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：软件工程（腾班）

指导教师：熊卫丹

报告人：洪子敬 学号：2022155033 班级：腾班

实验时间：2024 年 10 月 6 日 - 2024 年 10 月 30 日

实验报告提交时间：2024 年 10 月 23 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. **方块的绘制**

文档中给出了 7 种方块（“O”、“I”、“S”、“Z”、“L”、“J”和“T”），每种方块都有不同数量的形状变化，在写函数代码前先预定义好这些变量，使用 vec2 定义不同大小的数组进行存储，方块上的每一小块用一个 vec2 记录二维坐标，整体代码如下所示：

```
// 一个二维数组表示所有可能出现的方块和方向。
glm::vec2 allRotationsLshape[4][4] =
                                {{glm::vec2(0, 0), glm::vec2(-1,0), glm::vec2(1, 0), glm::vec2(-1,-1)}, //  "L"
                                {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)}};
// Do1:
glm::vec2 allRotationsJshape[4][4] =
                                {{glm::vec2(-1, 0), glm::vec2(0,0), glm::vec2(1, 0), glm::vec2(1,-1)},   //  "J"
                                {glm::vec2(0, -1), glm::vec2(0, 0), glm::vec2(0,1), glm::vec2(1, 1)},    //
                                {glm::vec2(-1, 0), glm::vec2(-1,1), 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 allRotationsTshape[4][4] =
                                {{glm::vec2(-1, 0), glm::vec2(0,0), glm::vec2(0, -1), glm::vec2(1,0)},   //  "T"
                                {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(0,1), glm::vec2(1, 0)},     //
                                {glm::vec2(-1,0), glm::vec2(0, -1), glm::vec2(0, 0), glm::vec2(0, 1)} };
glm::vec2 allRotationsOshape[4] = {glm::vec2(-1,-1), glm::vec2(-1,0), glm::vec2(0,-1), glm::vec2(0,0)};//  "O"
glm::vec2 allRotationsIshape[2][4] =
                                {{glm::vec2(-2,0), glm::vec2(-1,0), glm::vec2(0,0), glm::vec2(1,0)},     //  "I"
                                {glm::vec2(0,-2), glm::vec2(0,-1), glm::vec2(0,0), glm::vec2(0,1)} };    //
glm::vec2 allRotationsSshape[2][4] =
                                {{glm::vec2(-1,-1), glm::vec2(0,-1), glm::vec2(0,0), glm::vec2(1,0)},    //  "S"
                                {glm::vec2(0,0), glm::vec2(0,1), glm::vec2(1,-1), glm::vec2(1,0)} };     //
glm::vec2 allRotationsZshape[2][4] =
                                {{glm::vec2(-1,0), glm::vec2(0,-1), glm::vec2(0,0), glm::vec2(1,-1)},    //  "S"
HZJ                             {glm::vec2(0,-1), glm::vec2(0,0), glm::vec2(1,0), glm::vec2(1,1)} };     //
```

2. **方块的合法性检测**

为了防止方块之间、方块与边界之间的碰撞，编写函数 checkvalid 进行防范。主要的逻辑是判断当前方块不超过棋盘范围，同时不和已有方块重合；全局变量 board 数组存储着棋盘信息，棋盘上有方块是 true，没有是 false，所以通过真假即可分辨有无重叠。

函数整体代码如下：

```
// 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)
{
    //Do4:添加逻辑使得方块之间不能相互重叠
    if ((cellpos.x >= 0) && (cellpos.x < board_width) && (cellpos.y >= 0) && (cellpos.y < board_height) && !board[int(cellpos.x)][int(cellpos.y)])
        return true;
    else
        return false;
}
```

HZJ

3. 通过键盘控制方块的旋转

在棋盘有足够空间旋转当前方块的情况下，我们设置按向上键为对当前方块进行逆时针 90 度旋转，具体通过函数 rotate 实现。

思路：首先利用全局变量 type 判断当前方块的类型，并利用全局变量 rotation 计算得到方块的下一个旋转方向；接着判断此方向的旋转后的坐标是否合法，若合法，则更新当前方块，否则不做改变；最后更新方块的存储信息即可。由于不同类型的方块数量不一样，具体实现时要注意数量的不同，不能单纯复制粘贴。

