**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： 胡瑞珍**

**报告人： 学号： 班级：**

**实验时间：2022年 09月26日 -- 2022年 10月23 日**

**实验报告提交时间： 2022年 10月19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：  本次实验除了留空代码中已实现的功能，我还实现了以下几个功能。  \* - 自己实现的功能如下：   \* - 1) 绘制‘J’、‘Z’等七种形状的方块（allRotationsLshape[7][4][4]）   \* - 2) 随机生成方块并赋上了七种不同的颜色(colors[7])   \* - 3) 方块的自动向下移动（falling函数实现）   \* - 4) 方块之间、方块与边界之间的碰撞检测（checkvaild函数实现）   \* - 5) 棋盘格中每一行填充满之后自动消除，并计算积分，增加游戏乐趣（checkfullrow函数实现）   \* - 6) 实现了游戏的暂停与恢复，方便玩家   \* - 7) 方块向上移动，作为辅助技能   \* - 8) 每达到一定积分就加快方块下落速度，挑战升级   1. 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。  对棋盘格的渲染包括：绘制纵向线、水平线、将所有线赋成白色、将所有格子填充成黑色，进行各变量的初始化设置。  // 绘制网格线      // 纵向线      for (int i = 0; i < (board\_width+1); i++)      {          gridpoints[2\*i] = glm::vec4((tile\_width + (tile\_width \* i)), tile\_width, 0, 1);          gridpoints[2\*i + 1] = glm::vec4((tile\_width + (tile\_width \* i)), (board\_height+1) \* tile\_width, 0, 1);      }      // 水平线      for (int i = 0; i < (board\_height+1); i++)      {          gridpoints[ 2\*(board\_width+1) + 2\*i ] = glm::vec4(tile\_width, (tile\_width + (tile\_width \* i)), 0, 1);          gridpoints[ 2\*(board\_width+1) + 2\*i + 1 ] = glm::vec4((board\_width+1) \* tile\_width, (tile\_width + (tile\_width \* i)), 0, 1);      }      // 将所有线赋成白色      for (int i = 0; i < (board\_line\_num \* 2); i++)          gridcolours[i] = white;      // 初始化棋盘格，并将没有被填充的格子设置成黑色      glm::vec4 boardpoints[points\_num];      for (int i = 0; i < points\_num; i++)          board\_colours[i] = black;  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。  在生成随机方块前需要判断游戏是否结束。游戏结束的标志为棋盘最上方两行的中间6格是否被填充，被填充，则isfull设置为true，游戏结束，gameover设置为true；未被填充则isfull设置为false，游戏继续，生成新方块。  // 判断游戏是否结束      bool isfull = false; //判断上面二行中间是否被填满，填满则游戏结束      for (int i = 18; i < 20; i++) { //最上面两行          for (int j = 4; j < 7; j++) { //中间几格              if (board[j][i] == true)                  isfull = true;          }      }      // 游戏结束      if (isfull == true) {          cout <<"游戏结束，按下R键重新开始游戏，按下Q或Esc键退出游戏。"<< endl;          gameover = true;          return;      }  生成新方块，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，利用rand()函数及srand()函数生成随机数，获取随机数对应的方块形状作为新生成的方块，同样利用rand()函数生成随机数，获取方块的颜色。    // 游戏未结束      // 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;      // 利用随机数获取新方块形状      srand(time(0)); //改变随机数种子      ishape = (starttime + rand()) % 7; //随机方块      for (int i = 0; i < 4; i++)      {          tile[i] = allRotationsLshape[ishape][rotation][i]; // 新方块显示为第一种旋转形态      }      updatetile();      // 给新方块赋上颜色      icolor = (starttime + rand()) % 7;//随机颜色      glm::vec4 newcolours[24];      for (int i = 0; i < 24; i++)          newcolours[i] = colors[icolor];  方块自动下落。利用glutGet(GLUT\_ELAPSED\_TIME)函数记录当前时间，用current\_time - starttime > num \* fall\_interval的值判断方块是否需要下落，需要下落则调用movetile()函数使方块下移一格，如果移动失败则说明已碰撞到其他方块，需要重新生成新方块。  //方块自动下落  void falling(void)  {      //falling函数会不断地执行，时间也会不断地增加      current\_time = glutGet(GLUT\_ELAPSED\_TIME);      // 由于时间不断的增加，我们用时间是否大于num乘以fall\_interval来判断是否要进行下落操作      // 从而实现每 fall\_interval 的时间间隔方块自动向下掉落，随着难度的增加fall\_interval会越来越小，方块下落速度越来越快      if (current\_time - starttime > num \* fall\_interval) {          num++;          if (gameover == false && gamestop == false) {//当游戏还在进行并且没有暂停时              if (!movetile(glm::vec2(0, -1)))//自动下落              {                  settile();                  newtile();              }          }      }  }  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。  