Classic Game Development

Python Game Development Record

Name: Le Zhang

Update Date: Sept 5 th, 2023

Contents

1 Four Connect **1**
1.1 Module Structure . . . 1
1.2 Game Logic . . . 2
1.2.1 Drop Piece . . . 3
1.2.2 Winner . . . 3
1.2.3 Is Finish . . . 5
1.3 Display . . . 5
1.3.1 Draw Current Player . . . 6
1.3.2 Draw Board . . . 6
1.3.3 Show Message . . . 7
1.4 Main Module . . . 7
1.4.1 main.py . . . 9

Four Connect

1.1. Module Structure

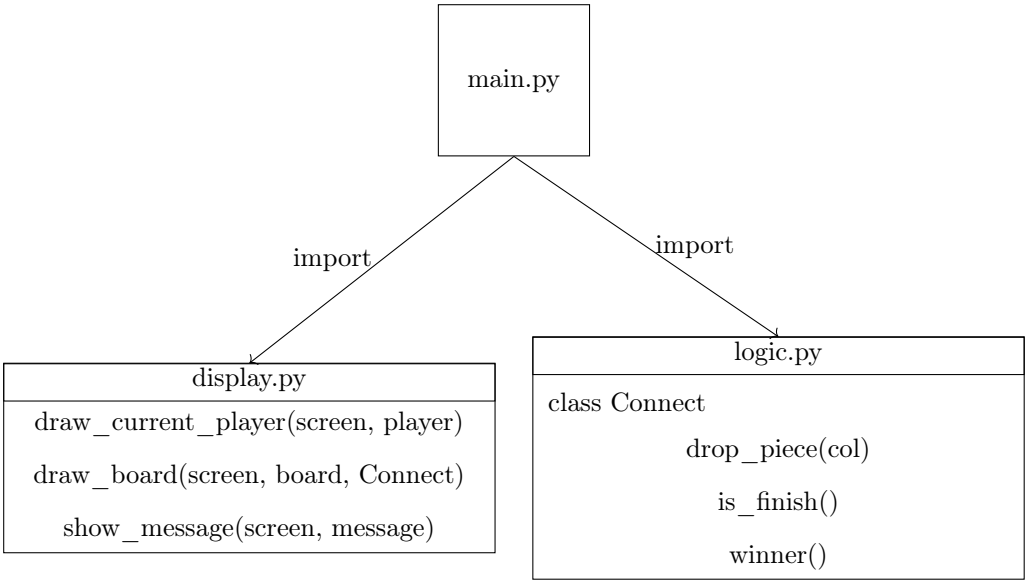
This section will introduce the module structure and the relation between them.

There are three module in this program.

- main.py
- display.py
- logic.py

The calling logic between modules is as follows. main is the client, which is used to execute the overall game loop and event processing. **`main.py`** will import the remaining two modules, **`logic.py`** and **`display.py`**. Just like the names of these two modules, **`logic.py`** will handle the execution logic in the game. In the four connect will handle the way the pieces fall, and check if the game is over and who the winner is or end game with draw. **`display.py`** will display the received data.

The following diagram indecates the relation between three modules.



There are more class methods in **`logic.py`** but will not be provided to **`main.py`**. The **`logic.py`** is the core module of this game implantation. The next section will dicuss the **`logic.py`** in detail.

1.2. Game Logic

logic.py

This module has one python class named **Connect**.

- **WINLENGTH**: Specifies the number of consecutive pieces needed to win the game, which is 4 in a standard Connect 4 game.
- **WIDTH**: Specifies the width of the game board, which is 7 columns.
- **HEIGHT**: Specifies the height of the game board, which is 6 rows.

```
class Connect:
    WINLENGTH = 4
    WIDTH = 7
    HEIGHT = 6

    def __init__(self):
        self.current_player = 1
        self.board = [0] * self.WIDTH * self.HEIGHT

    # ---continue---
```

In the initialization process, set the current player to **1**. In this game, **0** means no player, **1** means player **1** (red), and **2** means player **2** (yellow).

