.append((x * cell_size, 0)) #x,y points [(100,0), (200,0), .....]
        temp.append((x * cell_size, 900)) # x,y points [(100, 900), (200,900), ....]
        points.append(temp)
    #print(points)
    return points
```

- **Function `draw_lines`**: This function handles the rendering of the grid lines on the game window. It uses Pygame's drawing functions to

color the lines. Special colors are used to emphasize the thicker lines that separate the 3x3 subgrids, providing a clear visual distinction that helps players navigate the puzzle more easily.

```
def __draw_lines(self, pg, surface) -> None:
    for index, point in enumerate(self.line_coordinates):
        pg.draw.line(surface, (0, 50, 0), point[0], point[1])
        if index == 2 or index == 5 or index == 10 or index == 13:
            pg.draw.line(surface, (255, 200, 0), point[0], point[1])
        else:
            pg.draw.line(surface, (0, 50, 0), point[0], point[1])
```

The image below is the result of the steps until now.



4.4 Grid Numbers

The Grid Numbers component is crucial to the Sudoku Game Project, responsible for generating the initial Sudoku puzzle grid with a valid set of numbers. This component involves several key functions that collectively create the logic for the Sudoku numbers, ensuring the puzzle adheres to Sudoku rules and provides a challenging yet solvable experience. Here's a detailed breakdown of the main functions and their roles in generating the grid numbers:

Key Functions

4.4.1 Pattern Function

- **Purpose:** The `pattern` function is responsible for returning the specific pattern used to fill the Sudoku grid. This pattern ensures that the numbers are placed in a way that adheres to Sudoku rules.
- **Implementation:** The function typically defines the placement logic for the numbers based on the standard 9x9 grid structure. It ensures that each row, column, and 3x3 subgrid contains the numbers 1 through 9 without repetition.
- In this example, the `pattern` function uses a mathematical formula to distribute the numbers across the grid.

```
def pattern(row_num: int, col_num: int) -> int:  
    return (SUB_GRID_SIZE * (row_num % SUB_GRID_SIZE)  
           + row_num // SUB_GRID_SIZE + col_num) % GRID_SIZE
```

4.4.2 Shuffle Function

- **Purpose:** The `shuffle` function generates a randomized version of a list, providing a different sequence of numbers each time it is called. This function ensures the grid is not the same every time the game is played, adding to the replayability and challenge.
- **Implementation:** The function utilizes the `random` library to shuffle the numbers. `sample` is imported from `random` library.
- This function takes a list `samp` and returns a new list with the elements of `samp` shuffled randomly.

```
def shuffle(samp: range) -> list:  
    return sample(samp, len(samp))
```

4.4.3 Create Grid Function

- **Purpose:** The `create_grid` function is the heart of the grid generation process. It creates a 9x9 grid filled with numbers according to the Sudoku pattern and shuffled to ensure randomness while maintaining a valid Sudoku configuration.
- **Implementation:** This function combines the pattern and shuffle functions to generate the grid. It creates rows of numbers based on the pattern function and then shuffles the rows to ensure variety.
- In this example, `create_grid` first generates the indices for the rows and columns using the `shuffle` function to ensure randomness. It then fills the grid using the `pattern` function to determine the placement of the numbers.

```
def create_grid(sub_grid: int) -> list[list]:
    #create the 9x9 grid filled with random numbers
    row_base = range(sub_grid)
    rows = [g * sub_grid + r for g in shuffle(row_base) for r in shuffle(row_base)]
    cols = [g * sub_grid + c for g in shuffle(row_base) for c in shuffle(row_base)]
    nums = shuffle(range(1, sub_grid * sub_grid + 1))
    return [[nums[pattern(r, c)]] for c in cols] for r in rows]
```

Grid Generation Process

1. **Generate Base Pattern:** The `pattern` function is used to define the base structure for placing the numbers in the grid.
2. **Shuffle Rows, Columns, and Numbers:** The `shuffle` function is called to randomize the order of rows, columns, and the numbers themselves. This ensures that each generated grid is unique.
3. **Fill the Grid:** The `create_grid` function combines the shuffled rows, columns, and numbers to fill the grid according to the base pattern. This process guarantees that the resulting grid is a valid Sudoku puzzle, adhering to the rules of having each number 1-9 appear only once per row, column, and 3x3 subgrid.

4.5 Draw Numbers

The Draw the Numbers functionality is an essential aspect of the Sudoku Game Project, responsible for displaying the Sudoku numbers on the game grid. This functionality involves setting up fonts, drawing numbers, and managing cell values. Here's an in-depth explanation of how the numbers are drawn on the grid and how cell values are managed:

Setting Up the Font

4.5.1 Font Initialization

- **Purpose:** The font is initialized to ensure the numbers are displayed in a readable and aesthetically pleasing manner.
- **Implementation:** Using Pygame's `font.SysFont` method, the font is set to "Comic Sans MS" with a size of 50. This font choice provides a clear and friendly appearance suitable for a Sudoku game.

```
pygame.font.init()
game_font = pygame.font.SysFont(name= 'Comfortaa', size= 55)
game_font2 = pygame.font.SysFont(name= 'Comfortaa', size= 35)
|
#the game loop
grid = Grid(pygame, game_font)
running = True
```

Drawing the Numbers

4.5.2 Function `draw_numbers`

- **Purpose:** This function handles the rendering of the Sudoku numbers on the grid. It ensures that each cell in the grid displays the correct number.
- **Implementation:** The function iterates through each cell in the grid and draws the number if it exists. The numbers are drawn in a bluish color for clarity and visual appeal.

```
def draw_numbers(self, surface) -> None:
    """ Draw the grid numbers. """
    for y in range(len(self.grid)):
        for x in range(len(self.grid[y])):
            text_surface = self.game_font.render(str(self.get_cell(x, y)), False, (0, 200, 255))
            surface.blit(text_surface,
                          (x * self.cell_size + self.num_x_offset, y * self.cell_size + self.num_y_offset))
```

Managing Cell Values

4.5.3 Function `get_cell`

- **Purpose:** This function retrieves the value of a cell at specific (y, x) coordinates. It is useful for checking the current value of a cell, such as when validating user inputs.
- **Implementation:** The function accesses the grid at the given coordinates and returns the cell value.

