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：林宪亮 学号：2022150130 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 17 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. 方块/棋盘格的渲染和方块向下移动

1.1 绘制其它形状的方块

```
glm::vec2 allRotationsLshape[7][4][4] = {
    {    // O
        {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
        {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)},
        {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
        {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
        {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
        {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
            {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
            {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)}},
};
```

代码解释：

这段 C++ 代码是我定义的一个三维数组 allRotationsLshape，用来存储七种不同形状的所有旋转状态，就像俄罗斯方块里的那些形状。数组的结构如下：

数组维度：

glm::vec2 allRotationsLshape[7][4][4]：7 是我用来表示七种不同的形状，比如 O 型、I 型、S 型等。4 表示每种形状有四种旋转状态，分别是 0°、90°、180° 和 270°。

每个旋转状态下，我用四个 glm::vec2 存储形状中每个小方块的相对坐标。

每种形状的表示：

我为每种形状定义了四个旋转状态，每种状态下，每个形状由四个方块组成。这些方块的坐标是相对于形状的中心点来定义的。

1.2 随机选择方块以及颜色

```
tilepos = glm::vec2(5, 19);
rotation = 0;
srand(time(0));
shapeLike = rand() % 7;
```

我将新方块的位置初始化为 (5, 19)，即棋盘的最上行中间位置，用 glm::vec2 来表示坐标。这里 (5, 19) 是棋盘的相对坐标，其中 5 代表水平位置，19 代表垂直位置。初始的旋转方向 rotation 被设置为 0，表示没有旋转。

```
glm::vec4 newcolours[24];
for (int i = 0; i < 24; i++)
    newcolours[i] = colors[shapeLike];
```

新方块会根据其形状被赋予相应的颜色。我先创建一个 glm::vec4 newcolours[24] 数组来存储方块的颜色，并将颜色赋给 newcolours[i]，颜色信息来自 colors[shapeLike]。

1.3 方块自动下移

```
//使得每隔一秒方块下落一格
int now = clock();            //记录时间
if (now - flag >= 1000)            //相差 1s
{
    if (!movetile(glm::vec2(0, lowSpeed)))      //向下移动
    {
        settile();              //放置底部
        newtile();              //新建方块
    }
        flag = now;                     //更新时间
```

这段代码实现了使得每隔一秒钟，当前方块自动下落一格的逻辑。以下是具体解释：

• 记录当前时间：

`int now = clock();`：我使用 `clock()` 函数来获取当前的系统时间。`clock()` 返回的时间单位是以时钟周期计的，通常是毫秒。

• 检测是否经过了 1 秒：

`if (now - flag >= 1000)`：这里 `flag` 是上一次记录的时间，条件 `now - flag >= 1000` 意味着如果当前时间 `now` 与上次时间 `flag` 之间的差值大于等于 1000 毫秒（即 1 秒），则执行下方代码。换句话说，我在每 1 秒内让方块自动下落一格。

• 尝试向下移动方块：

`if (!movetile(glm::vec2(0, lowSpeed)))`：这里我调用了 `movetile()` 函数，试图让方块向下移动一格（`glm::vec2(0, lowSpeed)`）。`lowSpeed` 代表下落速度（通常是 1，表示移动一格）。

`movetile()` 函数返回一个布尔值。如果移动成功，返回 `true`；如果失败（例如方块到达底部或被其他方块阻挡），返回 `false`。因此 `!movetile()` 表示如果无法下移，则执行下一步。

• 将方块固定在底部并生成新方块：

`settile();`：如果方块无法继续下落，我会调用 `settile()` 函数，将当前方块固定在棋盘的底部或其接触到的地方。

`newtile();`：接着，我调用 `newtile()` 生成一个新的方块，开始下一轮的方块下落。

• 更新当前时间：

`flag = now;`：我更新 `flag` 的值为当前时间 `now`，这样可以确保下一次的 1 秒判断是从现在开始的。

这个逻辑的核心是每隔一秒尝试让方块下落一格，若方块无法下移，则将其固定并生成新的方块。

```
case GLFW_KEY_SPACE: // 空格键，快速下降到底部
    if (action == GLFW_PRESS || action == GLFW_REPEAT) {
        while (movetile(glm::vec2(0, -1))) {
            // 继续下移方块
        }
        // 下移到底部后，设置方块并生成新方块
        settile();
```

```
            newtile();
        }
        break;
}
```