- **0**: NONE player
- **1**: player **1** (red)
- **2**: player **2** (yellow)

For the storage of game data, a one-dimensional array (list) will be used in this implementation. Store two-dimensional data into this one-dimensional array through the following relationship **column + row * width of row**. This data will stores in the field **board** and initialized with zero (NONE player). The code below will help us to get and update array (list).

```
    def get_player(self, col, row):
        pos = col + row * self.WIDTH
        return self.board[pos]

    def set_player(self, col, row):
        pos = col + row * self.WIDTH
        self.board[pos] = self.current_player

    # ---continue---
```

Now we can move to one of game logic, drop piece.

1.2.1. Drop Piece

This method implements the action of dropping a piece in a column. It includes the following functionalities:

- Checks the top cell of the specified column to see if it's full.
- If it is, returns False (indicating the move is not allowed).
- If not, it finds the lowest available cell in the column and places the current player's piece there.
- Then, it switches the current player to the other player and returns True, indicating the move was successful.

drop_piece(self, col)

```
    def drop_piece(self, col):
        current_player = self.current_player
        next_player = 0
        if (current_player == 1):
            next_player = 2
        if (current_player == 2):
            next_player = 1
        top_item = self.get_player(col, 0)
        if (top_item != 0):
            return False
        else:
            for j in range(self.HEIGHT - 1, -1, -1):
                current_player = self.get_player(col, j)
                if (current_player == 0):
                    self.set_player(col, j)
                    self.current_player = next_player
                    return True
            return False

    # ---continue---
```

1.2.2. Winner

This method checks if there is a winner by examining:

1. **Rows**: Iterates through each row, checking for a sequence of **WINLENGTH** matching non-zero pieces.
2. **Columns**: Iterates through each column, doing the same as for rows.
3. **Diagonals**: Checks both upward and downward diagonals for a sequence of **WINLENGTH** matching non-zero pieces.

It returns the winning player number if found, or 0 if there is no winner yet.

winner(self)

```
    def winner(self):
        # row checktwo mo
        for i in range(self.HEIGHT):
            row_count = 1
            row_item = self.get_player(0, i)
            for j in range(1, self.WIDTH):
                current_item = self.get_player(j, i)
                if current_item == row_item:
                    row_count += 1
                    if row_count == self.WINLENGTH and row_item != 0:
                        return row_item
                else:
                    row_count = 1
                    row_item = current_item
        # col check
        for j in range(self.WIDTH):
            col_count = 1
            col_item = self.get_player(j, 0)
            for i in range(1, self.HEIGHT):
                current_item = self.get_player(j, i)
                if current_item == col_item:
                    col_count += 1
                    if col_count == self.WINLENGTH and col_item != 0:
                        return col_item
                else:
                    col_count = 1
                    col_item = current_item
        # dig check
        for i in range(self.HEIGHT):
            for j in range(self.WIDTH):
                up_dig_count = 1
                up_dig = self.get_player(j, i)
                dn_dig_count = 1
                dn_dig = self.get_player(j, i)
                for k in range(1, self.WINLENGTH):
                    # up
                    if i - k >= 0 and j + k < self.WIDTH:
                        current_item = self.get_player(j + k, i - k)
                        if current_item == up_dig:
                            up_dig_count += 1
                            if up_dig_count == self.WINLENGTH and up_dig != 0:
                                return up_dig
                        else:
                            up_dig_count = 1
                            up_dig = current_item
                    # down
                    if i + k < self.HEIGHT and j + k < self.WIDTH:
                        current_item = self.get_player(j + k, i + k)
                        if current_item == dn_dig:
                            dn_dig_count += 1
                            if dn_dig_count == self.WINLENGTH and dn_dig != 0:
                                return dn_dig
                        else:
                            dn_dig_count = 1
                            dn_dig = current_item
        return 0

    # ---continue---
```

1.2.3. Is Finish

This method checks if the game is finished, either by:

- A player winning (as determined by the `winner` method).
- A draw, which is determined by checking if the top row is full and there is no winner.