```
def get_cell(self, x: int, y: int) -> int:
    # get a cell value at y, x coordinates
    return self.grid[y][x]
```

4.5.4 Function `set_cell`

- **Purpose:** This function sets the value of a cell at specific (y, x) coordinates. It is used to update the grid based on user inputs.
- **Implementation:** The function modifies the grid at the given coordinates to the new value provided.

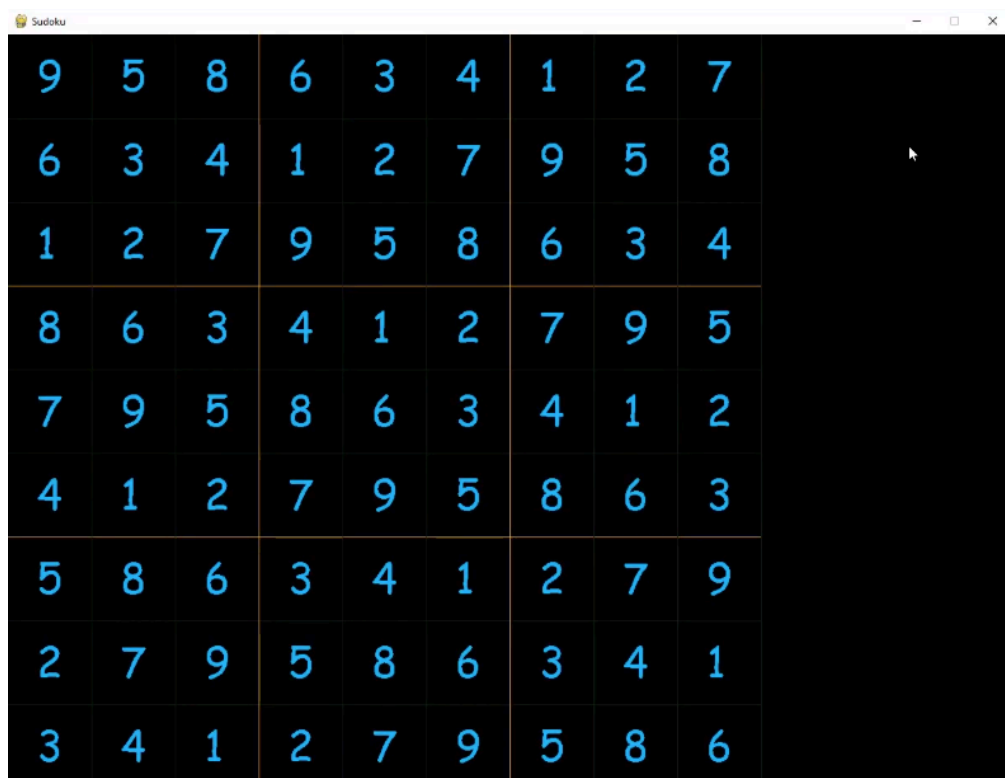
```
def set_cell(self, x: int, y: int, value: int) -> None:
    # set a cell value at y, x coordinates
    self.grid[y][x] = value
```

Drawing Text on the Grid

4.5.5 Using `surface.blit`

- **Purpose:** The `surface.blit` method is used to render the text (numbers) onto the game screen at the appropriate position.
- **Implementation:** This method takes the text surface created by the `render` method and draws it onto the screen at the specified coordinates.

The image below is the result of the grid number and draw number steps.



9	5	8	6	3	4	1	2	7
6	3	4	1	2	7	9	5	8
1	2	7	9	5	8	6	3	4
8	6	3	4	1	2	7	9	5
7	9	5	8	6	3	4	1	2
4	1	2	7	9	5	8	6	3
5	8	6	3	4	1	2	7	9
2	7	9	5	8	6	3	4	1
3	4	1	2	7	9	5	8	6

4.6 Remove Numbers

The Remove Numbers functionality is a crucial aspect of the Sudoku Game Project, responsible for creating the puzzle's challenge by removing a certain number of cells from the initially completed grid. This process involves randomly selecting cells and setting their values to zero, creating the blank spots that players need to fill. Here's a detailed explanation of how the `remove_numbers` function works and how it contributes to the game's difficulty:

Purpose of Remove Numbers

The primary goal of the `remove_numbers` function is to transform a fully completed Sudoku grid into a puzzle by removing specific numbers, leaving blank spaces for the player to solve. This step is essential in generating a playable Sudoku puzzle from the initial complete grid.

Implementation of Remove Numbers

4.6.1 Defining Variables

- `num_of_cells`: This variable represents the total number of cells in the Sudoku grid. For a standard 9x9 grid, this value is 81.
- `empties`: This variable determines how many cells will be emptied (set to zero) in the grid. It is calculated as `num_of_cells * 3 // 7`, which provides a balanced difficulty level. By adjusting the divisor (7), the game's difficulty can be fine-tuned; a smaller divisor results in more empty cells, making the game harder.

4.6.2 Randomly Selecting Cells to Empty

- **Loop with `random.sample`:** The function uses the `random.sample` method to select cells randomly to be emptied. This method ensures that the selection is unbiased and the puzzle varies with each playthrough.
- **Setting Cells to Zero:** The selected cells' values are set to zero, creating the blank spaces in the grid.

2. Adjusting Difficulty

- By modifying the divisor in the calculation of `empties` (e.g., changing 7 to a smaller number), the number of empty cells increases, thereby increasing the puzzle's difficulty. The lowest number that can replace 7 is 4. This provides a simple mechanism to scale the challenge according to the desired level of difficulty.

In this implementation:

- `num_of_cells` is set to 81 for a 9x9 grid.
- `empties` is calculated as `num_of_cells * 3 // 7`.
- `random.sample(range(num_of_cells), empties)` selects random cells to be emptied.
- A loop iterates through the selected cells, converting them to zero by setting the corresponding grid values to 0.

```
def remove_numbers(grid: int) -> None:
    # randomly sets numbers to 0 on the grid
    num_of_cells = GRID_SIZE * GRID_SIZE
    empties = num_of_cells * 3 // 199 # 7 is ideal - higher this number means easier game
    for i in sample(range(num_of_cells), empties):
        grid[i // GRID_SIZE][i % GRID_SIZE] = 0
```

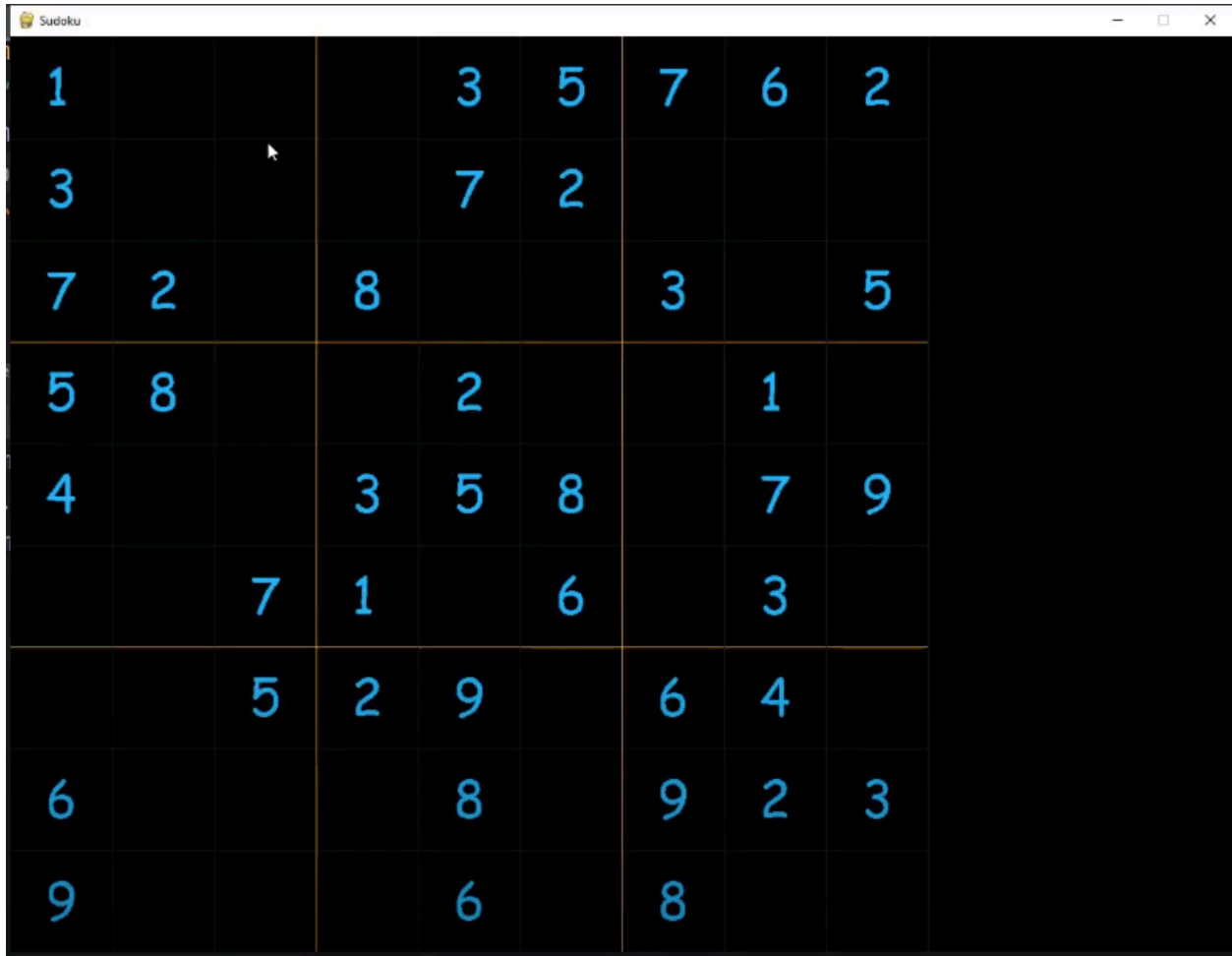



Figure of sudoku after we remove the numbers

4.7 Pre-occupied Cells

The Pre-Occupied Cells functionality is an important part of the Sudoku Game Project, focusing on identifying and managing cells that are pre-filled in the Sudoku grid. This function ensures that the game logic is aware of which cells contain initial values and should not be altered by the player. Here's an in-depth explanation of how the `pre_occupied_cells` function works and its significance in the game:

Purpose of Pre-Occupied Cells

The primary goal of the `pre_occupied_cells` function is to gather the coordinates of all cells in the grid that are pre-filled with numbers at the start of the game. These cells are considered immutable during gameplay, meaning the player cannot change their values. This distinction is crucial for maintaining the integrity of the puzzle.

Implementation of Pre-Occupied Cells

4.7.1 Function Definition

- The function `pre_occupied_cells` scans the entire Sudoku grid to identify cells that are pre-filled with numbers.
- It collects the coordinates (y, x) of these cells and stores them in a list for easy reference.

4.7.2 Iterating Through the Grid

- The function iterates through each cell in the 9x9 grid using nested loops. The outer loop iterates over the rows (y-coordinates), and the inner loop iterates over the columns (x-coordinates).

4.7.3 Identifying Pre-Filled Cells

- During each iteration, the function checks if a cell contains a non-zero value. Non-zero values indicate that the cell is pre-filled.
- If a cell is pre-filled, its coordinates (y, x) are added to the list of pre-occupied cells.

4.7.4 Returning the List of Coordinates

- After scanning the entire grid, the function returns the list of coordinates representing the pre-occupied cells.

