深 圳 大 学 实 验 报 告

课程名称：计算机图形学

实验项目名称：期中大作业 俄罗斯方块

学院：计算机与软件学院

专业：计算机科学与技术

指导教师：周虹

报告人：吴嘉楷 学号：2022150168 班级：国际班

实验时间：2024 年 09 月 23 日 -- 2024 年 10 月 21 日

实验报告提交时间：2024 年 10 月 18 日

教务部制

实验目的与要求：

1. 强化 OpenGL 的基本绘制方法、键盘等交互事件的响应逻辑，实现更加复杂的绘制操作，完成一个简化版俄罗斯方块游戏。
2. 方块/棋盘格的渲染和方块向下移动。创建 OpenGL 绘制窗口，然后绘制网格线来完成对棋盘格的渲染。随机选择方块并赋上颜色，从窗口最上方中间开始往下自动移动，每次移动一个格子。初始的方块类型和方向也必须随机选择，另外可以通过键盘控制方块向下移动的速度，在方块移动到窗口底部的时候，新的方块出现并重复上述移动过程。
3. 方块叠加。不断下落的方块需要能够相互叠加在一起，即不同的方块之间不能相互碰撞和叠加。另外，所有方块移动不能超出窗口的边界。
4. 键盘控制方块的移动。通过方向键（上/下/左/右）来控制方块的移动。按“上”键使方块以旋转中心顺（逆）时针旋转，每次旋转 90°，按“左”和“右”键分别将方块向左/右方向移动一格，按“下”键加速方块移动。
5. 游戏控制。当游戏窗口中的任意一行被方块占满，该行即被消除，所有上面的方块向下移动一格子。当整个窗口被占满而不能再出现新的方块时，游戏结束。通过按下“q”键结束游戏，和按下“r”键重新开始游戏。
6. 其他扩展。在以上基本内容的基础上，可以增加更多丰富游戏性的功能，如通过空格键使方块快速下落等。

实验过程及内容：

1. 绘制‘J’、‘Z’等形状的方块

```
glm::vec2 allRotationsLshape[7][4][4] = {
    {    // O 型
        {glm::vec2(0, 0), glm::vec2(-1, 0), glm::vec2(0, -1), glm::vec2(-1, -1)},
        {glm::vec2(0, 0), glm::vec2(-1, 0), glm::vec2(0, -1), glm::vec2(-1, -1)},
        {glm::vec2(0, 0), glm::vec2(-1, 0), glm::vec2(0, -1), glm::vec2(-1, -1)},
        {glm::vec2(0, 0), glm::vec2(-1, 0), glm::vec2(0, -1), glm::vec2(-1, -1)}
    },
    {    // I 型
        {glm::vec2(-2, 0), glm::vec2(-1, 0), glm::vec2(0, 0), glm::vec2(1, 0)},
        {glm::vec2(0, 1), glm::vec2(0, 0), glm::vec2(0, -1), glm::vec2(0, -2)},
        {glm::vec2(-2, 0), glm::vec2(-1, 0), glm::vec2(1, 0), glm::vec2(0, 0)},
        {glm::vec2(0, 1), glm::vec2(0, 0), glm::vec2(0, -1), glm::vec2(0, -2)}
    },
    {    // S 型
        {glm::vec2(0, 0), glm::vec2(0, -1), glm::vec2(-1, -1), glm::vec2(1, 0)},
        {glm::vec2(0, 1), glm::vec2(0, 0), glm::vec2(1, 0), glm::vec2(1, -1)},
        {glm::vec2(0, 0), glm::vec2(0, -1), glm::vec2(-1, -1), glm::vec2(1, 0)},
        {glm::vec2(0, 1), glm::vec2(0, 0), glm::vec2(1, 0), glm::vec2(1, -1)}
    },
    {    //   Z 型
        {glm::vec2(-1, 0), glm::vec2(0, 0), glm::vec2(0, -1), glm::vec2(1, -1)},
        {glm::vec2(0, 0), glm::vec2(0, -1), glm::vec2(1, 0), glm::vec2(1, 1)},
```

```
            {glm::vec2(-1, 0), glm::vec2(0, 0), glm::vec2(0, -1), glm::vec2(1, -1)},
            {glm::vec2(0, 0), glm::vec2(0, -1), glm::vec2(1, 0), glm::vec2(1, 1)}
        },
        {    // L 型
            {glm::vec2(0, 0), glm::vec2(-1, 0), glm::vec2(1, 0), glm::vec2(-1, -1)},
            {glm::vec2(0, 1), glm::vec2(0, 0), glm::vec2(0, -1), glm::vec2(1, -1)},
            {glm::vec2(1, 1), glm::vec2(-1, 0), glm::vec2(0, 0), glm::vec2(1, 0)},
            {glm::vec2(-1, 1), glm::vec2(0, 1), glm::vec2(0, 0), glm::vec2(0, -1)}
        },
        {    // J 型
            {glm::vec2(-1, 0), glm::vec2(0, 0), glm::vec2(1, 0), glm::vec2(1, -1)},
            {glm::vec2(0, 1), glm::vec2(0, 0), glm::vec2(0, -1), glm::vec2(1, 1)},
            {glm::vec2(-1, 0), glm::vec2(0, 0), glm::vec2(1, 0), glm::vec2(-1, 1)},
            {glm::vec2(-1, -1), glm::vec2(0, -1), glm::vec2(0, 0), glm::vec2(0, 1)}
        },
        {    // T 型
            {glm::vec2(-1, 0), glm::vec2(0, 0), glm::vec2(1, 0), glm::vec2(0, -1)},
            {glm::vec2(0, -1), glm::vec2(0, 0), glm::vec2(0, 1), glm::vec2(1, 0)},
            {glm::vec2(-1, 0), glm::vec2(0, 0), glm::vec2(1, 0), glm::vec2(0, 1)},
            {glm::vec2(-1, 0), glm::vec2(0, -1), glm::vec2(0, 0), glm::vec2(0, 1)}
        },
};
```