It returns **`True`** if the game is finished, and **`False`** otherwise.

`is_finish(self)`

```
    def is_finish(self):
        if (self.winner() != 0):
            return True
        top_count = 0
        for j in range(0, self.WIDTH):
            top_item = self.get_player(j, 0)
            if (top_item != 0):
                top_count += 1
        if (top_count == self.WIDTH):
            return True
        else:
            return  False

    # ---END---
```

Overall, this class encapsulates all the necessary logic for playing a game of Connect 4, providing a blueprint for creating game instances and methods to manipulate and check the game state.

1.3. Display

`display.py`

This module presents a detailed analysis of a Python script implementing a graphical interface for a Connect 4 game using the Pygame module.

First we define the size of the program window and the colors that we need to display.

```
import pygame
from pygame.locals import *

SCREEN_WIDTH = 700
SCREEN_HEIGHT = 650
CELL_SIZE = 100
WHITE = (200, 200, 200)
RED = (255, 0, 0)
YELLOW = (255, 255, 0)
BLACK = (50, 50, 50)

# ---continue---
```

Then initialize the screen with Pygame. The size of program window are defined before. We also need to display the game name.

```
pygame.init()
screen = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT))
pygame.display.set_caption('Connect 4')

# ---continue---
```

There are three functions in this module. We will introduce them in detail below.

1.3.1. Draw Current Player

This function accepts the screen object and the current player number as arguments and draws the current player information at the top of the screen.

draw_current_player(screen, player)

```
def draw_current_player(screen, player):
    font = pygame.font.SysFont(None, 36)
    text = font.render(f'Current Player: {"RED" if player == 1 else "YELLOW"}', True, WHITE)
    screen.blit(text, (20, 10))

# ---continue---
```

1.3.2. Draw Board

This function takes the screen object, the board object, and the Connect class as parameters, then draws the game board and the pieces (either red or yellow) according to the current state of the board object.

draw_board(screen, board, Connect)

```
def draw_board(screen, board, Connect):
    for row in range(Connect.HEIGHT):
        for col in range(Connect.WIDTH):
            pygame.draw.rect(screen, BLACK, (col * CELL_SIZE, row * CELL_SIZE + 50, CELL_SIZE,
            ↪  CELL_SIZE))
            pygame.draw.rect(screen, WHITE, (col * CELL_SIZE + 5, row * CELL_SIZE + 5 + 50, CELL_SIZE
            ↪   - 10, CELL_SIZE - 10))
            pygame.draw.circle(screen, WHITE,
                               (int(col * CELL_SIZE + CELL_SIZE / 2), int(row * CELL_SIZE + CELL_SIZE
                               ↪   / 2 + 50)), 40)
            char = board.get_player(col, row)
            if char == 1:
                pygame.draw.circle(screen, RED,
                                   (int(col * CELL_SIZE + CELL_SIZE / 2), int(row * CELL_SIZE +
                                   ↪  CELL_SIZE / 2 + 50)), 40)
            elif char == 2:
                pygame.draw.circle(screen, YELLOW,
                                   (int(col * CELL_SIZE + CELL_SIZE / 2), int(row * CELL_SIZE +
                                   ↪  CELL_SIZE / 2 + 50)), 40)

# ---continue---
```

1.3.3. Show Message

This function displays a message (passed as a parameter) in the center of the screen and also displays a "Continue" button below the message. The function enters a loop where it waits for the user to either close the window or click the "Continue" button to proceed.