```
def pre_occupied_cells(self) -> list[tuple]:
    #gather the u and x coordinates for all preoccupied cells
    occupied_cell_coordinates = []
    for y in range(len(self.grid)):
        for x in range(len(self.grid[y])):
            if self.get_cell(x, y) != 0:
                occupied_cell_coordinates.append((y, x)) # first the row, then the column: y,x
    return occupied_cell_coordinates
```

In this implementation:

- The function `pre_occupied_cells` takes the Sudoku grid as an input.
- An empty list `pre_occupied` is initialized to store the coordinates of pre-filled cells.
- Nested loops iterate through each cell in the grid, checking if the cell's value is non-zero.
- If a cell is pre-filled, its coordinates (y, x) are appended to the `pre_occupied` list.
- The function returns the list of pre-occupied cell coordinates.

4.8 Mouse Click

The Mouse Click functionality is a critical component of the Sudoku Game Project, enabling user interaction through mouse clicks to select and modify cells within the grid. This functionality involves detecting mouse clicks within a specific area of the game window, determining the exact cell coordinates of the click, and handling cell value changes based on whether the cell is preoccupied. Here's an

in-depth explanation of how the `get_mouse_click` function works and its significance:

Purpose of Mouse Click Detection

The primary goal of the `get_mouse_click` function is to detect when and where a user clicks within the game window, specifically focusing on the grid area. It ensures that clicks outside the designated grid area are ignored and facilitates the process of selecting and modifying cells based on user input.

Implementation of Mouse Click Detection

4.8.1 Defining the Function

- The `get_mouse_click` function is responsible for detecting mouse clicks within the grid area of the game window and identifying the corresponding cell coordinates.

4.8.2 Setting the Detection Range

- The function limits click detection to a specific area of the screen, defined as (900, 900) within the overall screen size of (1200, 900). This ensures that only clicks within the grid area are processed.
- The condition `x <= 900` is used to restrict click detection to the grid area.

4.8.3 Calculating Cell Coordinates

- When a mouse click is detected within the specified range, the function calculates the exact cell coordinates based on the click

position. This involves converting the pixel coordinates of the click to the corresponding cell coordinates in the 9x9 grid.

```
def get_mouse_click(self, x: int, y: int) -> None:
    if x <= 900:
        grid_x, grid_y = x // 100, y // 100
        #print(grid_x, grid_y)
        if not self.is_cell_preoccupied(grid_x, grid_y):
            self.set_cell([grid_x, grid_y, -1])
```

4.8.4 Checking Cell Occupancy

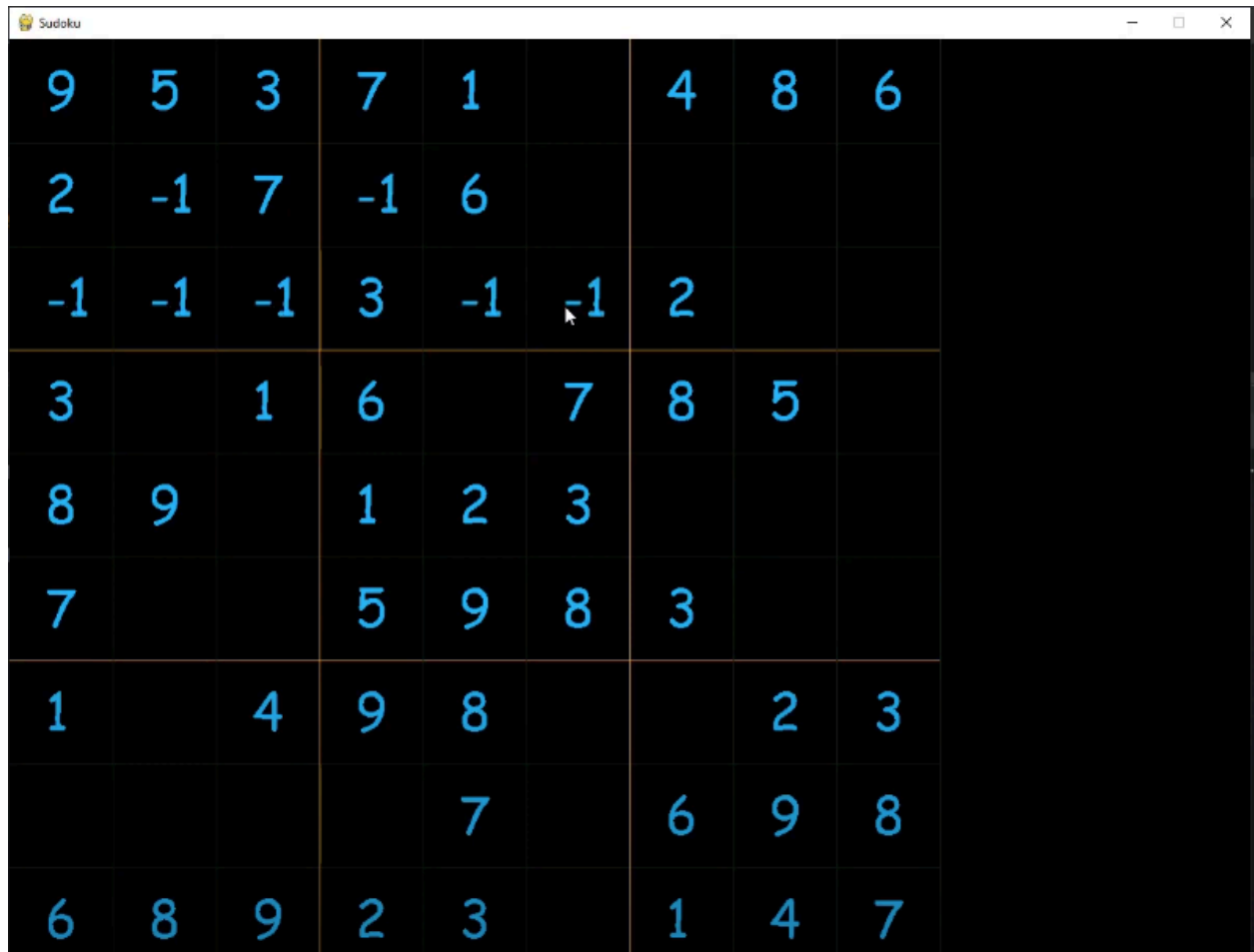
- The function uses `self.is_cell_preoccupied` to check if the clicked cell is preoccupied. This function returns a boolean value indicating whether the cell contains an initial value that should not be altered by the player.
- If the cell is not preoccupied, `self.set_cell` sets the cell value to -1, indicating that it is selected by the player.