代码说明：

这段代码定义了一个三维数组 allRotationsLshape，用于存储七种不同方块的旋转坐标。每种方块有四个旋转方向，每个方向由四个格子的坐标表示。数组的结构是 allRotationsLshape[7][4][4]：

（1）第一个维度 [7] 代表七种方块类型，包括 O 型、I 型、S 型、Z 型、L 型、J 型和 T 型。

（2）第二个维度 [4] 表示四个旋转角度（0°，90°，180°，270°）。

（3）第三个维度 [4] 表示每个方块的四个格子在二维平面上的位置，用 glm::vec2(x, y) 存储。

添加变量记录方块形状。

```
30    int rotation = 0;          // 控制当前窗口中的方块旋转
31    int shapeLike = 0          // 控制当前窗口中的方块形状
32    glm::vec2 tile[4];            // 表示当前窗口中的方块
```

图 1 添加 shapeLike 变量记录方块形状

通过改变 shapeLike 变量的值，我们可以很方便地控制生成的方块的形状。这样一来，我们只需要随机 shapeLike 的取值，即可随机生成不同类型的方块。

3. 随机生成方块，同时更新 tile 数组

通过 rand() % 7 随机生成一个介于 0 到 6 的整数，并将其赋值给变量 shapeLike，用于随机选择一种方块形状（0 表示 O 型，1 表示 I 型，以此类推）。这

样，程序每次生成一个新的方块时，都会选择不同的方块类型。

然后使用了一个循环，将 allRotationsLshape 数组中 shapeLike 对应的方块类型的初始旋转方向（[0] 表示默认的 0° 旋转方向）中的四个格子坐标依次赋值给 tile 数组，以便初始化新生成的方块在棋盘上的初始位置和形状。

```
void newtile()
{
    // 将新方块放于棋盘格的最上行中间位置并设置默认的旋转方向
    tilepos = glm::vec2(5 , 19);
    rotation = 0;
    // 随机一个方块形状
    shapeLike = rand() % 7;
    // 更新当前方块的位置
    for (int i = 0; i < 4; i++)
    {
        tile[i] = allRotationsLshape[shapeLike][0][i];
    }
}
```

