第13课：解决垒墙时方块重叠问题

步骤目标

上一篇我们实现了垒墙功能，末尾讲到了方块重叠问题，如图1所示。

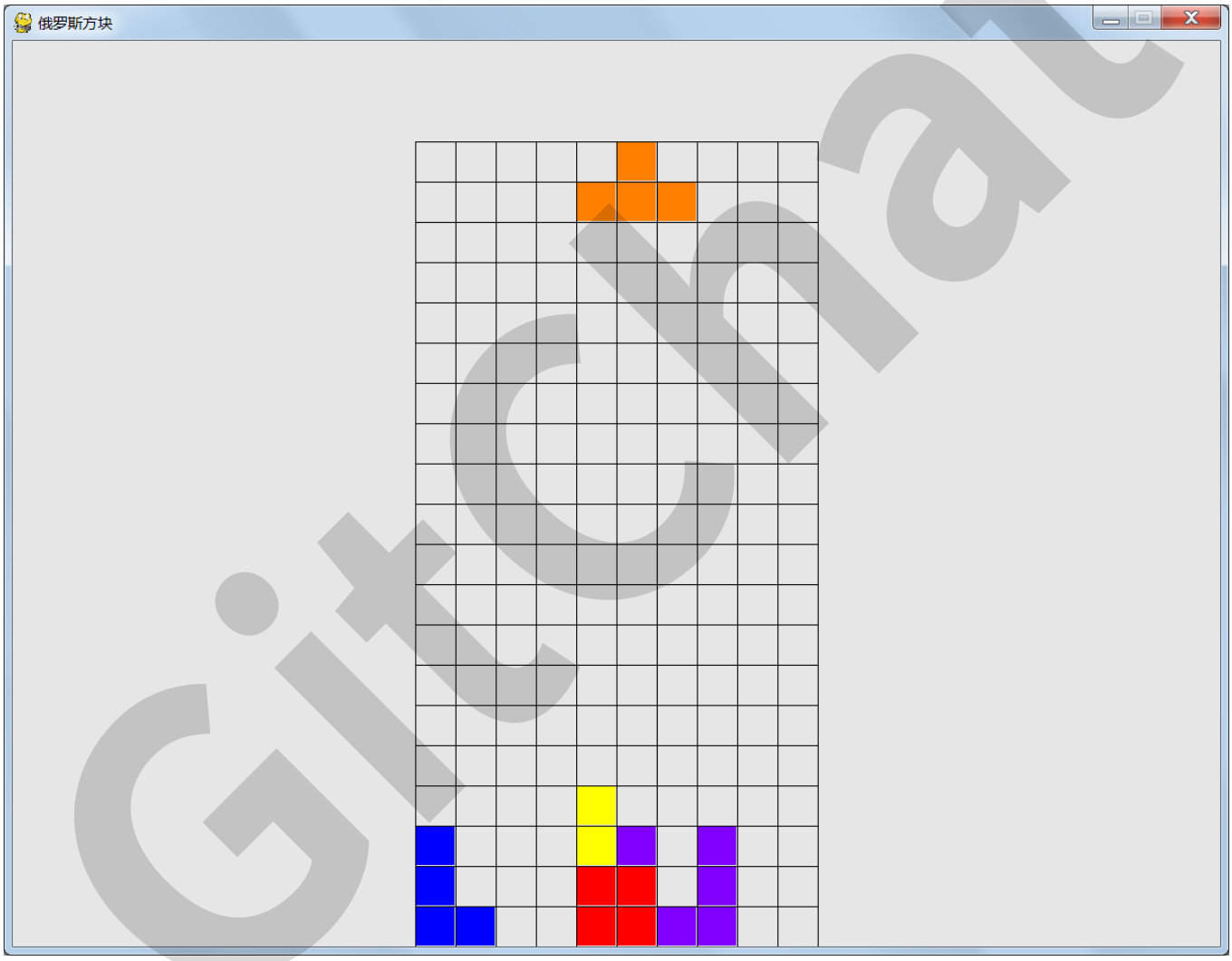

图1 方块重叠在底部几行

本文将来解决这个问题。完成本步骤后，垒墙功能就正常了，如图2所示。

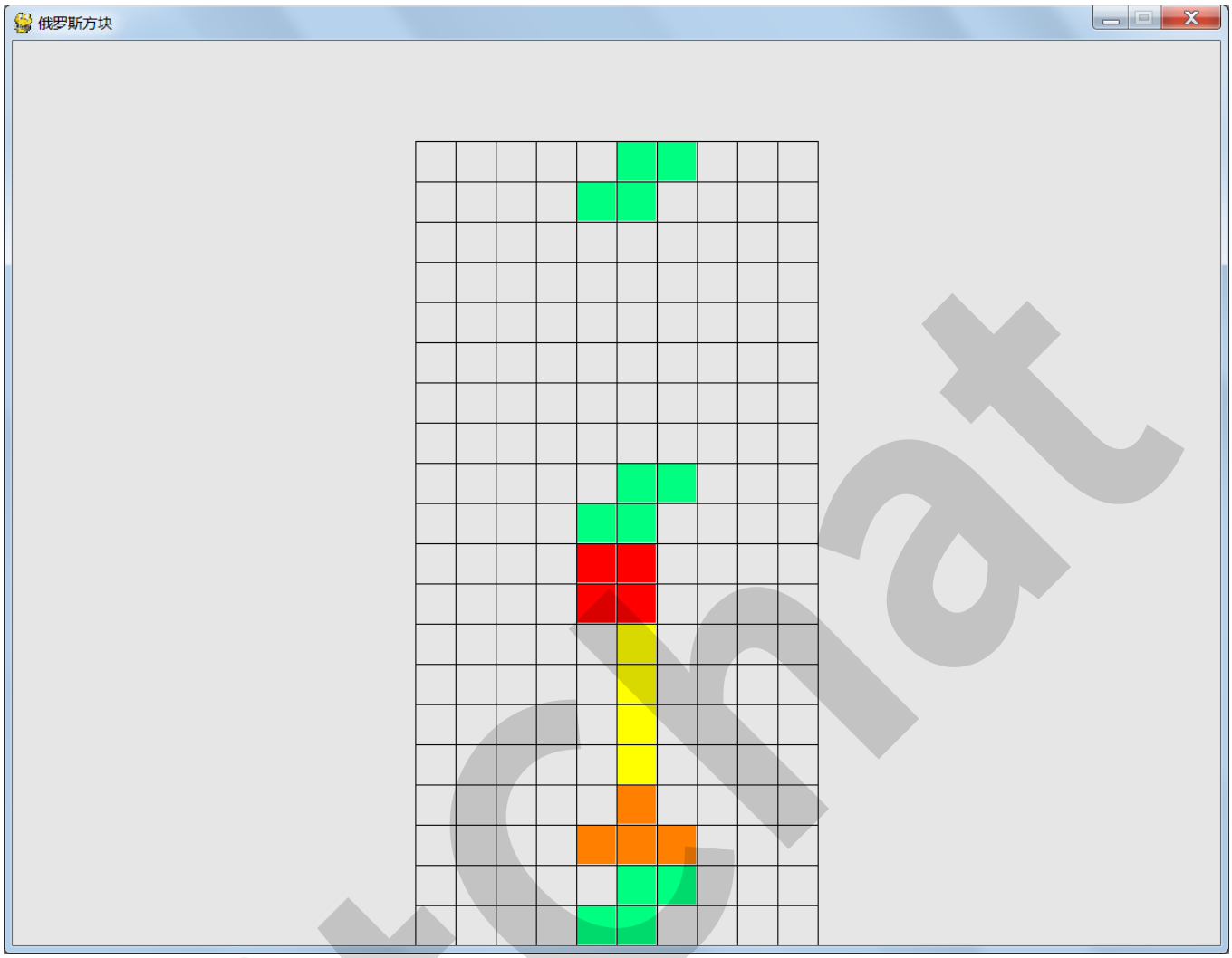

图2 垒墙不会发生方块重叠现象

在解决方块重叠问题之前，我们先来重构代码。重构代码就是修改代码，提高代码的质量，比如提高代码的可读性。

重构代码

这一次，我们重构代码的目标是在函数/方法参数中用“（行号，列号）”格式来标识一个单元格，英文写法是：`(row, column)`。在前面的版本中，有点混乱。

修改 GameDisplay 类的 `draw_cell` 方法

首先针对 GameDisplay 类的 `draw_cell` 静态方法动刀，如代码1和代码2所示。代码1是重构前，代码2是重构后。通过对比两段代码，很容易看出两者的不同。

```
@staticmethod
def draw_cell(screen, x, y, color):
    '''第y行x列的格子里填充color颜色。一种方块对应一种颜色。'''
    cell_position = (x * CELL_WIDTH + GAME_AREA_LEFT + 1,
                     y * CELL_WIDTH + GAME_AREA_TOP + 1)
    cell_width_height = (CELL_WIDTH - 2, CELL_WIDTH - 2)
    cell_rect = pygame.Rect(cell_position, cell_width_height)
    pygame.draw.rect(screen, color, cell_rect)
```