```
def is_cell_preoccupied(self, x: int, y: int) -> bool:
    # check for non playable cells
    for cell in self.occupied_cell_cordinated:
        if x == cell[1] and y == cell[0]: # x is column, y is row
            return True
    return False
```

In this implementation:

- The condition `x <= 900` ensures that only clicks within the grid area are processed.
- `grid_x` and `grid_y` are calculated by dividing the click coordinates by the cell size (100), converting pixel coordinates to grid coordinates.

- `self.is_cell_preoccupied(grid_y, grid_x)` checks if the cell is preoccupied. If not, `self.set_cell(grid_y, grid_x, -1)` sets the cell value to -1, indicating selection.



4.9 Number Selection

The Number Selection feature in the Sudoku Game Project is an integral part of the user interface, enabling players to choose numbers they want to place on the Sudoku grid. This functionality is encapsulated within the `SelectNumber` class, which defines buttons for each number and manages their appearance and

behavior. Here's an in-depth explanation of how the number selection feature is implemented:

SelectNumber Class

The `SelectNumber` class is responsible for creating and handling the selection buttons that players use to choose numbers. This class includes properties for button dimensions, colors, positions, and a function to draw the buttons on the screen.

Properties of SelectNumber Class

4.9.1 Button Dimensions

- `self.btn_w` and `self.btn_h`: These properties define the width and height of each selection button. Both are set to 80 pixels to create large, easily clickable buttons.

4.9.2 Button Colors

- `self.color_selected`: This property defines the color of a button when it is selected. It is set to green `(0, 255, 0)` to provide a clear visual indication of selection.
- `self.color_normal`: This property defines the normal (unselected) color of the buttons, set to a light gray `(200, 200, 200)` for a neutral appearance.

4.9.3 Button Positions

- **self.btn_positions**: This property is a list of tuples, each representing the (x, y) coordinates of the buttons on the screen. The positions are carefully chosen to arrange the buttons in a grid-like layout on the right side of the game window.

```
class SelectNumber:
    def __init__(self, pygame, font):
        self.pygame = pygame
        self.btn_w = 80 #button width
        self.btn_h = 80 #button height
        self.my_font = font
        self.selected_number = 0

        self.color_selected = (0,255,0)
        self.color_normal = (200,200,200)

        self.btn_positions = [(950,50), (1050, 50),
                               (950, 150), (1050, 150),
                               (950, 250), (1050, 250),
                               (950, 350), (1050, 350),
                               (1050, 450)]
```

Drawing the Buttons

4.9.4 Function **draw**

- **Purpose**: The **draw** function is responsible for rendering the selection buttons on the screen. It iterates through the list of button

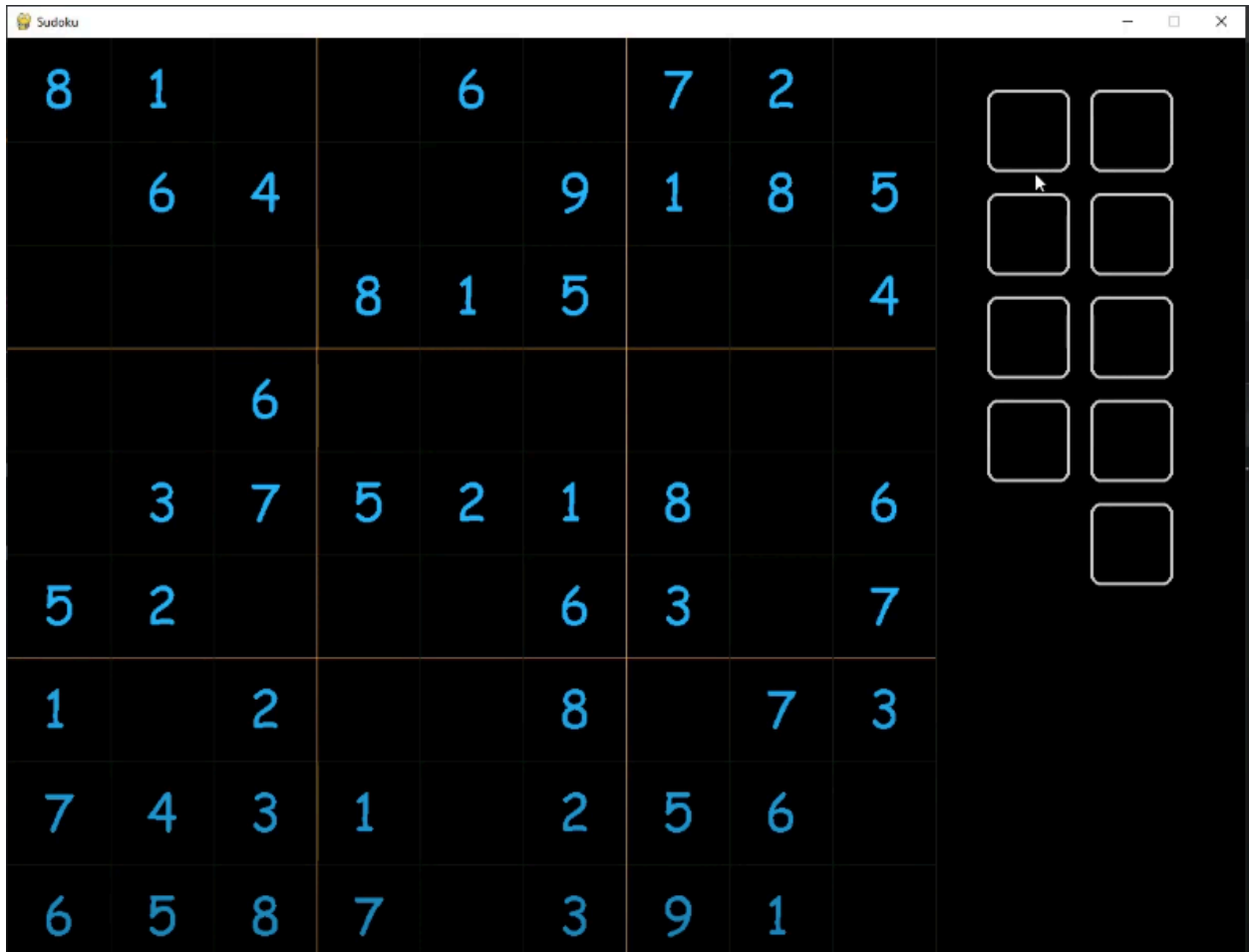
positions and draws each button with the appropriate color and dimensions.

- **Implementation:** The function uses Pygame's `draw.rect` method to draw rectangles representing the buttons. It sets the button color based on whether the button is selected or not and draws them at the specified positions.

```
def draw(self, pygame, surface):
    for index, pos in enumerate(self.btn_positions):
        pygame.draw.rect(surface, self.color_normal, [pos[0], pos[1], self.btn_w, self.btn_h], width=3, border_radius=10)
```

In this function:

- **selected_number:** This parameter indicates which number is currently selected by the player.
- The function iterates through `self.btn_positions`, using `enumerate` to get both the index and position of each button.
- It sets the button color to `self.color_selected` if the button corresponds to the selected number; otherwise, it uses `self.color_normal`.
- `pygame.draw.rect` draws the button, and `screen.blit` displays the number on the button.



4.10 Button Hover

The Button Hover feature in the Sudoku Game Project provides visual feedback to the player by changing the button color when the mouse cursor is over it. This interactivity is achieved through two functions: `on_button` and `button_hover`.

4.10.1 `on_button` Function:

- This function checks if the mouse cursor is on any of the buttons created.

- It returns a boolean value indicating whether the cursor is over a button.