这段代码处理按下 空格键 (GLFW_KEY_SPACE) 时，当前方块会快速下降到底部的功能。以下是具体解释：

• 检测按键：

case GLFW_KEY_SPACE: 这是对空格键的处理。当玩家按下或重复按下空格键时，方块会快速下降到底部。

• 检测按键动作：

if (action == GLFW_PRESS || action == GLFW_REPEAT)：我检查按键的动作是否是按下 (GLFW_PRESS) 或重复按下 (GLFW_REPEAT)。一旦检测到空格键被按下或长按，执行下面的快速下落逻辑。

• 方块快速下降：

while (movetile(glm::vec2(0, -1))) {}：这个 while 循环尝试让方块快速向下移动。我通过调用 movetile() 函数，试图将方块向下移动一格（通过 glm::vec2(0, -1)，表示在 y 轴方向下降一格）。

movetile() 返回 true 表示方块成功下移一格，返回 false 表示方块无法再向下移动（可能到底部或被其他方块阻挡）。因此，当 movetile() 返回 false 时，循环会停止，表示方块已经无法再继续下落。

• 固定方块并生成新方块：

settile();：当方块快速下降到底部后，我调用 settile()，将方块固定在底部或其接触到的其他方块上。

newtile();：接着调用 newtile()，生成一个新的方块，让游戏继续。

2. 方块叠加

2.1 方块间的碰撞检测以及窗口边界检测

```
// 检查在 cellpos 位置的格子是否被填充或者是否在棋盘格的边界范围内
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)//添加方块间的检测判断，判断是否已经有方块放置
            return true;
    else
            return false;
}
```

我定义了一个名为 checkvalid 的函数，它接受一个 glm::vec2 类型的参数 cellpos，表示要检查的格子的位置。

我要检查以下几点：

cellpos.x >= 0：确保格子在棋盘的左边界。

cellpos.x < board_width：确保格子在棋盘的右边界。
cellpos.y >= 0：确保格子在棋盘的下边界。
cellpos.y < board_height：确保格子在棋盘的上边界。
board[(int)cellpos.x][(int)cellpos.y] == false：检查该位置是否没有被其他方块占用。board 是一个布尔数组，表示棋盘的状态，如果该位置为 false，则表示该位置可以被占用。

3. 键盘控制方块的移动

3.1 方块旋转

```
    // 控制方块的移动方向，更改形态
case GLFW_KEY_UP:    // 向上按键旋转方块
    if (action == GLFW_PRESS || action == GLFW_REPEAT)
    {
        rotate();
        break;
    }
    else
    {
        break;
        }
```

当我检测到“向上”键被按下或持续按下时，便调用 rotate() 函数来更新方块的旋转状态，使其在游戏中呈现新的方向。

3.2 方块左右移动

```
case GLFW_KEY_LEFT:  // 向左按键移动方块
    if (action == GLFW_PRESS || action == GLFW_REPEAT) {
        movetile(glm::vec2(-1, 0));
        break;
    }
    else
    {
        break;
    }
case GLFW_KEY_RIGHT: // 向右按键移动方块
    if (action == GLFW_PRESS || action == GLFW_REPEAT) {
        movetile(glm::vec2(1, 0));
        break;
    }
    else
    {
        break;
        }
```

首先，我检查用户是否按下了“向左”键或保持按下状态。如果条件满足，我调用

movetile(glm::vec2(-1, 0)) 来将方块向左移动一格。接下来，类似的逻辑适用于“向右”键：当用户按下或重复按下“向右”键时，我会调用 movetile(glm::vec2(1, 0)) 将方块向右移动。

3.3 方块加速下移

```
case GLFW_KEY_DOWN: // 向下按键移动方块
    if (action == GLFW_PRESS || action == GLFW_REPEAT) {
        if (!movetile(glm::vec2(0, -1)))
        {
            settile();
            newtile();
            break;
        }
        else
        {
            break;
        }
        }
```

当我检测到用户按下或持续按下“向下”键时，首先调用 movetile(glm::vec2(0, -1)) 尝试将方块向下移动一格。如果移动成功，程序继续执行；如果移动失败（也就是方块已经到达底部或被其他方块阻挡），我就调用 settile() 将方块固定在当前的位置，并调用 newtile() 生成一个新的方块。