代码1 重构 `draw_cell` 方法之前

```
@staticmethod
def draw_cell(screen, row, column, color):
    '''第row行column列的格子里填充color颜色。一种方块对应一种颜色。'''
    cell_position = (column * CELL_WIDTH + GAME_AREA_LEFT + 1,
                     row * CELL_WIDTH + GAME_AREA_TOP + 1)
    cell_width_height = (CELL_WIDTH - 2, CELL_WIDTH - 2)
    cell_rect = pygame.Rect(cell_position, cell_width_height)
    pygame.draw.rect(screen, color, cell_rect)
```

代码2 重构`draw_cell方法之后`

相应地，GameDisplay 类的 `draw_wall` 方法内调用 `draw_cell` 方法的地方要作出修改，如代码3和代码4所示。

```
GameDisplay.draw_cell(game_wall.screen, c, r, PIECE_COLORS[game_wall.area[r][c]])
```

代码3 调用 `draw_cell` 方法的地方作出修改之前

```
GameDisplay.draw_cell(game_wall.screen, r, c, PIECE_COLORS[game_wall.area[r][c]])
```

代码4 调用 `draw_cell` 方法的地方作出修改之后

同样，Piece 类的 `draw_cell` 方法调用了 GameDisplay 类的 `Draw_cell` 方法，要作出修改，如代码5和代码6所示。

```
def draw_cell(self, x, y):
      GameDisplay.draw_cell(self.screen,  x,  y,
PIECE_COLORS[self.shape])
```

代码5 Piece 类的 `draw_cell` 方法中作出修改之前

```
def draw_cell(self, row, column):
       GameDisplay.draw_cell(self.screen, row, column,
PIECE_COLORS[self.shape])
```

代码6 Piece 类的 `draw_cell` 方法中作出修改之后

Piece 类的 paint 方法中调用该类的 `draw_cell` 方法的地方需要相应地作出修改，如代码7和代码8所示。

```
self.draw_cell(self.x + c, self.y + r)
```

代码7 Piece 类的 paint 方法内作出修改之前

```
self.draw_cell(self.y + r, self.x + c)
```

代码8 Piece 类的 paint 方法内作出修改之后

修改 GameWall 类的 `set_cell` 方法

GameWall 类的 `set_cell` 方法中，position 参数改为更加直观明了的 row、column 两个参数，如代码9和代码10所示。

```
def set_cell(self, position, shape_label):
    '''把第r行c列的格子打上方块记号（如S, L...），因为该格子被此方块占据。'''
    c, r = position
    self.area[r][c] = shape_label
```

代码9 GameWall 类的 `set_cell` 方法修改之前

```
def set_cell(self, row, column, shape_label):
    '''把第row行column列的格子打上方块记号（如S, L...），因为该格子被此方块占据
```

```
    。'''
3.      self.area[row][column] = shape_label
```

代码10 GameWall 类的 `set_cell` 方法修改之后

GameWall 类的 `add_to_call` 方法调用 `set_cell` 方法的地方需要作出相应的修改，如代码11和代码12所示。

```
1.  self.set_cell((piece.x + c, piece.y + r), piece.shape)
```

代码11 GameWall 类的 `add_to_call` 方法内修改之前

```
1.  self.set_cell(piece.y + r, piece.x + c, piece.shape)
```

代码12 GameWall 类的 `add_to_call` 方法内修改之后

解决方块重叠问题

解决这一问题的思路是，方块下落时，下方碰到墙体，就视为触底。也就是说，我们应该把触底概念加以扩展。以前，这一概念是指碰到游戏区域底部边界线。现在，这一概念是指碰到游戏区域底部边界线或墙体。