按下键盘向下按键移动方块可以达到控制方块向下移动的速度的目的，要使方块向上移动，只需修改movetile()函数的第二个参数为-1。  case GLFW\_KEY\_DOWN: // 向下按键移动方块                  if ((action == GLFW\_PRESS || action == GLFW\_REPEAT) && gamestop == false){                      if (!movetile(glm::vec2(0, -1)))                      {                          settile();                          newtile();                          break;                      }                      else                      {                          break;                      }                  }  额外设置了按下U键使得方块向上移动的附加技能。要使方块向上移动，只需修改movetile()函数的第二个参数为1。  case  GLFW\_KEY\_U: // U键向上移动方块                  if (action == GLFW\_PRESS) {                      movetile(glm::vec2(0, 1));                      break;                  }                  else                  {                      break;                  }   1. 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。  使用movetile()函数来处理碰撞叠加，首先计算移动之后方块的位置坐标，使用checkvalid()函数判断移动之后是否有效，如果无效则说明碰撞或者超出范围，不进行移动操作；如果有效则移动方块，接着调用updatetile()函数更新VBO。  // 给定位置(x,y)，移动方块。有效的移动值为(-1,0)，(1,0)，(0,-1)，分别对应于向左，向下和向右移动。如果移动成功，返回值为true，反之为false  bool movetile(glm::vec2 direction)  {      // 计算移动之后的方块的位置坐标      glm::vec2 newtilepos[4];      for (int i = 0; i < 4; i++)          newtilepos[i] = tile[i] + tilepos + direction;      // 检查移动之后的有效性      if (checkvalid(newtilepos[0])          && checkvalid(newtilepos[1])          && checkvalid(newtilepos[2])          && checkvalid(newtilepos[3]))          {              // 有效：移动该方块              tilepos.x = tilepos.x + direction.x;              tilepos.y = tilepos.y + direction.y;              updatetile();              return true;          }      return false;  }  checkvalid()函数检查在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，需要判断将要填充的格子是否已被填充，是否超出棋盘边界范围。  // 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)  {      //添加一个判断：格子是否已被填充      if((board[int(cellpos.x)][int(cellpos.y)] == false&&cellpos.x >=0) && (cellpos.x < board\_width) && (cellpos.y >= 0) && (cellpos.y < board\_height) )          return true;      else          return false;  }   1. 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。  在游戏还没有结束的情况下，按“上”键，调用rotate()函数，使方块以旋转中心顺（逆）时针旋转，每次旋转90°。  // 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)&&gamestop==false)                  {                      rotate();                      break;                  }                  else                  {                      break;                  }  rotate()函数，在棋盘上有足够空间的情况下旋转当前方块，首先计算下一个要旋转的方向，判断旋转后的位置是否有效，有效则对方块进行旋转，即更新成旋转之后的方块，更新VBO；无效则不进行任何操作。  // 在棋盘上有足够空间的情况下旋转当前方块  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[ishape][nextrotation][0]) + tilepos)          && checkvalid((allRotationsLshape[ishape][nextrotation][1]) + tilepos)          && checkvalid((allRotationsLshape[ishape][nextrotation][2]) + tilepos)          && checkvalid((allRotationsLshape[ishape][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[ishape][rotation][i];          updatetile();      }  }  按“左”和“右”键分别将方块向左/右方向移动一格，将movetile(glm::vec2(-1, 0))函数的第一个参数分别设为-1（向左）和1（向右）。              