部分代码展示如下所示：（代码较多，重复性较强，详细可见提交代码）

```
// 在棋盘上有足够空间的情况下旋转当前方块
void rotate()
{
    switch (type)
    {
        int nextrotation;
        case 0:
            // 计算得到下一个旋转方向
            nextrotation = (rotation + 1) % 4;
            // 检查当前旋转之后的位置的有效性
            if (checkvalid((allRotationsLshape[nextrotation][0]) + tilepos)
                && checkvalid((allRotationsLshape[nextrotation][1]) + tilepos)
                && checkvalid((allRotationsLshape[nextrotation][2]) + tilepos)
                && checkvalid((allRotationsLshape[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[rotation][i];
            }
            break;
        case 1:
            // 计算得到下一个旋转方向
            nextrotation = (rotation + 1) % 4;
            // 检查当前旋转之后的位置的有效性
            if (checkvalid((allRotationsJshape[nextrotation][0]) + tilepos)
                && checkvalid((allRotationsJshape[nextrotation][1]) + tilepos)
                && checkvalid((allRotationsJshape[nextrotation][2]) + tilepos)
                && checkvalid((allRotationsJshape[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] = allRotationsJshape[rotation][i];
            }
            break;
```

HZJ

```
        case 2:
            // 计算得到下一个旋转方向
            nextrotation = (rotation + 1) % 4;
            // 检查当前旋转之后的位置的有效性
            if (checkvalid((allRotationsTshape[nextrotation][0]) + tilepos)
                && checkvalid((allRotationsTshape[nextrotation][1]) + tilepos)
                && checkvalid((allRotationsTshape[nextrotation][2]) + tilepos)
                && checkvalid((allRotationsTshape[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] = allRotationsTshape[rotation][i];
            }
            break;
        case 3:
            break;
        case 4:
            // 计算得到下一个旋转方向
            nextrotation = (rotation + 1) % 2;
            // 检查当前旋转之后的位置的有效性
            if (checkvalid((allRotationsIshape[nextrotation][0]) + tilepos)
                && checkvalid((allRotationsIshape[nextrotation][1]) + tilepos)
                && checkvalid((allRotationsIshape[nextrotation][2]) + tilepos)
                && checkvalid((allRotationsIshape[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] = allRotationsIshape[rotation][i];
            }
            break;
```

HZJ

4. 新方块的生成与方块颜色绘制

在判断一个方块无法继续下落时，这时需要重新生成一个新的方块，对应编写 newtile 函数。在前面我们知道一共有 7 种类型的方块，这里我们采用随机生成的方法，7 种中随机选择一种，在此基础上，随机选择一种方向的方块即可。此处要注意每次生成新方块时要更新 type 的值，同时更新方块的信息，整体代码如下所示：

```
// 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;
// Do2
// 随机选择形状
int shapeIndex = rand() % 7;
switch (shapeIndex)
{
    case 0: //L形状
        rotation = rand() % 4;
        for (int i = 0; i < 4; i++) {
            tile[i] = allRotationsLshape[rotation][i];
        }
        type = 0;
        break;
    case 1: //J形状
        rotation = rand() % 4;
        for (int i = 0; i < 4; i++) {
            tile[i] = allRotationsJshape[rotation][i];
        }
        type = 1;
        break;
    case 2: //T形状
        rotation = rand() % 4;
        for (int i = 0; i < 4; i++) {
            tile[i] = allRotationsTshape[rotation][i];
        }
        type = 2;
        break;
    case 3: //O形状
        for (int i = 0; i < 4; i++) {
            tile[i] = allRotationsOshape[i];
        }
        type = 3;
        break;
```