```
def on_button(self, mouse_x: int, mouse_y: int, pos: tuple) -> bool:
    return pos[0] < mouse_x < pos[0] + self.btn_w and pos[1] < mouse_y < pos[1] + self.btn_h
```

4.10.2 **button_hover** Function:

- This function uses **on_button** to determine if the mouse is over a specific area.
- It returns **true** if the cursor is over the button, enhancing the user interaction.

```
def button_hover(self, pos: tuple) -> bool|None:
    mouse_pos = self.pygame.mouse.get_pos()
    if self.on_button(mouse_pos[0], mouse_pos[1], pos):
        return True
```

4.10.3 Updating the **draw** Function:

- The **draw** function is updated to change the button's color to green when the mouse is over it.
- If the mouse is not over the button, it remains the normal color.

```
def draw(self, pygame, surface):
    for index, pos in enumerate(self.btn_positions):
        pygame.draw.rect(surface, self.color_normal, [pos[0], pos[1], self.btn_w, self.btn_h], width=3, border_radius=10)
        # checking for mouse hover
        if self.button_hover(pos):
            pygame.draw.rect(surface, self.color_selected, [pos[0], pos[1], self.btn_w, self.btn_h], width=3, border_radius=10)
            text_surface = self.my_font.render(str(index + 1), False, (0, 255, 0))
        else:
            text_surface = self.my_font.render(str(index + 1), False, self.color_normal)
```

This feature ensures that players receive immediate visual feedback, making the game more interactive and engaging.



4.11 Button Click

The Button Click feature in the Sudoku Game Project allows players to select numbers by clicking on buttons. This functionality is achieved through the `button_clicked` function, which utilizes `self.on_button` to determine if a button has been clicked.

4.11.1 `button_clicked` Function:

- Responsible for handling the logic when a button is clicked.
- Uses `self.on_button` to check if a specific button is clicked and selects the corresponding number.

```
def button_clicked(self, mouse_x: int, mouse_y: int) -> None:
    for index, pos in enumerate(self.btn_positions):
        if self.on_button(mouse_x, mouse_y, pos):
            self.selected_number = index + 1
```

4.11.2 Integration with `get_mouse_click`:

- The `button_clicked` function is integrated into the `get_mouse_click` function to respond to mouse click events.
- Ensures that clicking on a button properly registers the selection.

```
def get_mouse_click(self, x: int, y: int) -> None:
    if x <= 900:
        grid_x, grid_y = x // 100, y // 100
        #print(grid_x, grid_y)
        if not self.is_cell_preocc (variable) grid_y: int
            self.set_cell(grid_x, grid_y, self.selection.selected_number)
        self.selection.button_clicked(x,y)
```

4.11.3 Updating the `draw` Function:

- The `draw` function is updated to maintain the button color as green when it is selected.
- Provides visual feedback to indicate the currently selected button.

This feature ensures that players can easily select numbers by clicking on buttons, with clear visual feedback indicating their selection.

```
if self.selected_number > 0:
    if self.selected_number - 1 == index:
        pygame.draw.rect(surface, self.color_selected, [pos[0], pos[1], self.btn_w, self.btn_h],
                           width=3, border_radius=10)
        text_surface = self.my_font.render(str(index + 1), False, self.color_selected)
        surface.blit(text_surface, (pos[0] + 30, pos[1] + 25))
```

4.12 Number Coloring

The Number Coloring feature in the Sudoku Game Project enhances gameplay by providing visual feedback on the correctness of the numbers placed by the player. This functionality is implemented through an updated `draw_number` function.

4.12.1 Correct and Incorrect Number Coloring:

- If the correct number is placed in an empty cell, its color turns green.
- If an incorrect number is placed in the cell, its color turns red.

```
if (y, x) in self.occupied_cell_cordinated:
    text_surface = self.game_font.render(str(self.get_cell(x, y)), False, (0, 200, 255))
else:
    text_surface = self.game_font.render(str(self.get_cell(x, y)), False, (0, 255, 0))

if self.get_cell(x, y) != self.__test_grid[y][x]: #self.get_cell(x, y) != self.__test_grid[y][x]
    text_surface = self.game_font.render(str(self.get_cell(x,y)), False, (255, 0, 0))
surface.blit(text_surface, (x * self.cell_size + self.num_x_offset, y * self.cell_size + self.num_y_of
```

4.12.2 Grid Copy for Validation:

- A copy of the original grid numbers is created before removing some for gameplay.

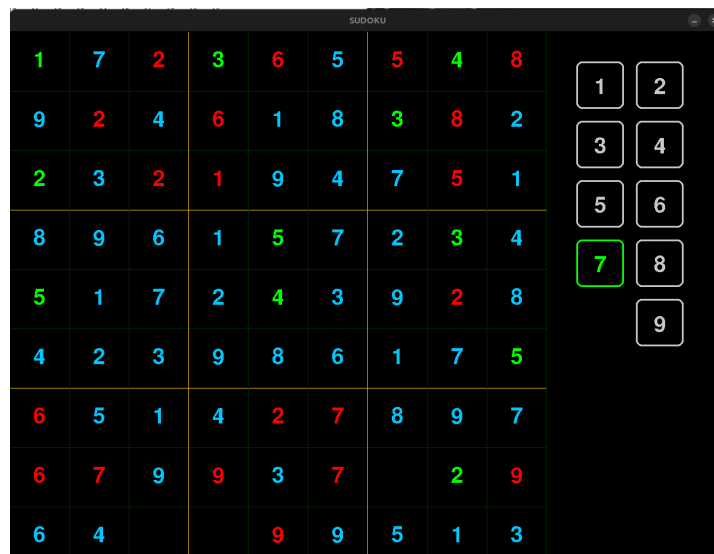
- This copy is used to validate the numbers entered by the player, ensuring accurate feedback.

```
class Grid:
    def __init__(self, pygame, font):
        self.cell_size = 100
        self.num_x_offset = 40
        self.num_y_offset = 30
        self.line_coordinates = create_line_coordinates(self.cell_size)
        self.win = False
        self.game_font = font

        self.grid = create_grid(SUB_GRID_SIZE)
        self.__test_grid = deepcopy(self.grid) # create a copy before re

        #print(self.__test_grid)
        remove_numbers(self.grid)
        self.occupied_cell_cordinated = self.pre_occupied_cells()
        #print(self.occupied_cell_cordinated)
```

This feature improves the user experience by immediately indicating whether the player's input is correct or incorrect, making the game more interactive and engaging.



4.13 Check Grids

The Check Grids feature in the Sudoku Game Project ensures that the player's progress is continually evaluated for completion. This is implemented through the `check_grids` function.

4.13.1 Function `check_grids`:

- Compares each cell in the main grid with the corresponding cell in the test grid.
- After each mouse click, it verifies if the two grids are identical.

```
def check_grids(self):  
    #checks if all the cells in the main grid and the test grid are equal  
    for y in range(len(self.grid)):  
        for x in range(len(self.grid[y])):  
            if self.grid[y][x] != self.__test_grid[y][x]:  
                return False  
    return True
```

4.13.2 Completion Indicator:

- Introduces a `self.win` variable.
- This boolean variable indicates whether the game is complete.
- If all cells match, `self.win` is set to `True`, signaling that the player has successfully completed the game.