图 2 修改 newtile 函数

4. 修正 rotate 函数

由于我们将 allRotationsLshape 数组从二维提升到了三维，因此涉及到 allRotationsLshape 数组的使用之处都需要进行修改。由下图可见，shapeLike 变量可以控制某一个单一形状方块的旋转，而不改变方块的类型或形状。

```
void rotate()
{
    // 计算得到下一个旋转方向
    int nextrotation = (rotation + 1) % 4;

    // 检查当前旋转之后的位置的有效性
    if (checkvalid((allRotationsLshape[shapeLike][nextrotation][0]) + tilepos)
        && checkvalid((allRotationsLshape[shapeLike][nextrotation][1]) + tilepos)
        && checkvalid((allRotationsLshape[shapeLike][nextrotation][2]) + tilepos)
        && checkvalid((allRotationsLshape[shapeLike][nextrotation][3]) + tilepos))
    {
        // 更新旋转，将当前方块设置为旋转之后的方块
        rotation = nextrotation;
        for (int i = 0; i < 4; i++)
            tile[i] = allRotationsLshape[shapeLike][rotation][i];

        updatetile();
    }
}
```

图 3 修改 rotate 函数

5. 添加颜色的种类并将名字存储到数组中

```
// 绘制窗口的颜色变量
glm::vec4 orange = glm::vec4(1.0, 0.5, 0.0, 1.0);
glm::vec4 white  = glm::vec4(1.0, 1.0, 1.0, 1.0);
glm::vec4 black  = glm::vec4(0.0, 0.0, 0.0, 1.0);
glm::vec4 red = glm::vec4(1.0, 0.0, 0.0, 1.0);
glm::vec4 green = glm::vec4(0.0, 1.0, 0.0, 1.0);
glm::vec4 blue = glm::vec4(0.0, 0.0, 1.0, 1.0);
glm::vec4 yellow = glm::vec4(1.0, 1.0, 0.0, 1.0);
glm::vec4 purple = glm::vec4(0.5, 0.0, 0.5, 1.0);
glm::vec4 cyan = glm::vec4(0.0, 1.0, 1.0, 1.0);
glm::vec4 colors[] = { orange, red, green, blue, yellow, purple, cyan };
```

图 4 添加颜色变量

6. 为每一个类型的方块都赋上不同的颜色

（1）在 newtile 函数中修改 newcolours 数组的赋值：

```
// 给新方块赋上颜色
glm::vec4 newcolours[24];
for (int i = 0; i < 24; i++)
    newcolours[i] = colors[shapeLike];
```

图 5 根据方块形状赋上不同的颜色

newtile 函数说明：

函数 newtile() 负责生成新的方块并初始化其相关属性。

首先，将方块置于棋盘顶部中间位置 (5, 19)，并将旋转方向设为 0，即默认方向。然后，使用 shapeLike = rand() % 7; 随机生成一个 0 到 6 的整数，来选择不同的方块类型。

接下来，通过循环将 allRotationsLshape 中 shapeLike 所对应的方块形状在初始方向下的四个格子坐标赋值给 tile 数组，从而设置当前方块的初始位置。

之后调用 updatetile() 函数更新方块状态。为了赋予方块颜色，定义一个 newcolours 数组，并用 colors[shapeLike] 为每个元素赋值，对应该形状的方块的颜色。

随后，绑定颜色缓冲区 vbo[5] 并用 glBufferSubData 更新 newcolours 数据，确保方块渲染时颜色正确。

最后，解绑 VAO (`glBindVertexArray(0)`) 以恢复渲染状态。

（2）修改 settile 函数，使用不同的颜色渲染方格：

```
void settile()
{
    // 每个格子
    for (int i = 0; i < 4; i++)
    {
        // 获取格子在棋盘格上的坐标
        int x = (tile[i] + tilepos).x;
        int y = (tile[i] + tilepos).y;
        // 将格子对应在棋盘格上的位置设置为填充
        board[x][y] = true;
        // 并将相应位置的颜色修改
        changecellcolour(glm::vec2(x, y), colors[shapeLike]);
    }
}
```

图 6 修改 settile 函数

settile 函数说明：

该函数 settile() 用于将当前方块的格子位置记录到棋盘上，并在这些位置上显示对应的颜色。首先，通过循环遍历方块的每个格子，计算格子在棋盘上的实际位置 (x, y)，其中 tile[i] + tilepos 表示格子相对于棋盘的坐标。接着，将棋盘上该位置的 board[x][y] 设置为 true，表示该位置已被填充。然后，通过调用 changecellcolour(glm::vec2(x, y), colors[shapeLike]) 设置填充位置的颜色，使棋盘显示该方块的颜色。