HZJ

```
    case 4: //I形状
        rotation = rand() % 2;
        for (int i = 0; i < 4; i++) {
            tile[i] = allRotationsIshape[rotation][i];
        }
        type = 4;
        break;
    case 5: //S形状
        rotation = rand() % 2;
        for (int i = 0; i < 4; i++) {
            tile[i] = allRotationsSshape[rotation][i];
        }
        type = 5;
        break;
    case 6: //Z形状
        rotation = rand() % 2;
        for (int i = 0; i < 4; i++) {
            tile[i] = allRotationsZshape[rotation][i];
        }
        type = 6;
        break;
}

updatetile();

//判断游戏是否结束
if (!checkvalid(tilepos + tile[0])
    ||!checkvalid(tilepos + tile[1])
    || !checkvalid(tilepos + tile[2])
    || !checkvalid(tilepos + tile[3]))
{
    gameover = true;
    return;
}
```

HZJ

上面代码其实我们还看到函数结尾还判断了游戏是否结束，这是因为当新的方块生成时很可能会使当前方块超过棋盘（因为我们设置方块是从棋盘中间生成的），此时就会导致游戏结束；具体实现时设置全局标志变量 gameover 为 true 即可，待到下次主函数判断就会自行退出程序。

同时此处我们还要设置方块的颜色，为了界面的美观，我们使用随机 RGB 值对方块进行赋值，但限定 RGB 值范围在 0.5-1 之间，这是为了使颜色明亮一些，防止与背景黑色相近。详细代码如下所示：

```
// 给新方块随机赋上颜色
// 明亮颜色
glm::vec4 newColor = glm::vec4(static_cast<float>(rand()) / RAND_MAX * 0.5f + 0.5f, // 0.5 - 1.0 范围
                               static_cast<float>(rand()) / RAND_MAX * 0.5f + 0.5f,
                               static_cast<float>(rand()) / RAND_MAX * 0.5f + 0.5f, 1.0);
glm::vec4 newcolours[24];
for (int i = 0; i < 24; i++)
    newcolours[i] = newColor;

glBindBuffer(GL_ARRAY_BUFFER, vbo[5]);
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(newcolours), newcolours);
glBindBuffer(GL_ARRAY_BUFFER, 0);

glBindVertexArray(0);
```

HZJ

5. **方块的自动向下移动与下落速度设置**

在主函数的 while 循环中，每次都 display 更新着当前的棋盘上方块的情况，为了实现方块的自动下落，需要在每次的 display 之前进行判断方块是否需要下落，这里我们通过时间来判断自动下落；通过记录前后的时间变化是否超过定义的方块下落间隔（此处设置为 0.5s），若超过则判断当前游戏是否结束，若未结束，则调整方块往下移动一格。

此外，这里还设置了方块下落速度的调整，有两个档位，正常档每次自动下落一格，速度档每次自动下落 1.5 格，通过一个标志变量 speed 连接，并绑定按键“S”；每次按一下“S”键都可以在两个档位来回切换。

主函数中此部分代码如下：

```
//定义方块下落的时间间隔
float dropInterval = 0.5f;
float lastDropTime = 0.0f;
while (!glfwWindowShouldClose(window))
{
    float currentTime = glfwGetTime();
    //判断是否方块该落下
    if (currentTime - lastDropTime >= dropInterval) {
        //判断方块能否落下一格
        bool judge;
        if (!speed)
            judge = movetile(glm::vec2(0, -1));
        else
            judge = movetile(glm::vec2(0, -1.5));
        if (!judge) {
            //不能落下的话 重新生成新的方块
            settile();
            newtile();
            if (gameover) {
                std::cout << "-------游戏结束!-------" << std::endl;
                std::cout << "-------分数: " << scores <<"-------"<< std::endl;
                exit(0);
            }
        }
        lastDropTime = currentTime;
    }
    display();
    glfwSwapBuffers(window);
    glfwPollEvents();
```