4. 游戏逻辑

4.1 行填满后消除

```
// 放置当前方块，并且更新棋盘格对应位置顶点的颜色 VBO
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]);
    }

    //对每一个格子所在的行进行检测，看该行是否已满
    for (int i = 0; i < 4; i++)
    {
        // 获取格子在棋盘格上的纵坐标并进行检测
```

```
            int y = (tile[i] + tilepos).y;
            checkfullrow(y);
        }
        }
```

在这段代码中，我实现了 settile() 函数，用于放置当前方块并更新棋盘格中对应位置的顶点颜色。在这个函数中，我首先通过循环遍历当前方块的每个格子，计算它们在棋盘上的坐标。具体来说，我将当前方块的位置 tilepos 与每个格子的相对位置 tile[i] 相加，得到 (x, y) 坐标。接着，我将这些位置在棋盘格上的状态设置为填充，即将 board[x][y] 设为 true。
然后，我调用 changecellcolour(glm::vec2(x, y), colors[shapeLike])，将当前方块颜色应用到棋盘格中对应位置的顶点。这确保了用户可以在视觉上看到方块的填充状态。
最后，我再一次遍历所有格子，检查它们在棋盘格上的纵坐标 y 是否有满行。通过调用 checkfullrow(y)，我能及时检测到并处理已满的行，以便清除并更新游戏状态。

4.2 结束和重启游戏

```
case GLFW_KEY_ESCAPE:
        if (action == GLFW_PRESS) {
            exit(EXIT_SUCCESS);
            break;
        }
        else
        {
            break;
        }
case GLFW_KEY_Q:
        if (action == GLFW_PRESS) {
            exit(EXIT_SUCCESS);
            break;
        }
        else
        {
            break;
        }

case GLFW_KEY_R:
        if (action == GLFW_PRESS) {
            restart();
            break;
        }
        else
        {
            break;
            }
```

在这段代码中，我实现了一些游戏控制的响应功能，以便玩家能够通过特定按键与游戏进行交互。当玩家按下 Esc 键（GLFW_KEY_ESCAPE）时，如果检测到按键的动作为按下（GLFW_PRESS），我将调用 exit(EXIT_SUCCESS) 函数以正常退出程序。同样地，当玩家按下 Q 键（GLFW_KEY_Q）时，程序也将执行退出操作，实现相同的效果。

此外，我为 R 键（GLFW_KEY_R）设置了重启游戏的功能。如果玩家按下该键，我将调用 restart() 函数，以重启游戏。这允许玩家在游戏进行中随时选择重新开始，而无需关闭和重新启动程序。

5. 其它扩展

5.1 计分功能

```
//消除成功，进行加分
    score += 10;
if (score > maxScore)
    maxScore = score;
```

我增加了游戏的计分功能以及历史最高分记录功能，增加游戏的趣味性以及竞技性，提高用户粘性。

5.2 逐渐提高游戏难度

```
//消除成功后进行加分
score += 10;
lowSpeed -= 0.05;    //加快降落速度
```

当获得加分后，会提高方块的下落速度，使得游戏难度提升，使玩家更能获得更高的分数，增加游戏挑战性。

5.3 增加游戏说明

```
void welcome() {
    // 欢迎信息与玩法说明
    cout << "欢迎来到 LXL 的俄罗斯方块，快来挑战我吧！\n\n";
    cout << "游戏玩法：\n 跟你以前玩的俄罗斯方块一样。随着分数增加，方块下落速度也会加快！\n\n";
    // 按键说明
    cout << "按键说明：\n"
        << "  上键    - 旋转方块\n"
        << "  左键    - 向左移动方块\n"
        << "  右键    - 向右移动方块\n"
        << "  下键    - 加速方块下落\n"
        << "  空格键  - 立即下落至底部\n"
        << "  Q 键 或 Esc 键 - 退出游戏\n"
        << "  R 键    - 重新开始游戏\n\n";
    // 当前游戏状态
    cout << "当前游戏状态：\n"
```

```
            << "   下落速度：" << lowSpeed << "\n"
            << "   当前分数：" << score << "\n";
}
// 重新启动游戏
void restart()
{
    system("cls");          //清除终端内容
    cout << "游戏已重新开始！\n 加油，特种兵！！！" << endl << endl;
    init();
}
```