7. 实现方块的自动向下移动

```
init();
int last = clock();
while (!glfwWindowShouldClose(window))
{
    display();
    glfwSwapBuffers(window);
    glfwPollEvents();
    //使得每隔一秒方块下落一格
    int now = clock();      //记录当前时间
    if (now - last >= 1000)     //相差1s
    {
        if (!movetile(glm::vec2(0, -1)))    //向下移动，若下移失败则执行下面部分
        {
            settile();          //放置在底部
            newtile();          //新建方块
        }
        last = now;             //更新当前时间
    }
}
glfwTerminate();
```

图 7 修改 main 函数的循环渲染逻辑

该代码段实现了一个简单的游戏主循环，在其中控制方块每隔一秒下落一格。首先，int last = clock(); 记录初始时间，然后进入 while 循环，该循环会持续运行，直到窗口关闭 (!glfwWindowShouldClose(window))。

在循环中，首先调用 display() 进行绘制，并通过 glfwSwapBuffers(window) 刷新窗口显示，通过 glfwPollEvents() 处理用户输入事件。接下来，用 int now = clock(); 获取当前时间，与之前的 last 比较，如果时间间隔达到 1 秒 (now - last >= 1000)，则尝试将方块向下移动一格，调用 movetile(glm::vec2(0, -1))。

如果向下移动失败，说明方块已到达底部或被阻挡，此时调用 settile() 将方块固定在当前位置，并通过 newtile() 生成一个新的方块。最后更新 last = now，将当前时间作为下次判断的起点。

8. 完善 restart 函数，使得按 r 键可以重新开始游戏

函数源码：

```
void restart()
{
    system("cls");          //清除终端内容
    cout << "游戏重启的号角已吹响！\n 再次踏上征程，闪耀你的辉煌时刻！！！" << endl << endl;
    init();
}
```

函数说明：

该 restart() 函数用于重启游戏。首先，通过 system("cls"); 清除终端内容，以提供一个干净的输出界面。然后，输出一条激励性消息 "游戏重启的号角已吹响！\n 再次踏上征程，闪耀你的辉煌时刻！！！"，提示玩家游戏已重新开始。最后调用 init() 函数，重新初始化游戏状态或变量，准备进入新一轮游戏。

9. 添加方块之间、方块与边界之间的碰撞检测

```
bool checkvalid(glm::vec2 cellpos)
{
    if((cellpos.x >=0) && (cellpos.x < board_width) && (cellpos.y >= 0) && (cellpos.y < board_height)
        && board[(int)cellpos.x][(int)cellpos.y] == false) // 判断是否在边界范围内并且没有被填充
        return true;
    else
        return false;
}
```

图 8 修改 checkvalid 函数

函数说明：

checkvalid() 函数用于检查指定位置 cellpos 是否有效，即是否在棋盘边界范围内且未被填充。具体而言：

条件 (cellpos.x >= 0) && (cellpos.x < board_width) 确保 cellpos.x 在棋盘宽度范围内。

条件 (cellpos.y >= 0) && (cellpos.y < board_height) 确保 cellpos.y 在棋盘高度范围内。

??. 条件 board[(int)cellpos.x][(int)cellpos.y] == false 检查该位置是否为空（未被填充）。若上述条件均满足，则返回 true 表示该位置有效；否则返回 false。

创建一个数组记录每个格子的颜色。（方便每个方格颜色的整体下移）

```
// 记录每个格子的颜色
glm::vec4 tile_colours[board_width][board_height];
```