HZJ

6. 方块的放置与棋盘中行满时的自动消除

当某一行方块填满时，我们需要对这一行进行消除，并将上面的方块往下移动。**由于此坐标系的原点在左下角，所以横轴时行，纵轴是列。**

编写函数 checkfullrow，对输入的某一行进行检测，通过判断此行每个格子的布尔值来判断是否都填满，若未填满，则返回 false，否则返回 true。若填满了，则对该行进行消除（将每个格子布尔值更改为 false，并把颜色改为 black 即可），同时将上方的格子整体往下移动即可。函数代码如下：

```
// 检查棋盘格在row行有没有被填充满
bool checkfullrow(int row)
{
    for (int j = 0; j < board_width; j++) {
        if (!board[j][row]) return false; // 只要有一个格子未填满，返回
    }

    // 消除该行
    for (int j = 0; j < board_width; j++) {
        board[j][row] = false; // 清空该行
        changecellcolour(glm::vec2(j, row), black); // 更新颜色
    }
    scores += 10; // 更新分数
    std::cout << "当前分数: " << scores << std::endl; // 输出当前分数

    // 向下移动填充的行
    for (int i = row + 1; i < board_height; i++) {
        for (int j = 0; j < board_width; j++) {
            if (board[j][i]) {
                board[j][i - 1] = true; // 移动到下方
                changecellcolour(glm::vec2(j, i - 1), orange); // 更新颜色
                board[j][i] = false; // 清空当前行
                changecellcolour(glm::vec2(j, i), black); // 更新颜色
            }
        }
    }
    return true;
}
```

HZJ

编写函数 settile，对变化后方块的状态进行更新，即方块位置的更新，格子在棋盘上的状态设置为填充状态（true）以及颜色改为 orange（**本实验方块无法移动后颜色均变为橙色**）。同时需要注意，此时由于又有方块的加入，很可能导致某一行被填满了，所以需要利用 checkfullrow 判断棋盘的状态。函数代码如下：

```
// 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), orange);
    }

    // 检查是否有行被填满并消除
    for (int i = 0; i < board_height; i++) {
        if (checkfullrow(i))
            i -= 1;
    }
}
```

HZJ

7. 游戏的重启、暂停和分数的统计

分数统计功能：每次消除一行可以获得 10 分的分数，并且终端会输出当前的分数；**暂停功能**：绑定了按键“P”，通过按此键可以暂停游戏，再按一下则会正常游戏，具体通过全局变量 pause 影响 display 函数的渲染来实现；**重启游戏功能**：绑定了按键“R”，通过按此键可以重启游戏，此时需要重置游戏结束标志变量 gameover、棋盘状态数组 board、速度标志变量 speed（恢复为正常速度）等以及调用 init 函数进行函数重启。

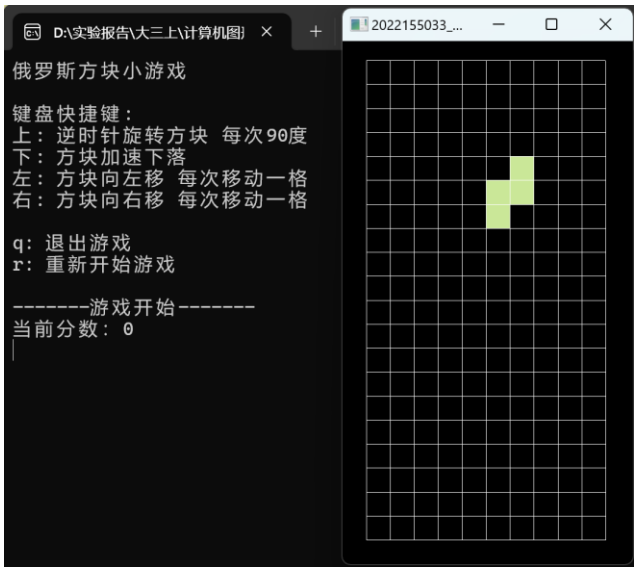
```
void restart()
{
    gameover = false;
    for (int i = 0; i < board_width; i++) {
        for (int j = 0; j < board_height; j++) {
            board[i][j] = false;
        }
    }
    pause = false;
    scores = 0;
    speed = false;
    init();
    std::cout << "-------重启游戏--------" << std::endl;
    std::cout << "当前分数: 0" << std::endl;
}
```