show_message(screen, message)

```
def show_message(screen, message):
    font = pygame.font.SysFont(None, 48)
    text = font.render(message, True, WHITE)
    text_rect = text.get_rect(center=(SCREEN_WIDTH / 2, SCREEN_HEIGHT / 2 - 50))

    button_width = 200
    button_height = 60
    button_x = (SCREEN_WIDTH - button_width) / 2
    button_y = SCREEN_HEIGHT / 2
    button_rect = pygame.Rect(button_x, button_y, button_width, button_height)

    button_font = pygame.font.SysFont(None, 36)
    button_text = button_font.render("Continue", True, BLACK)
    button_text_rect = button_text.get_rect(center=button_rect.center)

    while True:
        screen.fill(BLACK)
        screen.blit(text, text_rect.topleft)

        pygame.draw.rect(screen, WHITE, button_rect)
        screen.blit(button_text, button_text_rect.topleft)

        pygame.display.flip()

        for event in pygame.event.get():
            if event.type == QUIT:
                pygame.quit()
                exit()
            if event.type == MOUSEBUTTONDOWN:
                if button_rect.collidepoint(event.pos):
                    return

# ---END---
```

1.4. Main Module

This module contains the main loop for a Connect 4 game, incorporating both game logic and graphical display. By using the **Pygame** module, it provides an interactive and visually appealing interface for playing Connect 4.

Import Modules

```
import pygame
from pygame.locals import *
from logic import *
from display import *

# ---continue---
```

This section imports the necessary modules: **pygame**, constants from **pygame.locals**, and all functions from **logic**

and **display** modules.

Game Initialization

```
game = Connect()
running = True

# ---continue---
```

Here, an instance of the ‘Connect‘ class is created to represent the game state, and a boolean variable ‘running‘ is initialized to ‘True‘ to control the main game loop.

Main Game Loop

```
while running:
    screen.fill(BLACK)

    for event in pygame.event.get():
        ...
        # ---continue---
```

The code enters a while loop, which will continue running as long as the ‘running‘ variable is ‘True‘. Inside the loop, the screen is cleared at the beginning of each iteration.

Event Handling

```
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False
        if event.type == MOUSEBUTTONDOWN:
            col = event.pos[0] // CELL_SIZE
            game.drop_piece(col)
            ...
            # ---continue---
```

Within the loop, the script handles different events such as quitting the game or mouse button clicks. If a mouse button is clicked, it calculates the column where a piece should be dropped and attempts to drop a piece in that column.

Game Finish Check and Message Display

```
            if game.is_finish():
                if game.winner() == 0:
                    show_message(screen, "The game ended with a draw.")
                else:
                    show_message(screen, f"Player {game.winner()} wins!")
                game = Connect()
                # ---continue---
```

This section checks if the game has finished (either through a draw or a win) and displays an appropriate message, then resets the game state.

Drawing Board and Current Player

```
    draw_board(screen, game, Connect)
    draw_current_player(screen, game.current_player)
    pygame.display.flip()
```

At the end of each loop iteration, the board and the current player's turn are redrawn to reflect the current game state. After all updates, the display is refreshed to show the changes.

Exiting the Game

```
pygame.quit()
```

Once the loop exits, the **Pygame** module is cleanly exited, thereby ending the program.

1.4.1. main.py

Full code

```
import pygame
from pygame.locals import *
from logic import *
from display import *


game = Connect()
running = True
while running:
    screen.fill(BLACK)

    for event in pygame.event.get():
        if event.type == QUIT:
            running = False
        if event.type == MOUSEBUTTONDOWN:
            col = event.pos[0] // CELL_SIZE
            game.drop_piece(col)
            if game.is_finish():
                if game.winner() == 0:
                    show_message(screen, "The game ended with a draw.")
                else:
                    show_message(screen, f"Player {game.winner()} wins!")
                game = Connect()

    draw_board(screen, game, Connect)
    draw_current_player(screen, game.current_player)
    pygame.display.flip()

pygame.quit()
```