图 9 创建记录每个方格颜色的数组

```
void changecellcolour(glm::vec2 pos, glm::vec4 colour)
{
    // 每个格子是个正方形，包含两个三角形，总共6个定点，并在特定的位置赋上适当的颜色
    for (int i = 0; i < 6; i++)
        board_colours[(int)( 6 * ( board_width*pos.y + pos.x) + i)] = colour;

    glm::vec4 newcolours[6] = {colour, colour, colour, colour, colour, colour};

    glBindBuffer(GL_ARRAY_BUFFER, vbo[3]);

    // 计算偏移量，在适当的位置赋上颜色
    int offset = 6 * sizeof(glm::vec4) * (int)( board_width * pos.y + pos.x);
    glBufferSubData(GL_ARRAY_BUFFER, offset, sizeof(newcolours), newcolours);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    // 记录每个格子的颜色
    tile_colours[(int)pos.x][(int)pos.y] = colour;
}
```

图 10 为数组元素赋值（记录颜色）

```
// 将棋盘格所有位置的填充与否都设置为false（没有被填充）
for (int i = 0; i < board_width; i++)
    for (int j = 0; j < board_height; j++) {
        board[i][j] = false;
        tile_colours[i][j] = black; // 方格颜色初始化为黑色
    }
```

图 11 初始化颜色记录数组

代码说明：

此步骤创建了一个名为 tile_colours 的二维数组，用于记录每一个方格的颜色。数组元素初始化为黑色，并且当触发 changecellcolour 函数时同时进行对 tile_colours 数组元素的修改，从而实时记录每一个方格的颜色，便于消除棋盘某一行后方块的整体下移。

11. 检测棋盘是否存在某一行充满的情况

```
void settile()
{
    // 每个格子
    for (int i = 0; i < 4; i++)
    {
        // 获取格子在棋盘格上的坐标
        int x = (tile[i] + tilepos).x;
        int y = (tile[i] + tilepos).y;
        // 将格子对应在棋盘格上的位置设置为填充
        board[x][y] = true;
        // 并将相应位置的颜色修改
        changecellcolour(glm::vec2(x, y), colors[shapeLike]);
    }
    // 检查棋盘格在当前方块所在的行有没有被填充满
    for (int i = 0; i < 4; i++)
    {
        // 获取格子在棋盘格上的纵坐标
        int y = (tile[i] + tilepos).y;
        checkfullrow(y); // 检查第y行是否被填充满
    }
}
```

图 12 修改 settile 函数检测棋盘情况

函数说明：

该函数的主要功能是将当前方块的位置更新到棋盘，并检查当前方块所在的行是否被填满。

首先，前半部分通过循环将方块的每个格子的坐标计算出来。具体步骤如下：

（1）循环遍历当前方块的四个格子。

（2）计算出格子在棋盘中的坐标 (x, y)，即通过 tile[i] + tilepos 获取相对位置，并提取 x 和 y 值。

（3）将棋盘上的对应位置 board[x][y] 标记为填充，即设置为 true。

（4）调用 changecellcolour(glm::vec2(x, y), colors[shapeLike])，为该位置设置对应方块的颜色。

在处理完当前方块的所有格子之后，第二部分则检查当前方块所在的行是否被填满：
（1）再次循环遍历方块的四个格子。
（2）计算出每个格子所在的行 y。
（3）调用 checkfullrow(y)，检查当前行 y 是否已被填满。

如果某一行被填满，则消除该行并更新游戏状态。

12. 完善 checkfullrow 函数，使棋盘格中每一行填充满之后自动消除
函数源码：

```
void checkfullrow(int row)
{
    // 如果没有填充满 row 行，直接返回
    for (int i = 0; i < board_width; i++)
        if (!board[i][row])
            return;
    // 否则，将 row 行上面的所有行都下移一行
    for (int j = row; j < board_height; j++) // 从被消除的行开始，向上判断
        for (int i = 0; i < board_width; i++)
        {
            // 若上面一行有格子，将其颜色赋给当前行
            if (j < board_height - 1)
            {
                board[i][j] = board[i][j + 1];
                changecellcolour(glm::vec2(i, j), tile_colours[i][j + 1]);
            }
            else
            {
                board[i][j] = false;
                changecellcolour(glm::vec2(i, j), black);
            }
        }
}
```

函数说明：

checkfullrow(int row) 函数的作用是检查指定的行是否已经被填满，如果填满了，就将该行删除，并将其上方的所有行下移一行。

首先，函数通过一个循环检查 row 行的每个格子，判断是否都被填充。如果发现任何一个格子未被填充（board[i][row] == false），就直接返回，不执行任何进一步的操作。如果所有格子都被填充，则认为这一行满了，接着进入下一个步骤。