HZJ

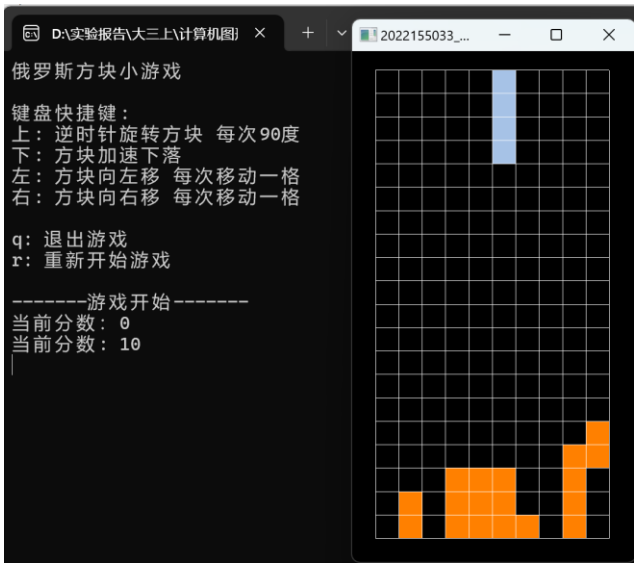
```
void display()
{
    if (pause) {
        return;
    }
    glClear(GL_COLOR_BUFFER_BIT);

    glUniform1i(locxsize, xsize);
    glUniform1i(locysize, ysize);

    glBindVertexArray(vao[1]);
    glDrawArrays(GL_TRIANGLES, 0, points_num); // 绘制棋
    glBindVertexArray(vao[2]);
    glDrawArrays(GL_TRIANGLES, 0, 24);   // 绘制当前方块
    glBindVertexArray(vao[0]);
    glDrawArrays(GL_LINES, 0, board_line_num * 2 );
```

HZJ

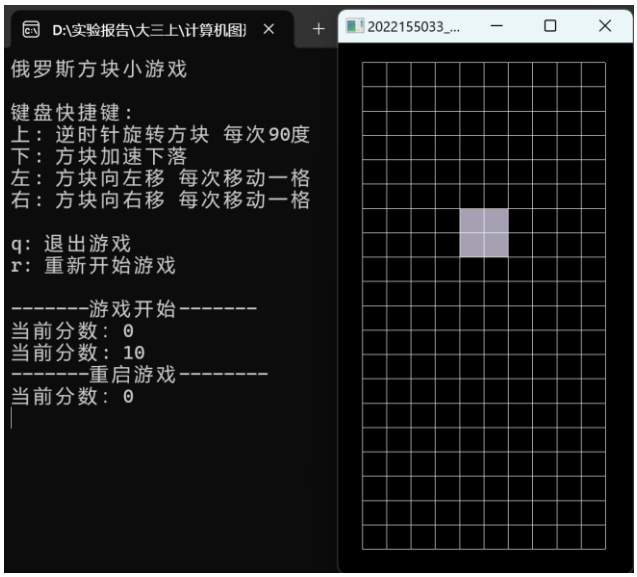
实验结果展示：（由于展示效果有限，详细请见提交的 main.exe 文件）



（游戏开始时）



（消除一行时）



（重启游戏时）

实验结论： 本次实验利用 OpenGL 做了一个简化版的俄罗斯方块，强化了个人对 OpenGL 的基本绘制方法、键盘等交互事件的响应逻辑的理解。在完成给定要求基础上，本人还添加了对**方块颜色美观的改进、方块的加速下落、游戏的暂停以及分数的统计**，较为成功地完成了本次实验。本次实验圆满结束。
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。