```
if self.check_grids():  
    print("Won, Game Over!!!")  
    self.win = True
```

This feature allows for continuous checking of the player's progress, providing immediate feedback on game completion.

4.14 Draw Text

The Draw Text feature in the Sudoku Game Project enhances the end-game experience by displaying congratulatory and instructional messages when the player wins.

4.14.1 Winning Message:

- When the `check_grids` function confirms that the grids match, indicating a win, a "You won" message is displayed.
- This text is rendered at coordinates (950, 650) on the screen in green (0, 255, 0).

4.14.2 Restart Instruction:

- An additional message, "Press space to restart!", is displayed to guide the player on how to restart the game.
- This text is rendered at coordinates (950, 750) in color (0, 255, 200).

```
if grid.win:
    won_surface = game_font.render("YOU WON", False, (0, 255, 0))
    surface.blit(won_surface, dest=(950, 650))

    press_space_surf = game_font2.render("Press Space to restart!", False, (0, 255, 200))
    surface.blit(press_space_surf, dest=(920, 750))
```

These messages provide clear and immediate feedback upon game completion, enhancing the overall user experience.



4.15 Restart Game

The Restart Game feature in the Sudoku Game Project allows players to reset the game and start over. This functionality is implemented through the `restart` function, which reinitializes the game state.

1. Grid Creation:

- The function first creates a new Sudoku grid.

2. Grid Copy:

- A copy of the newly created grid is made to retain the original state.

3. Remove Numbers:

- Randomly removes numbers from the grid to create the puzzle.

4. Preoccupied Cells:

- Identifies and stores the coordinates of preoccupied cells.

5. Reset Win State:

- The `win` variable is set to `false` to indicate that the game is not yet completed.

```
def restart(self) -> None:
    self.grid = create_grid(SUB_GRID_SIZE)
    self.__test_grid = deepcopy(self.grid)
    remove_numbers(self.grid)
    self.occupied_cell_cordinated = self.pre_occupied_cells()
    self.win = False
```

By repeating these steps, the `restart` function effectively resets the game, allowing players to enjoy a new puzzle each time they restart.

```
while running:
    # check for input events
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        if event.type == pygame.MOUSEBUTTONDOWN and not grid.win:
            if pygame.mouse.get_pressed()[0]:
                pos = pygame.mouse.get_pos()
                grid.get_mouse_click(pos[0], pos[1])
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_SPACE and grid.win:
                grid.restart()
```

SUDOKU

4	6	8	7	2	9	3	1	5
3	1	5	4	8	6	7	9	2
7	9	2	3	5	1	4	6	8
6	8	3	9	4	2	1	5	7
9	2	4	1	7	5	6	8	3
1	5	7	6	3	8	9	2	4
5	7	9	8	1	3	2	4	6
8	3	1	2	6	4	5	7	9
2	4	6	5	9	7	8	3	1

1

2

3

4

5

6

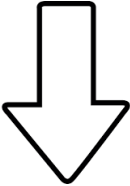
7

8

9

YOU WON

Press Space to restart!



SUDOKU

7		5	1		9		4	
3	9	1	8			5	6	
2			5	7		1	9	3
8			6			9	7	1
5	2		9				3	
1				8	3		2	5
	5	7	3			2	8	6
6				9	5		1	4
	1		2	6	8	7		

1

2

3

4

5

6

7

8

9

4.16 Code

```
import pygame
import os
from grid import Grid

# set the window position relative to the screen upper left corner
os.environ["SDL_VIDEO_WINDOW_POS"] = "%d,%d" % (400, 100)

# create the window surface and set the window caption
surface = pygame.display.set_mode((1200, 900))
pygame.display.set_caption(["SUDOKU"])

pygame.font.init()
game_font = pygame.font.SysFont(name= 'Comfortaa', size= 55)
game_font2 = pygame.font.SysFont(name= 'Comfortaa', size= 35)

#the game loop
grid = Grid(pygame, game_font)
running = True

while running:

    # check for input events
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        if event.type == pygame.MOUSEBUTTONDOWN and not grid.win:
            if pygame.mouse.get_pressed()[0]:
                pos = pygame.mouse.get_pos()
                grid.get_mouse_click(pos[0], pos[1])
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_SPACE and grid.win:
                grid.restart()

    #clear the window surface to black
    surface.fill((0, 0, 0))

    #draw the grid here
    grid.draw_all(pygame, surface)

    if grid.win:
        won_surface = game_font.render("YOU WON", False, (0, 255, 0))
        surface.blit(won_surface, dest=(950, 650))

        press_space_surf = game_font2.render("Press Space to restart!", False, (0, 255, 200))
        surface.blit(press_space_surf, dest=(920,750))
    #update the window surface
    pygame.display.flip()
```

Game.py

```

class SelectNumber:
    def __init__(self, pygame, font):
        self.pygame = pygame
        self.btn_w = 80 #button width
        self.btn_h = 80 #button height
        self.my_font = font
        self.selected_number = 0

        self.color_selected = (0,255,0)
        self.color_normal = (200,200,200)

        self.btn_positions = [(950,50), (1050, 50),
                               (950, 150), (1050, 150),
                               (950, 250), (1050, 250),
                               (950, 350), (1050, 350),
                               (1050, 450)]

    def draw(self, pygame, surface):
        for index, pos in enumerate(self.btn_positions):
            pygame.draw.rect(surface, self.color_normal, [pos[0], pos[1],
                                                           self.btn_w, self.btn_h], width=3, border_radius=10)

            # checking for mouse hover
            if self.button_hover(pos):
                pygame.draw.rect(surface, self.color_selected, [pos[0], pos[1],
                                                                  self.btn_w, self.btn_h], width=3, border_radius=10)
                text_surface = self.my_font.render(str(index + 1), False, (0, 255, 0))
            else:
                text_surface = self.my_font.render(str(index + 1), False, self.color_normal)

            #check if a number was selected , then draw it green
            if self.selected_number > 0:
                if self.selected_number - 1 == index:
                    pygame.draw.rect(surface, self.color_selected, [pos[0], pos[1],
                                                                      self.btn_w, self.btn_h], width = 3, border_radius = 10)
                    text_surface = self.my_font.render(str(index + 1), False, self.color_selected)
                    surface.blit(text_surface, (pos[0]+ 30, pos[1]+ 25))

    def button_clicked(self, mouse_x: int, mouse_y: int) -> None:
        for index, pos in enumerate(self.btn_positions):
            if self.on_button(mouse_x, mouse_y, pos):
                self.selected_number = index + 1

    def button_hover(self, pos: tuple) -> bool|None:
        mouse_pos = self.pygame.mouse.get_pos()
        if self.on_button(mouse_pos[0], mouse_pos[1], pos):
            return True

    def on_button(self, mouse_x: int, mouse_y: int, pos: tuple) -> bool:
        return pos[0] < mouse_x < pos[0] + self.btn_w and pos[1] < mouse_y < pos[1] + self.btn_h

```

Selection.py

```

from random import sample
from selection import SelectNumber
from copy import deepcopy

def create_line_coordinates(cell_size: int) -> list[list[tuple]]:
    #creates the x,y coordinates for drawing the grid lines.
    points = []
    for y in range(1,9):
        #horizontal lines
        temp = []
        temp.append((0, y * cell_size)) # x,y points [(0, 100), (0, 200), .....]
        temp.append((900, y * cell_size)) # x,y points [(900, 100), (900, 200), .....]
        points.append(temp)

    for x in range(1,10):
        # vertical lines - from 1 to 10, to close the grid on the right side
        temp = []
        temp.append((x * cell_size, 0)) #x,y points [(100,0), (200,0), .....]
        temp.append((x * cell_size, 900)) # x,y points [(100, 900), (200,900), .....]
        points.append(temp)
    #print(points)
    return points