接下来，函数会逐行将 row 行及其上方的所有行向下移动一行。通过一个嵌套的循环，外层循环从 row 行开始，逐行处理上面的每一行，直到顶部。内层循环遍历当前行的每个格子，将上方一行的填充状态和颜色复制到当前行。也就是说，每一行都会将它上面一行的格子状态赋值给自己，实现行的下移。

最后，当到达最顶部的行时，因为上面没有更多的行可以复制，所以这行需要被清空。顶部的格子会被标记为未填充，并将颜色设置为黑色，表示该行为空。

整个过程确保在某一行填满后，该行会被删除，上方的行依次下移，从而模拟游戏中方块消除的效果。

13. 修改运行窗口的标题和尺寸

```
// 创建窗口。
GLFWwindow* window = glfwCreateWindow(800, 1440, "2022150168-wjk-MidProj", NULL, NULL);
```

图 13 修改窗口配置

注意：如果使用中文需要设置字符集为“GBK”格式的，否则会由于编解码字符集不统一而产生乱码的现象。

14. 查看运行效果

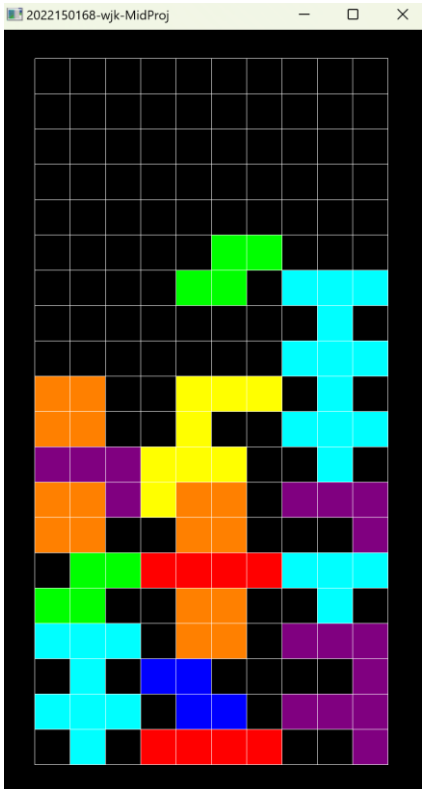

图 14 部分运行结果

15. 拓展功能一：通过空格键使方块快速下落

在 key_callback 函数中添加对空格键的响应逻辑：

```
case GLFW_KEY_SPACE: // 空格键，快速下降到底部
    if (action == GLFW_PRESS) {
        while (movetile(glm::vec2(0, -1)));
        settile();
        newtile();
    }
    break;
```

图 15 按空格键快速下落

代码说明：

这段代码的作用是在检测到玩家按下空格键时，使当前方块快速下降到底部，并生成一个新方块。

首先，当空格键被按下时，if (action == GLFW_PRESS) 这一条件判断会成立，进入代码执行部分。紧接着，while (movetile(glm::vec2(0, -1))); 这一行会执行一个循环，使方

块不断向下移动，直到不能再继续下移为止。movetile(glm::vec2(0, -1)) 的作用是将方块向下移动一个单位（在 Y 轴上减 1），如果移动成功则返回 true，如果遇到障碍或到底部而不能继续移动，则返回 false，使得循环终止。

在方块无法再向下移动时，代码调用 settile(); 将当前方块固定到棋盘上。接着，newtile() 生成一个新的方块，以继续游戏的流程。

16. 扩展功能二：添加欢迎语、按键功能说明

函数源码：

```
void welcome() {
    cout << "欢迎来到俄罗斯方块，祝您玩的愉快！" << endl << endl;
    cout << "游戏玩法：" << endl;
    cout << "\t 通过调整下落的随机方块，使其填充满一行后消除获得分数。" << endl;
    cout << "\t 分数越高，下落速度越快哦！" << endl << endl;
    cout << "按键说明：" << endl;
    cout << "\t↑键：旋转方块" << endl;
    cout << "\t←或→键：移动方块" << endl;
    cout << "\t↓键：加速方块下落" << endl;
    cout << "\t 空格键：下落方块至底部" << endl;
    cout << "\tq 键或 Esc 键：退出游戏" << endl;
    cout << "\tr 键：重新开始游戏" << endl << endl;
}
```