当游戏开始以及游戏重新开始的时候，都会输出提示信息，给玩家更好的游戏体验。

6. 实验结果

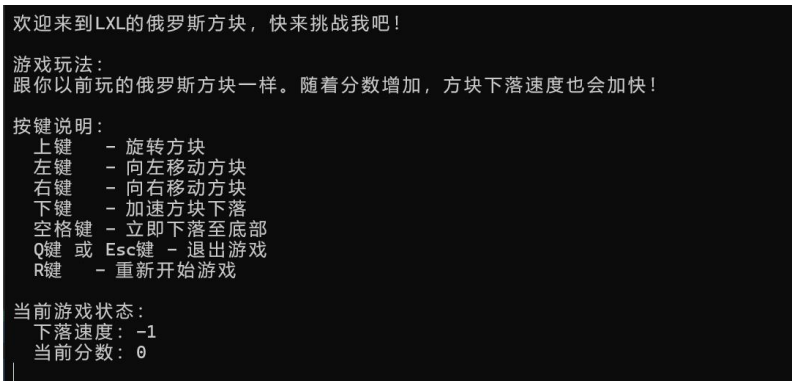

图 1 游戏说明

如图 1，点击运行，窗口会先输出游戏相关信息。

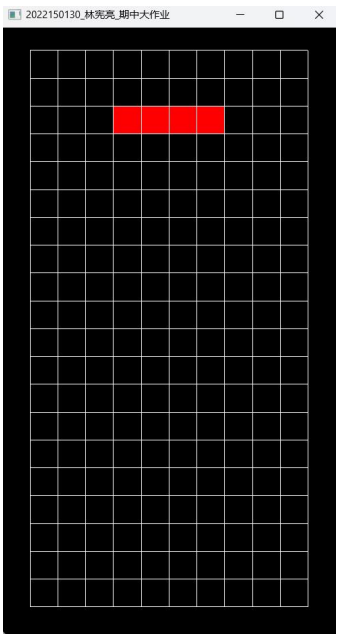

图 2 初始化

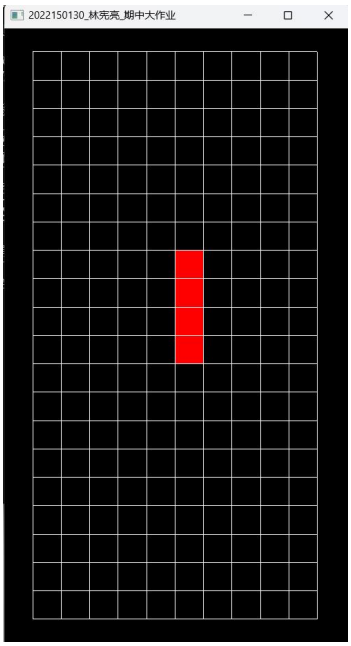

图 3 旋转

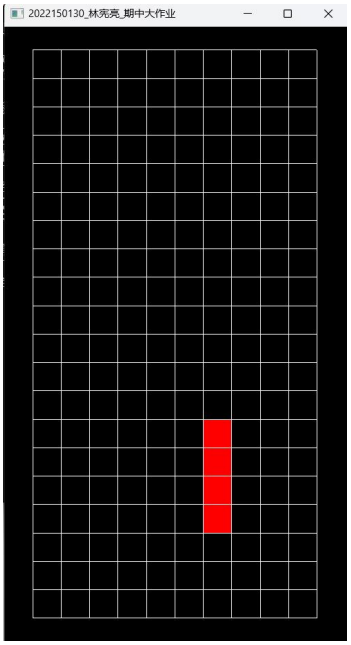

图 4 移动

如图，通过图 2，图 3 可以看出我设计的游戏实现了旋转功能，从图 3，图 4 可以看出我设计的游戏实现了移动的功能。从三幅图可以看出方块会自动下落。

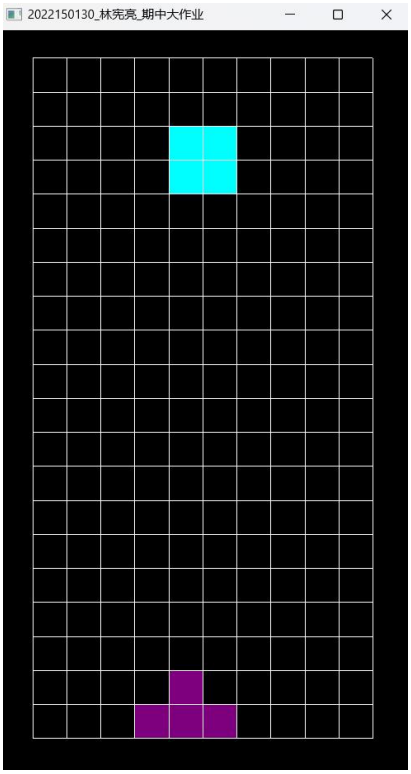

图 5 随机生成

结合图 5 可以看出我们的方块不管是颜色还是形状都是随机生成的。

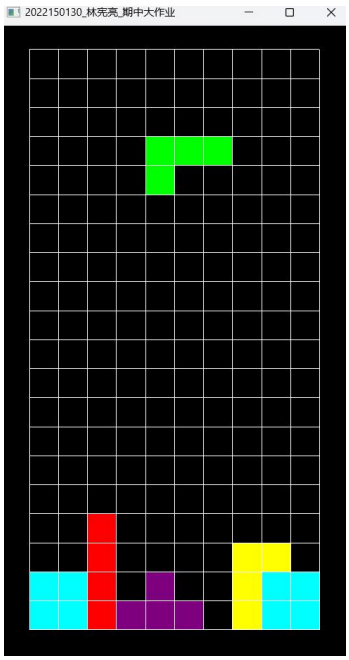

图 6 行消除（1）

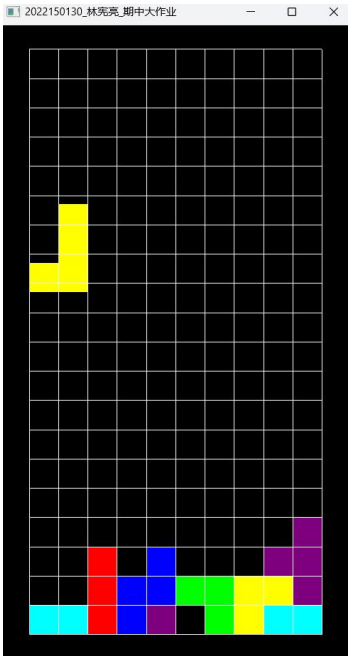

图 7 行消除（2）

结合图 6 和图 7，可以看出我的游戏具有行满即消除的功能。

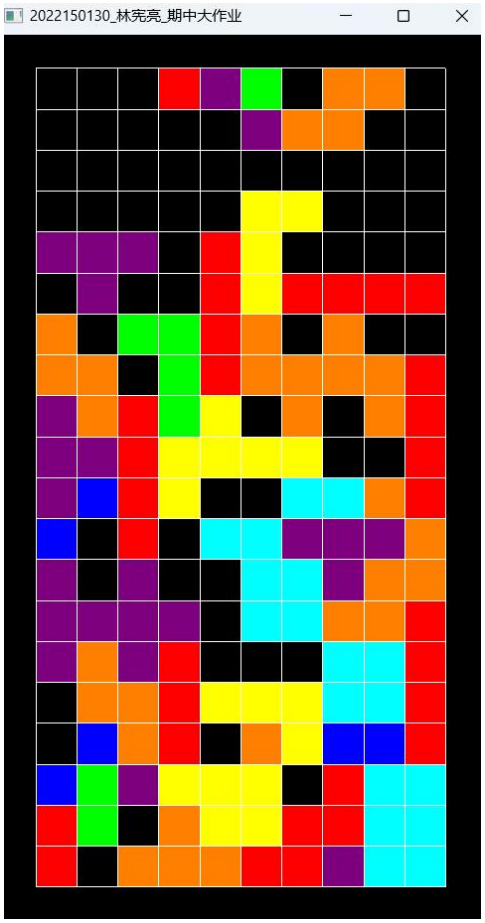