SUB_GRID_SIZE = 3
GRID_SIZE = SUB_GRID_SIZE * SUB_GRID_SIZE

def pattern(row_num: int, col_num: int) -> int:
    return(SUB_GRID_SIZE * (row_num % SUB_GRID_SIZE) + row_num//SUB_GRID_SIZE + col_num) % GRID_SIZE

def shuffle(samp: range) -> list:
    return sample(samp, len(samp))

def create_grid(sub_grid: int) -> list[list]:
    #create the 9x9 grid filled with random numbers
    row_base = range(sub_grid)
    rows = [g * sub_grid + r for g in shuffle(row_base) for r in shuffle(row_base)]
    cols = [g * sub_grid + c for g in shuffle(row_base) for c in shuffle(row_base)]
    nums = shuffle(range(1, sub_grid * sub_grid + 1))
    return [[nums[pattern(r, c)] for c in cols] for r in rows]

def remove_numbers(grid: int) -> None:
    # randomly sets numbers to 0 on the grid
    num_of_cells = GRID_SIZE * GRID_SIZE
    empties = num_of_cells * 3 // 7 # 7 is ideal - higher this number means easier game
    for i in sample(range(num_of_cells), empties):
        grid[i // GRID_SIZE][i % GRID_SIZE] = 0

class Grid:
    def __init__(self, pygame, font):
        self.cell_size = 100
        self.num_x_offset = 40
        self.num_y_offset = 30
        self.line_coordinates = create_line_coordinates(self.cell_size)
        self.win = False
        self.game_font = font

        self.grid = create_grid(SUB_GRID_SIZE)
        self.__test_grid = deepcopy(self.grid) # create a copy before removing numbers

```

```

self.__test_grid = deepcopy(self.grid) # create a copy before removing numbers
|
|
|
# print(self.__test_grid)
remove_numbers(self.grid)
self.occupied_cell_cordinated = self.pre_occupied_cells()
# print(self.occupied_cell_cordinated)

self.selection = SelectNumber(pygame, self.game_font)

def restart(self) -> None:
    self.grid = create_grid(SUB_GRID_SIZE)
    self.__test_grid = deepcopy(self.grid)
    remove_numbers(self.grid)
    self.occupied_cell_cordinated = self.pre_occupied_cells()
    self.win = False

def check_grids(self):
    # checks if all the cells in the main grid and the test grid are equal
    for y in range(len(self.grid)):
        for x in range(len(self.grid[y])):
            if self.grid[y][x] != self.__test_grid[y][x]:
                return False
    return True

def is_cell_preoccupied(self, x: int, y: int) -> bool:
    # check for non playable cells
    for cell in self.occupied_cell_cordinated:
        if x == cell[1] and y == cell[0]: # x is column, y is row
            return True
    return False

def get_mouse_click(self, x: int, y: int) -> None:
    if x <= 900:
        grid_x, grid_y = x // 100, y // 100
        # print(grid_x, grid_y)
        if not self.is_cell_preoccupied(grid_x, grid_y):
            self.set_cell(grid_x, grid_y, self.selection.selected_number)
    self.selection.button_clicked(x,y)
    if self.check_grids():
        print("Won, Game Over!!!")
        self.win = True

def pre_occupied_cells(self) -> list[tuple]:
    # gather the u and x coordinates for all preoccupied cells
    occupied_cell_coordinates = []
    for y in range(len(self.grid)):
        for x in range(len(self.grid[y])):
            if self.get_cell(x, y) != 0:
                occupied_cell_coordinates.append((y, x)) # first the row, then the column: y,x
    return occupied_cell_coordinates

def _draw_lines(self, pg, surface) -> None:
    for index, point in enumerate(self.line_coordinates):
        pg.draw.line(surface, (0, 50, 0), point[0], point[1])
        if index == 2 or index == 5 or index == 10 or index == 13:

```



```

def pre_occupied_cells(self) -> list[tuple]:
    occupied_cell_coordinates = []
    for y in range(len(self.grid)):
        for x in range(len(self.grid[y])):
            if self.get_cell(x, y) != 0:
                occupied_cell_coordinates.append((y, x)) # first the row, then the column: y,x
    return occupied_cell_coordinates

def _draw_lines(self, pg, surface) -> None:
    for index, point in enumerate(self.line_coordinates):
        pg.draw.line(surface, (0, 50, 0), point[0], point[1])
        if index == 2 or index == 5 or index == 10 or index == 13:
            pg.draw.line(surface, (255, 200, 0), point[0], point[1])
        else:
            pg.draw.line(surface, (0, 50, 0), point[0], point[1])

def _draw_numbers(self, surface) -> None:
    #Draw the grid numbers:
    for y in range(len(self.grid)):
        for x in range(len(self.grid[y])):
            if self.get_cell(x,y) != 0:
                if (y, x) in self.occupied_cell_cordinated:
                    text_surface = self.game_font.render(str(self.get_cell(x, y)), False, (0, 200, 255))
                else:
                    text_surface = self.game_font.render(str(self.get_cell(x, y)), False, (0, 255, 0))

                if self.get_cell(x, y) != self.__test_grid[y][x]: #self.get_cell(x, y) != self.__test_grid[y][x]
                    text_surface = self.game_font.render(str(self.get_cell(x,y)), False, (255, 0, 0))
            surface.blit(text_surface, (x * self.cell_size + self.num_x_offset, y * self.cell_size + self.num_y_offset))

def draw_all(self, pg, surface):
    self._draw_lines(pg, surface)
    self._draw_numbers(surface)
    self.selection.draw(pg, surface)

def get_cell(self, x: int, y: int) -> int:
    # get a cell valor at y, x coordinates
    return self.grid[y][x]

def set_cell(self, x: int, y: int, value: int) -> None:
    # set a cell value at y, x coordinates
    self.grid[y][x] = value

def show(self):
    for cell in self.grid:
        print(cell)

