

# DMFB模拟系统

---

洪昊昀 计82 2017010591

---

## 1. 项目简介

本项目运用 `Qt 5.13.0` 编写，界面十分精美，作为一个数字微流控生物芯片模拟界面，通过读入文件指令模拟液滴在芯片上的流动与变化过程。

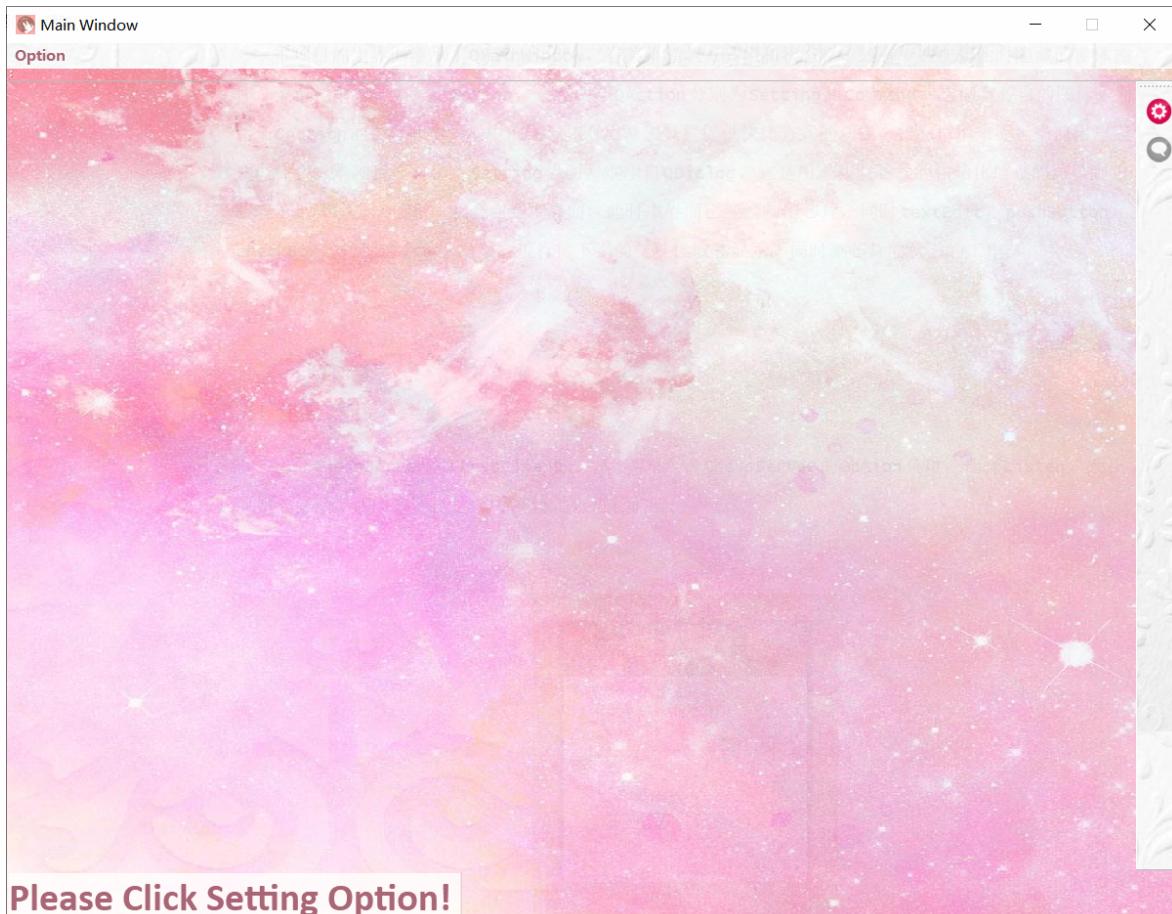
本项目中，除了完成助教布置的任务外，我也有根据自己的理解，做出的自定义的附加功能，这些内容会在第3部分用淡蓝色高亮标出