因此，修改的地方是在 Piece 类的 `can_move_down()` 方法内。前面的步骤中，当检测到方块到达底部，该方法就返回 False。现在，我们要加一个条件，变成“当检测到方块到达底部或者碰到墙体”，`can_move_down()` 方法返回 False。

新问题来了——我们怎么知道往下移动会/不会碰到墙体。方块对象本身是没有足够的信息以得出会还是不会这一结论的。

我们的做法是：

1. 在方块对象内安排一个属性，引用墙体对象。在方块对象的构造方法内传入墙体对象。
2. 墙体对象拥有告知第 r 行第 c 列单元格有没有砖块的方法 `is_wall()`。有砖块，该方法返回 True，否则返回 False。
3. 在 Piece 类的 `can_move_down()` 方法内，调用墙体对象的 `is_wall()` 方法。

方块对象引用墙体对象

定义方块对象的是 Piece 类。Piece 类的构造函数内处作出两处修改，即下面代码中第二行和最后一行。下面的代码是修改完成后的。你可以跟以前的版本进行比较。

```
class Piece():
    def __init__(self, shape, screen, gamewall):
        self.x = 4
        self.y = 0
        self.shape = shape
        self.turn_times = 0    #翻转了几次，决定显示的模样
        self.screen = screen
        self.is_on_bottom = False     #到达底部了吗？
        self.game_wall = gamewall
```

相应地，main.py 文件内创建方块对象的两个地方要作出修改，即下面代码中第一行、第二行和最后一行。生成 `game_wall` 对象的语句要在生成方块对象（赋值给 piece 变量）的语句之前。

```

 ……    #未改动
game_wall = GameWall(screen)   #此处作了修改
piece = Piece(random.choice(PIECE_TYPES), screen, game_wall) # game_wal
l 作了修改
#游戏主循环
while True:
    #方块触底的话
    if piece.is_on_bottom:
        game_wall.add_to_wall(piece)
        piece = Piece(random.choice(PIECE_TYPES), screen, game_wall) #g
ame_wall 作了修改
    ……   #未改动
```

GameWall 类的 `is_wall` 方法

GameWall 类中增设 `is_wall()` 方法，作用是判别单元格内是否有砖块。只要单元格对应的墙体矩阵元素的值不是 `-`（即常量 `WALL_BLANK_LABEL`），就表明该单元格内有砖块。方法定义如下。

```
def is_wall(self, row, column):
```

```
        return self.area[row][column] != WALL_BLANK_LABEL
```

调整 `can_move_down` 方法

Piece 类的 `can_move_down` 方法用于检测方块能否下落。它的修改如下面代码中所示。这一部分实现了碰到墙体即视为触底，将不会继续下落。

```

def can_move_down(self):
    shape_mtx = PIECES[self.shape][self.turn_times]  #姿态矩阵
    # print(shape_mtx)
    for r in range(len(shape_mtx)):
        for c in range(len(shape_mtx[0])):
            if shape_mtx[r][c] == 'O':
                if self.y + r >= LINE_NUM - 1 or self.game_wall.is_wall(self.y + r + 1, self.x + c): # 此行作了修改
                    return False
    return True
```

对修改处的代码说明如下：

1. `self.y + r` 是组成当前方块的小块所在的单元格的行号。`self.y + r + 1` 就是下方单元格的行号。这里，单元格是指 20X10 网格中的一格。`self.x + c` 是小块所在的单元格的列号。
2. 如果 `is_wall(self.y + r + 1, self.x + c)` 返回 True，意味着方块正下方是墙体，不能再向下移动。
3. 强调一点，`self.x + c` 不能写作 c。这里的 c，只是小块在姿态矩阵中的列号。而 `self.x + c` 是在 20X10 网格内的列号。

小结

到此，本文解决了方块重叠问题。一开始，我们重构了代码，目的是用统一的格式来标识单元格。在软件开发过程中，重构代码是常规操作。你有必要养成经常重构代码的习惯。

解决方块重叠问题的思路是检测方块有没有碰到底部边界或墙体。为检测有没有碰到墙体，我们在方块类 Piece 内定义 wall 属性来引用墙体对象，并调用墙体对象的 `is_wall` 方法来检测某个单元格是否是属于墙体的一部分。

完整的代码可从以下链接下载：

- Github

下一篇我们将实现方块自动下落的功能，比如每过1秒下落一行。