图 8 游戏画面

如图 8，当没有位置放放方块时游戏就会停止。

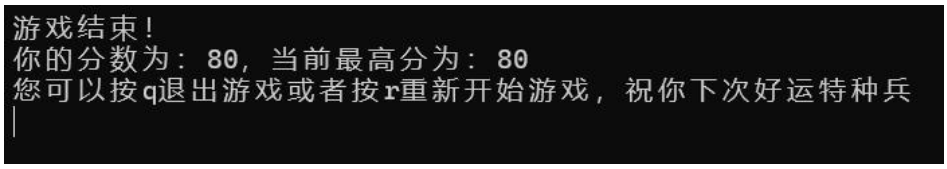

图 9 结算界面

如图 9，每局游戏结束都会输出本局得分和最高得分。

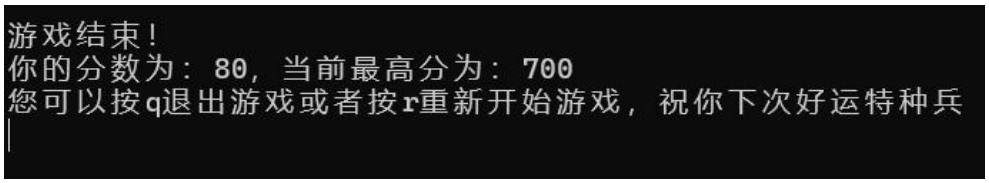

图 10 得分记录

如图 10，我最高已经得到 700 分，快来挑战我吧。

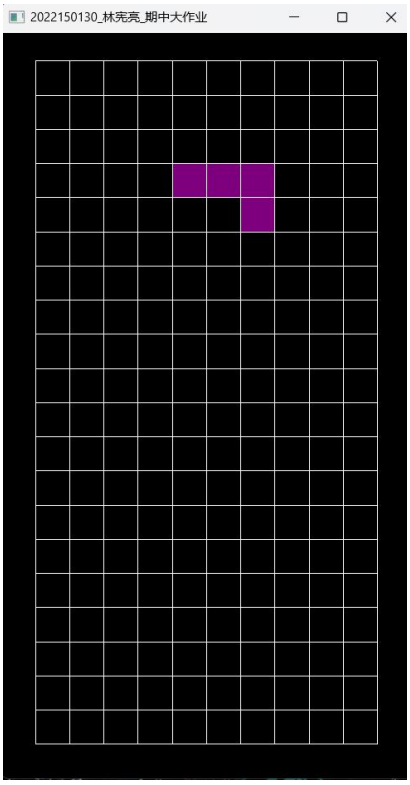

图 11 重新开始

可以点击 r 重新开始游戏，也可以点击 q 退出游戏。

综上所述，俄罗斯方块的旋转移动，碰撞检测等功能都已经实现，游戏功能完善。实验成功！

实验结论：

在此次俄罗斯方块游戏的实验中，我通过对代码的编写与优化，实现了多个关键功能，丰富了游戏体验。本实验的主要目标是开发一个具备基础功能和增强用户体验的俄罗斯方块游戏，并通过编写代码来实现方块的下落、移动、旋转等基本操作，同时添加多个特色功能以提升游戏的可玩性。

方块的随机生成与形状类型扩展：我实现了多种方块类型（包括 O 型、I 型、L 型等），每种方块都有独特的形状和颜色。通过随机生成方块，使得游戏不再单调，增加了游戏的挑战性和趣味性。

UI 优化：添加了详细的玩法介绍与按键说明，优化了 UI 设计，确保用户能够顺利理解和操作游戏。