17. 扩展功能三：添加计分功能

```
36    int ysize = 720;
37    int score = 0;          // 游戏得分
38
```

图 16 添加得分变量

```
// 欢迎界面
void welcome() {
    cout << "欢迎来到俄罗斯方块，祝您玩的愉快！" << endl << endl;
    cout << "游戏玩法：" << endl;
    cout << "\t通过调整下落的随机方块，使其填充满一行后消除获得分数。" << endl;
    cout << "\t分数越高，下落速度越快哦！" << endl << endl;
    cout << "按键说明：" << endl;
    cout << "\t↑键：旋转方块" << endl;
    cout << "\t←或→键：移动方块" << endl;
    cout << "\t↓键：加速方块下落" << endl;
    cout << "\t空格键：下落方块至底部" << endl;
    cout << "\tq键或Esc键：退出游戏" << endl;
    cout << "\tr键：重新开始游戏" << endl << endl;

    cout << "当前游戏得分：" << score << endl;
}
```

图 17 添加得分提示

```
void checkfullrow(int row)
{
    // 如果没有填充满row行，直接返回
    for (int i = 0; i < board_width; i++)
        if (!board[i][row])
            return;
    // 否则，将row行上面的所有行都下移一行，并进行加分
    // 消除成功，加分
    score += 10;
    system("cls"); // 清空终端内容
    welcome();

    // 将row行上面的所有行都下移一行
```

图 18 消除一行时加 10 分并打印提示语

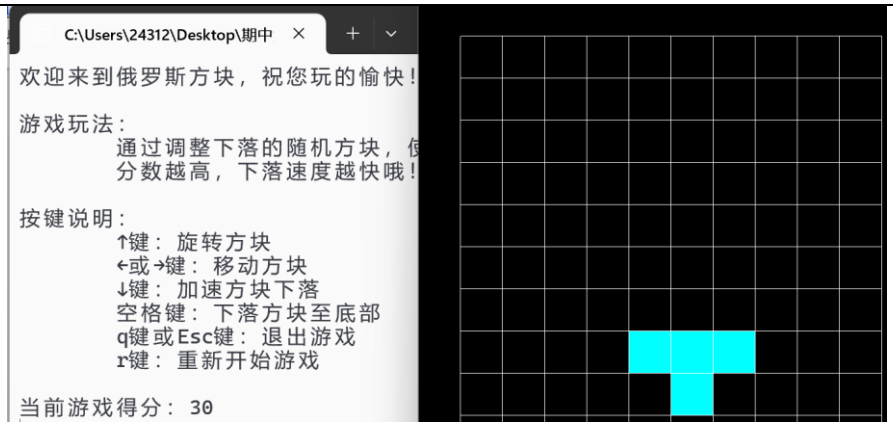

图 19 消除 3 行后的控制台输出

18. 扩展功能四：记录历史最高分，以提高用户粘性

```
int score = 0;          // 游戏得分
int maxScore = 0;       // 最高分
// 单个网格大小
```

图 20 定义记录最高分的变量

```
if (!myGameOver) {
    cout << "游戏结束！" << endl;
    cout << "您的得分为：" << score << endl;
    if (score > maxScore)
    {
        maxScore = score;
        cout << "恭喜您创造了新的最高分！" << endl;
    }
    cout << "最高分为：" << maxScore << endl;
    cout << "按r键重新开始游戏" << endl;
}
```

图 21 游戏结束时更新最高分

19. 扩展功能五：补充游戏结束判断条件，不能生成方块则游戏结束

```
bool myGameOver = false;
void newTile()
{
    // 将新方块放于棋盘格的最上行中间位置并设置默认的旋转方向
    tilepos = glm::vec2(5 , 19);
    rotation = 0;
    // 随机一个方块形状
    // 判断是否有足够的空间来放置新的方块
    for (int i = 0; i < 4; i++)
    {
        if (board[(int)allRotationsLshape[shapeLike][0][i].x + 5][(int)allRotationsLshape[shapeLike][
        {
            if (!myGameOver) {
                cout << "游戏结束！" << endl;
                cout << "您的得分为：" << score << endl;
                cout << "最高分为：" << maxScore << endl;
                if (score > maxScore)
                {
                    maxScore = score;
                    cout << "恭喜您创造了新的最高分！" << endl;
                }
                cout << "按r键重新开始游戏" << endl;
            }
            myGameOver = true;
            return;
        }
    }
    // 更新当前方块的位置
```