if __name__ == "__main__":
    grid = Grid()
    grid.show()

```

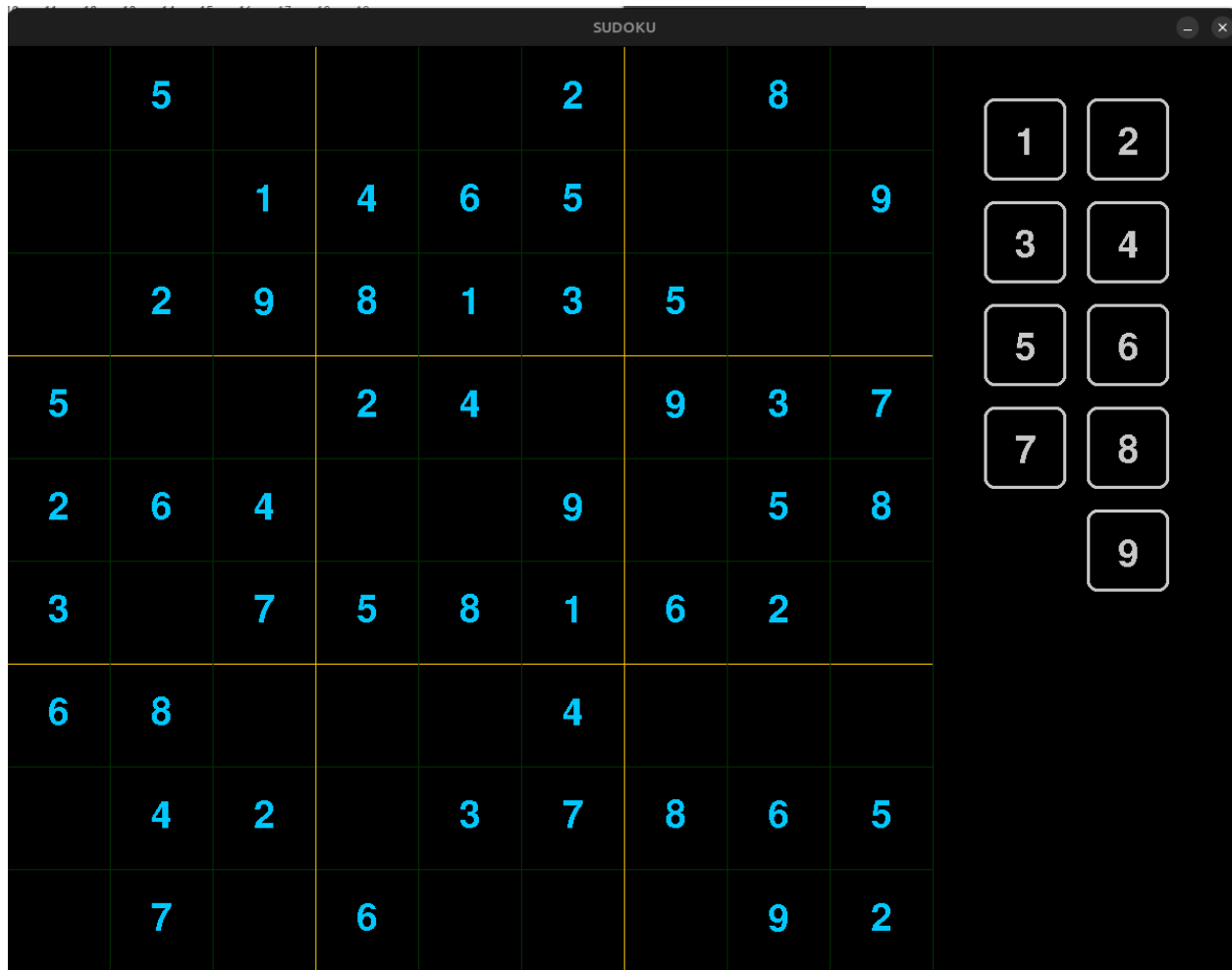
grid.py

CHAPTER 5

RESULTS

5.1 Game window

Firstly there pops a game window having an unsolved game of sudoku in the left and digits from 1 to 9 to its right from where we can select the numbers and put it in the empty cells



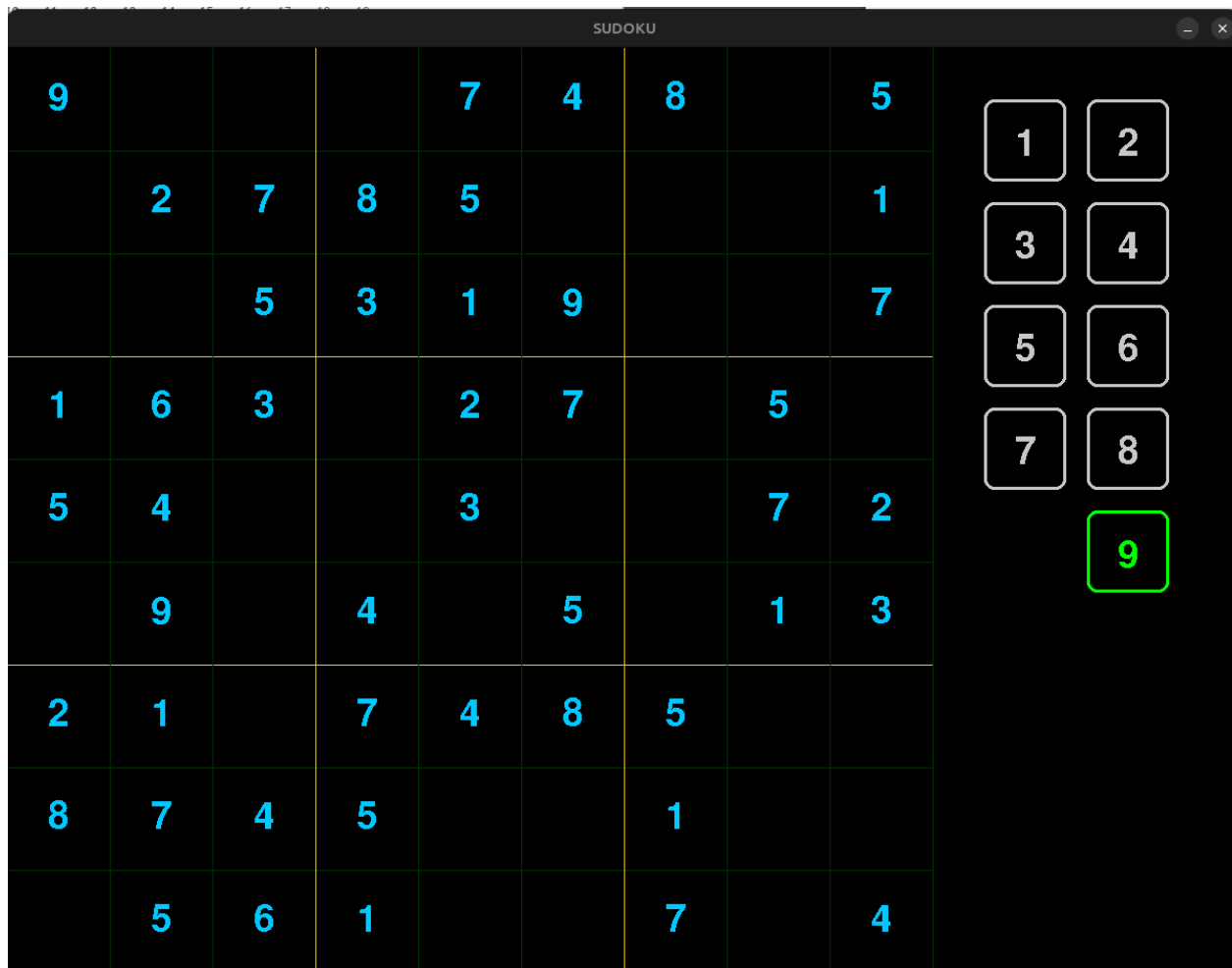
5.2 Solving the game

After the game window appears the user can fill the empty cells by using the buttons to the right of the game. If the number is placed in the correct cell then the number color appears to be green whereas if the number placed is in the incorrect cell then the number color appears to be red. When the game ends then the congratulatory message appears in the bottom-right of the game window. And that message also asks for the restart of the game.



5.3 Restart of the game

When the game is finished then the game asks the user to restart the game by saying “Press space to restart”. When the user presses the spacebar on the keyboard then the game starts over again with the new puzzle and the congratulatory message disappears.



CHAPTER 6

CONCLUSION

The Sudoku Game Project developed using Python and Pygame successfully combines engaging gameplay with a user-friendly interface. This project demonstrates various programming concepts, including game logic, event handling, and visual feedback mechanisms. The game offers an interactive experience where players can fill in the grid, receiving immediate visual cues for correct and incorrect inputs. Upon completing the puzzle correctly, the game provides a congratulatory message and an option to restart by pressing the spacebar. This project not only provides an enjoyable way to play Sudoku but also serves as a robust example of using Python and Pygame for game development. Through this project, we have effectively created a digital version of the classic Sudoku puzzle, enhancing both the challenge and enjoyment for players.

CHAPTER 6

REFERENCES

[1]. Pygame Documentation provides comprehensive information on the Pygame library, which was used extensively in this project for handling graphics, input events, and game logic.- URL: <https://www.pygame.org/docs/>

[2]. Python Standard Library Documentation: This resource offers detailed documentation on Python's standard libraries, including `os`, `random`, and `copy` modules utilized in the project.- URL: <https://docs.python.org/3/library/>

[3]. Sudoku Puzzle Algorithms: An in-depth look at algorithms used to generate and solve Sudoku puzzles, which provided the theoretical foundation for the grid generation and validation logic.

URL: <https://stackoverflow.com/questions/45471152/how-to-create-a-sudoku-puzzle-in-python>

[4]. Sudoku Explained: How to Solve Sudoku Puzzles:- This resource explains different techniques and strategies for solving Sudoku puzzles, aiding in the understanding of how to implement and validate Sudoku logic in the project.
-URL: <https://www.sudokuoftheday.com/how-to-play/sudoku-explained/>