## 2. 运行方式

安装 `Qt creator` 后，将源代码拷贝至本机，双击文件根目录的 `BigProject` 文件夹下的 `BigProject.pro` 文件，直接用 `Qt creator` 编译即可。

## 3. 功能演示与实现思路

本项目的主界面继承了 `QMainWindow`，有菜单栏、工具栏和状态栏，还有一些在初始时隐藏的窗体控件。菜单栏中有一个菜单 `Option`，它有两个 `QAction` 分别为 `Setting` 和 `Command`，它们有各自的图标。而两个 `QAction` 的图标和工具栏中的图标相对应，工具栏提供快捷方式的功能。状态栏用来提示当前用户需要做什么。本项目还有一个 `Setting` 界面，继承自 `QDialog`，它是用来获取芯片的初始化信息的。等到它获取了足够的合法信息，就会回到主界面，此时主界面已经绘制出芯片，同时 `TextEdit`，`pushButton`，`checkbox` 等控件也会显示出来，进行接下来的文件读取和显示不同时刻芯片的状态的工作。



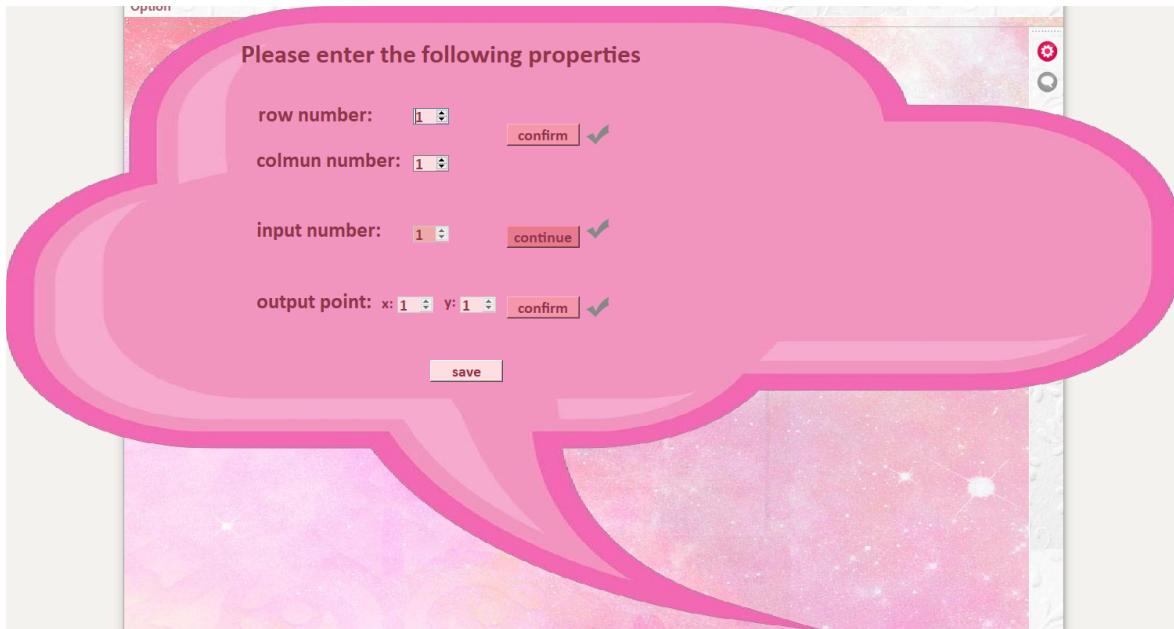
我的代码结构如下图：

```
BigProject [master]
├── BigProject.pro
└── Headers
    ├── line.h
    ├── matrix.h
    ├── matrixcomb.h
    ├── mymainwindow.h
    ├── node.h
    ├── operation.h
    ├── operationsignal.h
    ├── setinputdialog.h
    └── settingwidget.h
└── Sources
    ├── line.cpp
    ├── main.cpp
    ├── matrix.cpp
    ├── matrixcomb.cpp
    ├── mymainwindow.cpp
    ├── node.cpp
    ├── operation.cpp
    ├── setinputdialog.cpp
    └── settingwidget.cpp
> Forms
> Resources
> Other files
```