图 22 新增游戏结束判断逻辑

20. 扩展功能六：添加梯度难度设计，消除方块后下落速度提高。
初始时，每隔一秒下落一格：

```
int maxScore = 0;          // 最高分
int moveTime = 1000;       // 方块下落时间间隔
```

图 23 定义下落时间间隔的变量

在 init 函数中进行初始化：

```
// 游戏和OpenGL初始化
void init()
{
    // 分数初始化为0
    score = 0;

    myGameOver = false;
    moveTime = 1000; // 初始化方块下落时间间隔

    // 输出提示语
    welcome();
```

图 24 初始化下落时间间隔

在消除成功时，加快方块下落速度：

```
// 检查棋盘格在row行有没有被填充满
void checkfullrow(int row)
{
    // 如果没有填充满row行，直接返回
    for (int i = 0; i < board_width; i++)
        if (!board[i][row])
            return;
    // 否则，将row行上面的所有行都下移一行，并进行加
    // 消除成功，加分
    score += 10;
    system("cls"); // 清空终端内容
    welcome();
    // 加快方块下落速度
    moveTime -= 100;
    cout << "注意：方块下落速度加快了！" << endl;
```

图 25 缩短方块下落时间间隔

实验结论：

通过本次实验，我加深了对计算机图形学的理解，特别是 OpenGL 的使用和实现复杂绘制任务的能力。在实现俄罗斯方块的过程中，我掌握了基本图形的渲染方法，包括棋盘网格、方块的绘制、方块的自动下落和相互叠加。同时，键盘事件的处理使我体会到如何通过输入设备控制图形对象，进一步强化了交互式图形编程的能力。通过分解任务，我有效实现了碰撞检测、方块消除以及得分机制等功能，展现了较为完整的游戏逻辑。实验中还探索了额外的功能扩展，如通过空格键实现方块的快速下落、动态速度调整等。这不仅巩固了我对 OpenGL 的掌握，也锻炼了如何将复杂系统分解为多个模块并独立调试和优化的能力。

本次实验让我深刻体会到图形学中矩阵变换和碰撞检测的关键作用，尤其是如何在二维平面上通过坐标变换实现方块的旋转和移动。同时，GLFW 与 OpenGL 的结合使用为我提供了一个高度可扩展的交互平台，让我能实现更加复杂的交互功能。在调试过程中，我也感受到优化代码结构的重要性，尤其是在多个模块间传递数据时，保证代码的简洁性与可读性显得尤为重要。实验中遇到的一些困难，比如方块旋转导致的碰撞检测

问题，促使我深入理解了矩阵变换的原理，并找到了合理的解决方案。

总的来说，这次实验不但提升了我在 OpenGL 编程方面的技能，也让我对图形学的实际应用有了更加深刻的认识。

实验过程中遇到的一些问题：

1. 修改运行窗口的标题时，使用中文产生了乱码现象
解决方法：添加 `#pragma execution_character_set("utf-8")` ，从而告知编译器源代码文件所使用的字符集为 UTF-8。这样可以确保源代码中的字符串文字被正确解析为 UTF-8 编码。在编译程序时，编译器将使用 UTF-8 字符集来解释这些字符串，以便在程序运行时以 UTF-8 编码的形式进行处理。

2. 在补充游戏结束判断条件时，将 gameover 变量设置为 true 后，程序无法继续执行
原因：在 key_callback 函数中有一段原有逻辑，它只会在 gameover = false 的时候触发键盘控制逻辑，因此不能修改 gameover 的值，否则将无法使用 r 键重新开始游戏。
解决方法：新增一个 myGameOver 变量，从而判断游戏是否结束并输出提示语。

指导教师批阅意见：

成绩评定：

指导教师签字：

年　　月　　日

备注：

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。