case GLFW\_KEY\_LEFT:  // 向左按键移动方块                  if ((action == GLFW\_PRESS || action == GLFW\_REPEAT) && gamestop == false){                      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) && gamestop == false){                      movetile(glm::vec2(1, 0));                      break;                  }                  else                  {                      break;                  }  按“下”键加速方块移动。  case GLFW\_KEY\_DOWN: // 向下按键移动方块                  if ((action == GLFW\_PRESS || action == GLFW\_REPEAT) && gamestop == false){                      if (!movetile(glm::vec2(0, -1)))                      {                          settile();                          newtile();                          break;                      }                      else                      {                          break;                      }                  }   1. 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。   使用checkfullrow()函数判断row行是否已被填充满，遍历row行的每一个格子，只要有一个格子没被填充，则将isfull设置为false，不进行任何操作；若格子全被填充，则isfull仍为true，需要进行消行操作，我额外设置了积分值的计算，因此每消除一行，积分point++，且积分每升10，难度等级升1，即方块下落速度变快。因此还需要判断增加后当前积分值是否达到了升难度的要求：point >= hard\_level \* 10，如果达到了，则难度等级hard\_level++，方块下落速度通过将下落时间间隔减小fall\_interval /= 2来设置。被消除一行后，还需要将该行格子颜色设成黑色，并且设为未填充状态，然后将上方的所有方格全部下移。  // 检查棋盘格在row行有没有被填充满  void checkfullrow(int row)  {      bool isfull = true;      for (int j = 0; j < 10; j++) {          if (board[j][row] == false) {              isfull = false; //这一行有格子没被填满则说明不能消行          }      }      // 被填满，消行，积分增加      if (isfull == true) {          point += 1;          cout << "你现在已有 " << point <<" 积分"<< endl;          // 积分数量每增加10，难度增加1，方块下落速度加快          if (point >= hard\_level \* 10) {              hard\_level++;              fall\_interval /= 2; //调整下落时间间隔              num \*= 2; //与自动下落有关，使得下落时间间隔和下落数量的乘积与之前相同              cout << "你现在所处第 " << hard\_level << " 级难度" << endl;          }          // 处理被消除一行          for (int j = 0; j < 10; j++) {              changecellcolour(glm::vec2(j, row), black); //格子颜色全部变黑              board[j][row] = false; //该行全部变成未填满状态          }          // 将现有全部方块下移          for (int k = row + 1; k < 20; k++) {              for (int j = 0; j < 10; j++) {                  if (board[j][k] == true) { //如果上一行有方块，则需要往下移一行                      board[j][k] = false; //将上一行置为未填满状态                      changecellcolour(glm::vec2(j, k - 1), board\_color[j][k]);                      //下一行对应位置的颜色变成上一行对应位置的颜色                      changecellcolour(glm::vec2(j, k), black); //上一行清空为黑色                      board[j][k - 1] = true; //下一行对应位置设置为填满状态                  }              }          }      }  }  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。  case GLFW\_KEY\_ESCAPE:                  if(action == GLFW\_PRESS){                      cout << "您已退出游戏，欢迎下次再来玩。" << endl;                      exit(EXIT\_SUCCESS);                      break;                  }                  else                  {                      break;                  }              case GLFW\_KEY\_Q:                  if(action == GLFW\_PRESS){                      cout << "您已退出游戏，欢迎下次再来玩。" << endl;                      exit(EXIT\_SUCCESS);                      break;                  }                  else                  {                      break;                  }                case GLFW\_KEY\_R:                  if(action == GLFW\_PRESS){                      restart();                      break;                  }                  else                  {                      break;                  }  重新开始游戏需要先将所有格子清空，即格子填充状态设为false，颜色设为黑色，其他变量重新初始化。  // 重新启动游戏  void restart()  {      // 先将所有格子清空      for (int i = 0; i < 10; i++) {          for (int j = 0; j < 20; j++) {              board[i][j] = false; // 所有格子填充状态设为false              board\_color[i][j] = black; // 所有格子颜色设为黑色              changecellcolour(glm::vec2(i, j), black);          }      }      starttime = glutGet(GLUT\_ELAPSED\_TIME);      hard\_level = 1;      fall\_interval = 1000; //难度和下落时间重新设立      num = 1; //初始化      gamestop = false; //设置游戏为运行状态      gameover = false; //结束游戏的标志置为false      point = 0; //积分清零      cout << "您已重启游戏，积分已清零，难度已恢复为1" << endl;      cout << "你现在已有 " << point << " 积分，" << "处于第 " << hard\_level << " 级难度" << endl;      newtile();  }   1. 自己另外实现的功能。   （1）按U键向上移动方块。  case  GLFW\_KEY\_U: // U键向上移动方块                  if (action == GLFW\_PRESS) {                      movetile(glm::vec2(0, 1));                      break;                  }                  else                  {                      break;                  }  （2）按S键实现游戏的暂停与继续。  case GLFW\_KEY\_S: //实现游戏的暂停                  if (action == GLFW\_PRESS) {                      if (gamestop == false) {                          gamestop = true;                          cout << "你已暂停游戏!" << endl;                      }                      else {                          gamestop = false;                          cout << "游戏继续!" << endl;                      }                      break;                  }                  else                  {                      break;                  }  （3）增加积分设置，根据积分增加游戏难度（方块下落速度）。  // 积分数量每增加10，难度增加1，方块下落速度加快          if (point >= hard\_level \* 10) {              hard\_level++;              fall\_interval /= 2; //调整下落时间间隔              num \*= 2; //与自动下落有关，使得下落时间间隔和下落数量的乘积与之前相同              cout << "你现在所处第 " << hard\_level << " 级难度" << endl;          }   1. 游戏逻辑   游戏开始后，方块会从正上方自动下落，可以通过左/右/下/U键实现方块的左/右/下/上移，可以按S键暂停游戏，再按S键可继续游戏，按R键重新开始游戏，按Q或esc键退出游戏，方块每填充满一行，自动消除，并且积分加1，积分每增加10，难度增加1，方块下落速度变成原来的两倍，起始积分为0，难度为1级。  以下展示一下我试玩的过程：  从终端可以看出先是输出了游戏指导语，告知玩家游戏规则及按键操作，接着开始游戏。游戏开始没多久，还没得到积分时，我就按下R键重启了游戏。重新开始游戏后，累积到5积分后我按下S键暂停了游戏，接着继续游戏。当我累积到10积分时，提示我进入第2难度，我发现方块下落的速度明显加快了，于是对于玩俄罗斯方块不怎么强的我频繁使用U键使方块上移以给自己更多思考的时间。最后当积分累积到13积分时，游戏结束，我按下R键重启游戏。    重启游戏后，积分清零，难度级数恢复为1。我继续游戏，直到累积到17积分游戏结束，按下Q键退出游戏。      游戏中途截图 游戏结果截图 |

深圳大学学生实验报告用纸

|  |
| --- |
| 实验结论：  实验所要求实现的功能：   1. 绘制‘J’、‘Z’等形状的方块 2. 随机生成方块并赋上不同的颜色 3. 方块的自动向下移动 4. 方块之间、方块与边界之间的碰撞检测 5. 棋盘格中每一行填充之后自动消除（积分值增加代表有行被消除）       以上功能均已实现。其他功能如下：   1. U键向上移动（一般俄罗斯方块游戏中没有，此设置为了辅助新手）。 2. 游戏暂停与继续（方便玩家，有别的急事需要离开不必重启游戏）。      1. 计算积分值，积分值累计到一定数目，难度增加，方块下落速度加快。（积分制、难度值增加了玩家玩游戏的兴趣、乐趣和挑战性）       实验体会：本实验留空代码已经写好了部分函数，这使得我能更快地理解整体框架，明白要实现所要的功能需要设置哪些条件。通过运行留空代码清楚哪些代码对应已实现功能，然后研究代码含义。首先需要设置合适的变量，初始化各变量，再来编写对应的函数实现。还好之前有玩过俄罗斯方块，对游戏的逻辑有了解，因此除了完成老师要求的任务后，我还试玩了游戏，并根据自己的游戏体验为游戏增加了更多功能，如使方块上移，实现游戏暂停与继续，增加积分制和难度值等。通过试玩游戏，也能让我发现程序中存在的bug，如按键未响应等，找到问题所在，可以快速改正代码。为了使玩家知晓游戏规则，在游戏开始前应该告知，并指导玩家如何进行操作。本来还想实现消除一行后闪出特效，奈何能力有限，做不到想要的效果，希望通过以后的学习可以实现！ |