方块的自动下落与手动控制：通过设置时钟，每隔一秒方块会自动下落一格，模拟了方块受重力作用的效果。玩家可以通过键盘的方向键实现方块的左右移动、旋转，以及加速下落的功能，提升了操作的流畅性。

空格键快速下落与重新开始功能：添加了空格键使方块能快速下落到底部的功能，简化操作。同时，我完善了 R 键重新开始游戏的功能，玩家能够在任何时候快速重置游戏。

行消除与梯度难度设计：当某一行填满时，该行会自动消除，并更新上方方块的位置。在消除方块的同时，游戏的难度会随着得分增加，方块下落的速度逐渐加快，增加了游戏的挑战性。

记分板与实时分数更新：每次消除方块后，分数会实时更新，增加了记分板功能，让玩家能够随时查看当前得分与最高分，进一步提升游戏的互动性。

通过本次实验，我不仅熟练掌握了 C++中的图形处理技术，还强化了 GLM 数学库在坐标变换中的应用。在设计与实现过程中，我深入理解了游戏开发的逻辑结构，尤其是游戏状态的管理、方块的移动检测以及碰撞处理等核心部分。同时，UI 优化与梯度难度设计使我对如何提升用户体验有了更深刻的理解。此次实验不仅达到了预期的目标，还让我在游戏开发和图形处理方面有了更多的实践经验。通过不断调试和优化，我实现了一个功能完善、界面友好的俄罗斯方块游戏，并且进一步理解了如何将算法逻辑与图形界面结合起来，创造出有趣的用户体验。

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。