除了 `mymainwindow`, `setinputdialog`, `settingwidget` 之外, 都是用来处理数据的自定义类。`line` 是用来处理指令的, `matrix` 代表的是芯片的某一格, 它里面封装了很多属性, `matrixcomb` 里面有  $m \times n$  个 `matrix` 对象, 相当于存了芯片某个时刻的一张照片, `node` 是用来处理清洁模式的, `operation` 里面涵盖了上述所有的处理数据的类, 将 `mymainwindow` 里面的数据处理操作和绘图分割开来。

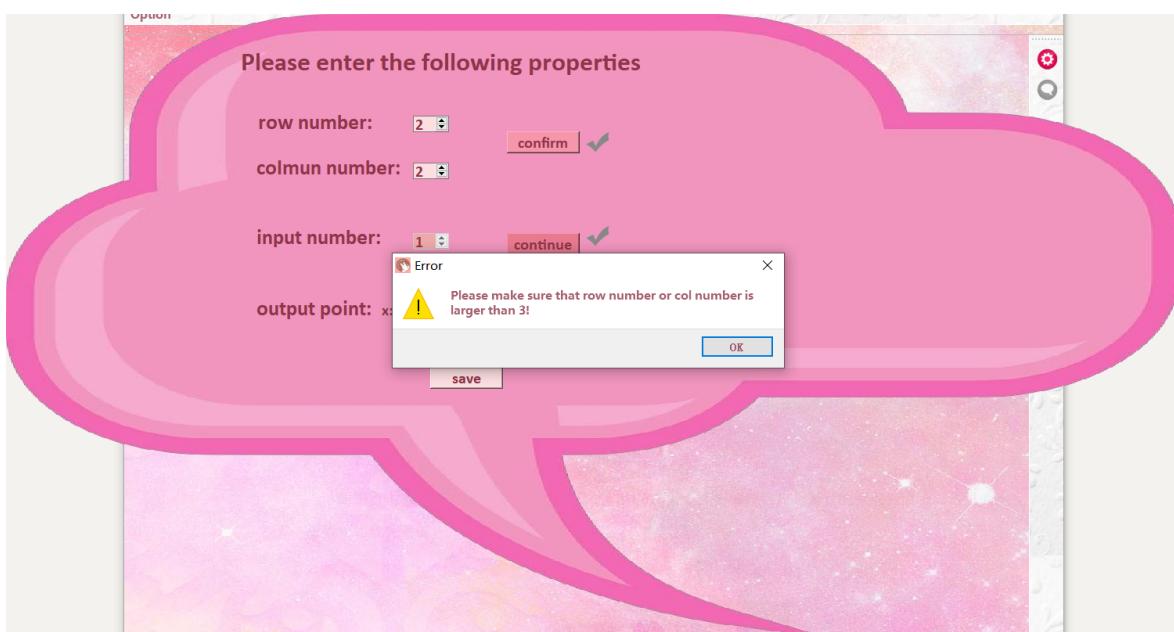
### 3.1 设定初始值

运行 `BigProject` 程序, 首先点击 `Main window` 的 `Option` 菜单, 选择 `Setting`, 或者也可以点击停靠在页面右侧的工具栏的第一个图标, 会出现如下窗口, 让用户进行芯片参数的输入:

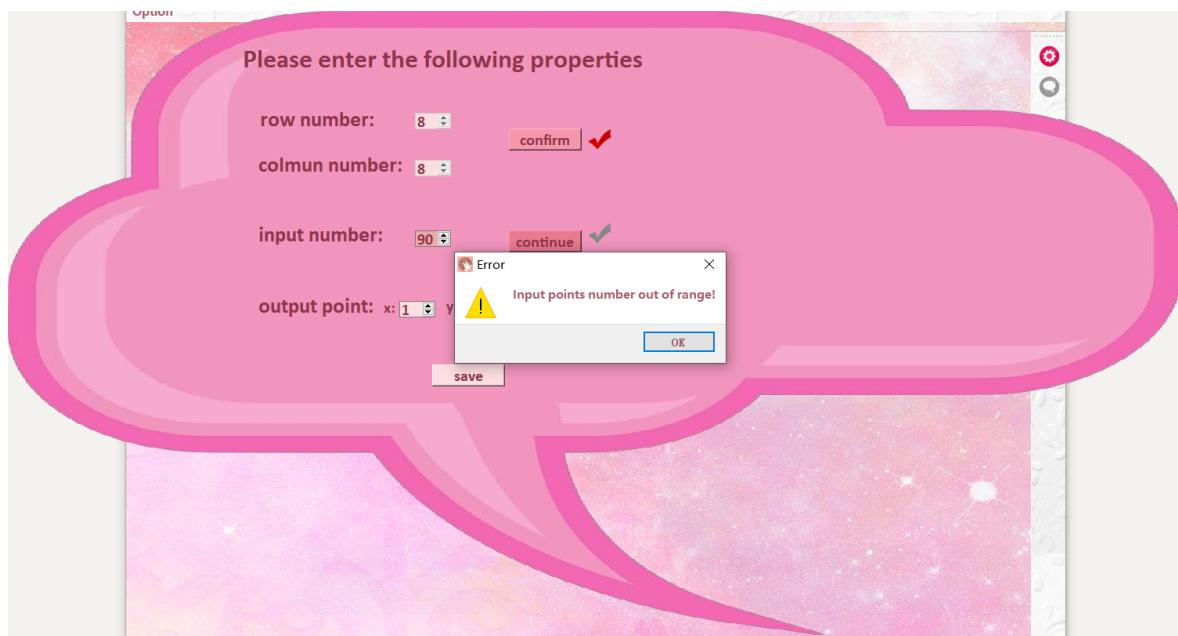


这个弹窗被我做成了不规则窗口, 使得项目的界面更加新奇好看。而且这个窗口承担了很多没有被大作业要求写入但是很重要的数据合法性检查工作。

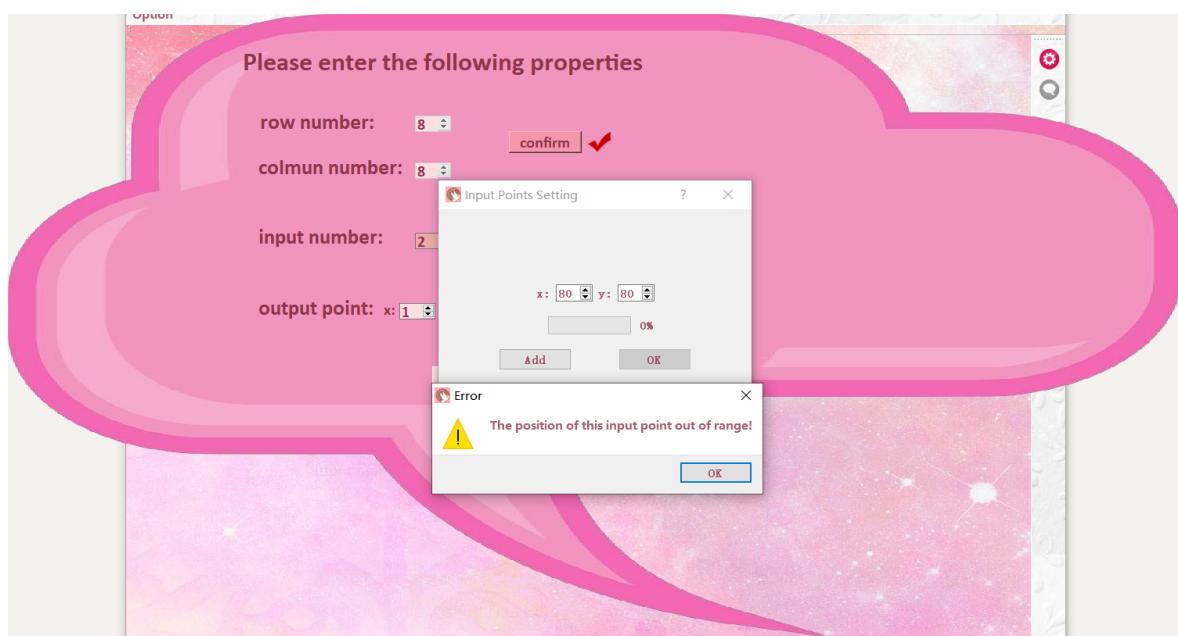
行和列数的约束:



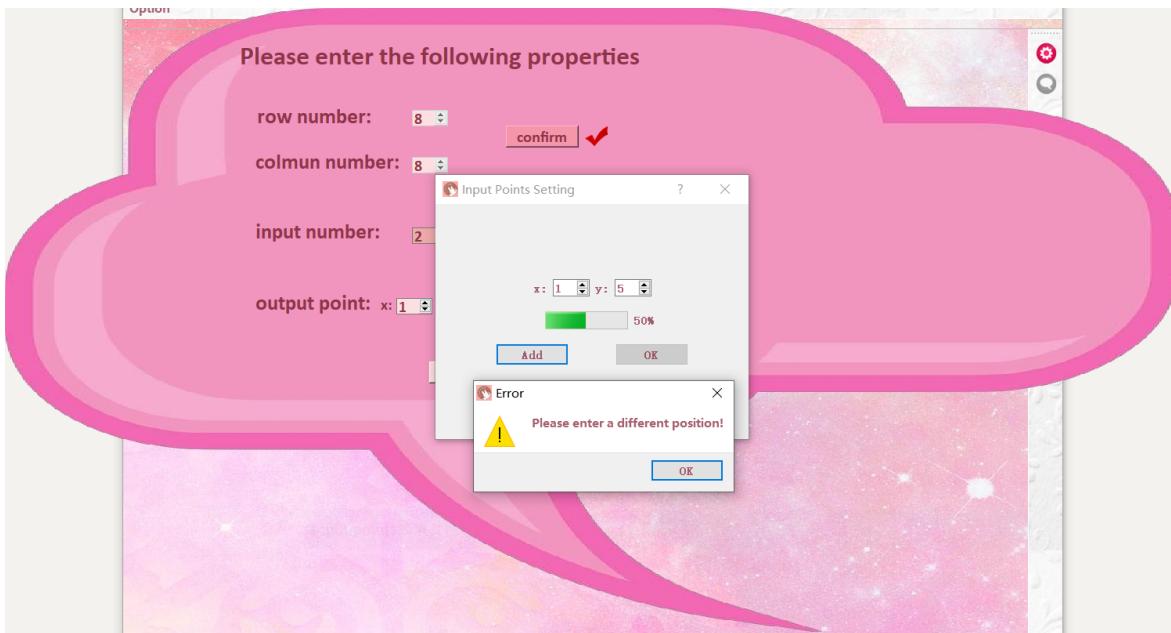
input points的数目的合法性检查:



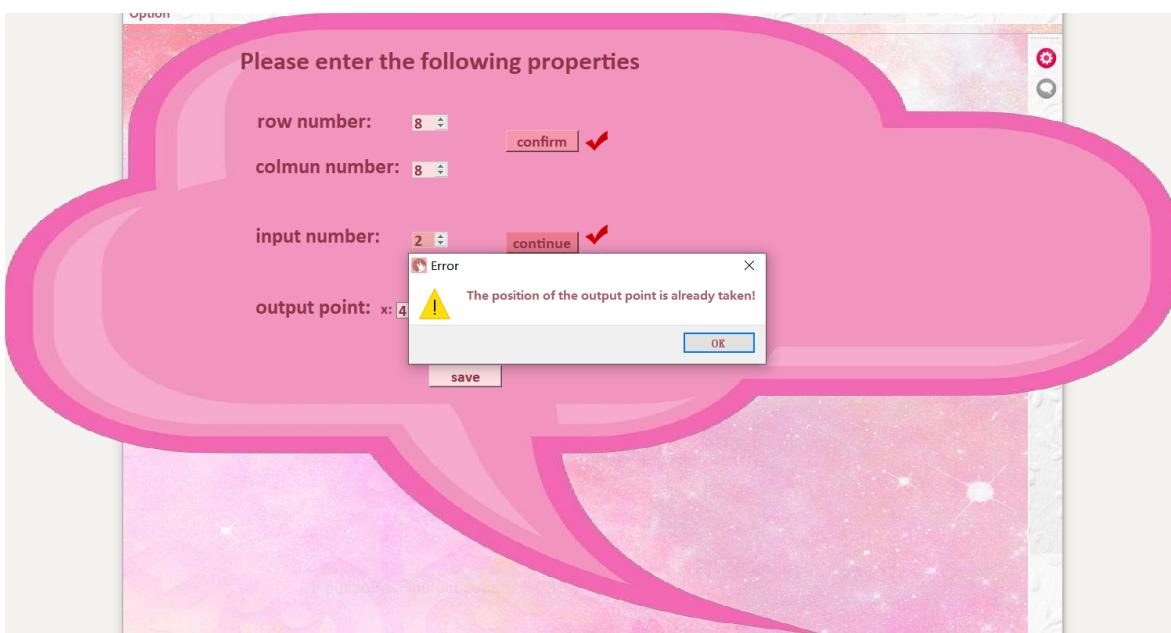
input points的位置的合法性检查:



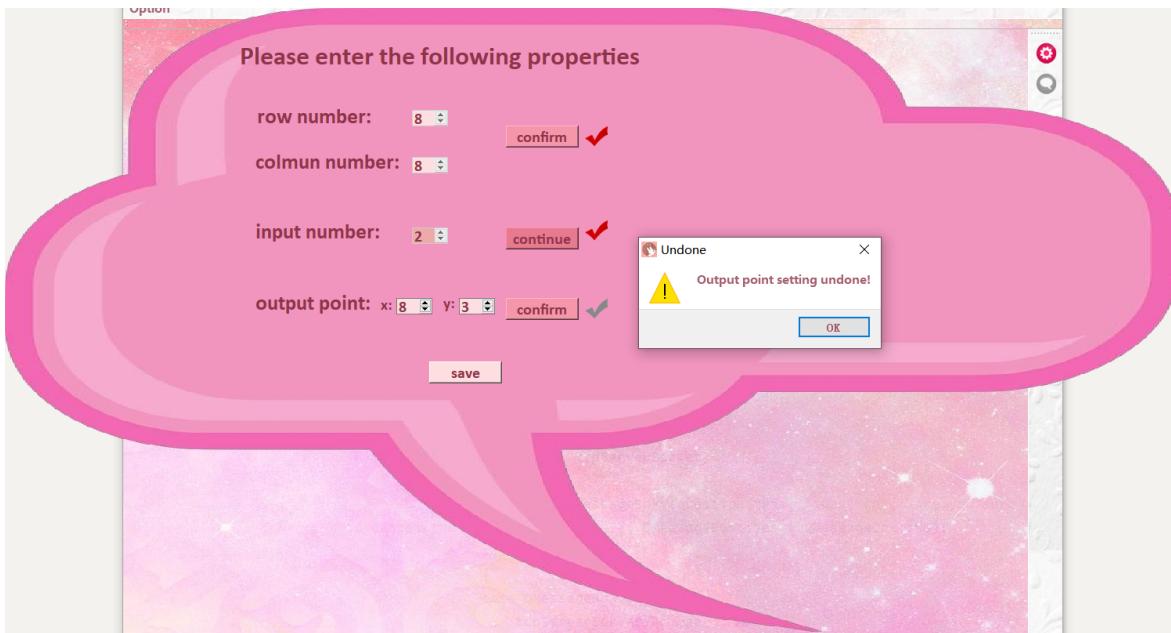
input points的重复性的检查:



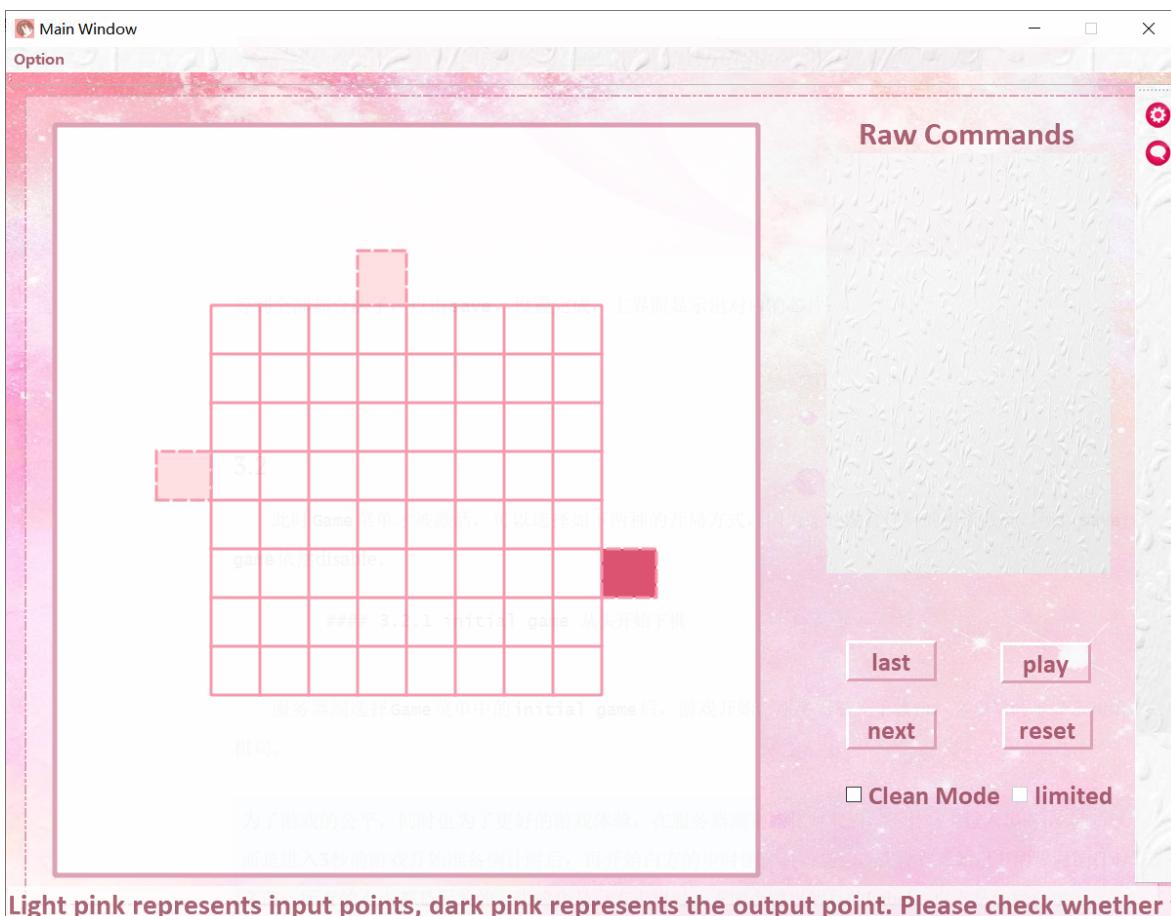
input points和output point的重复性的检查：



如果有的部分没有使得该部分之后的对勾变红，直接点击save会报错：



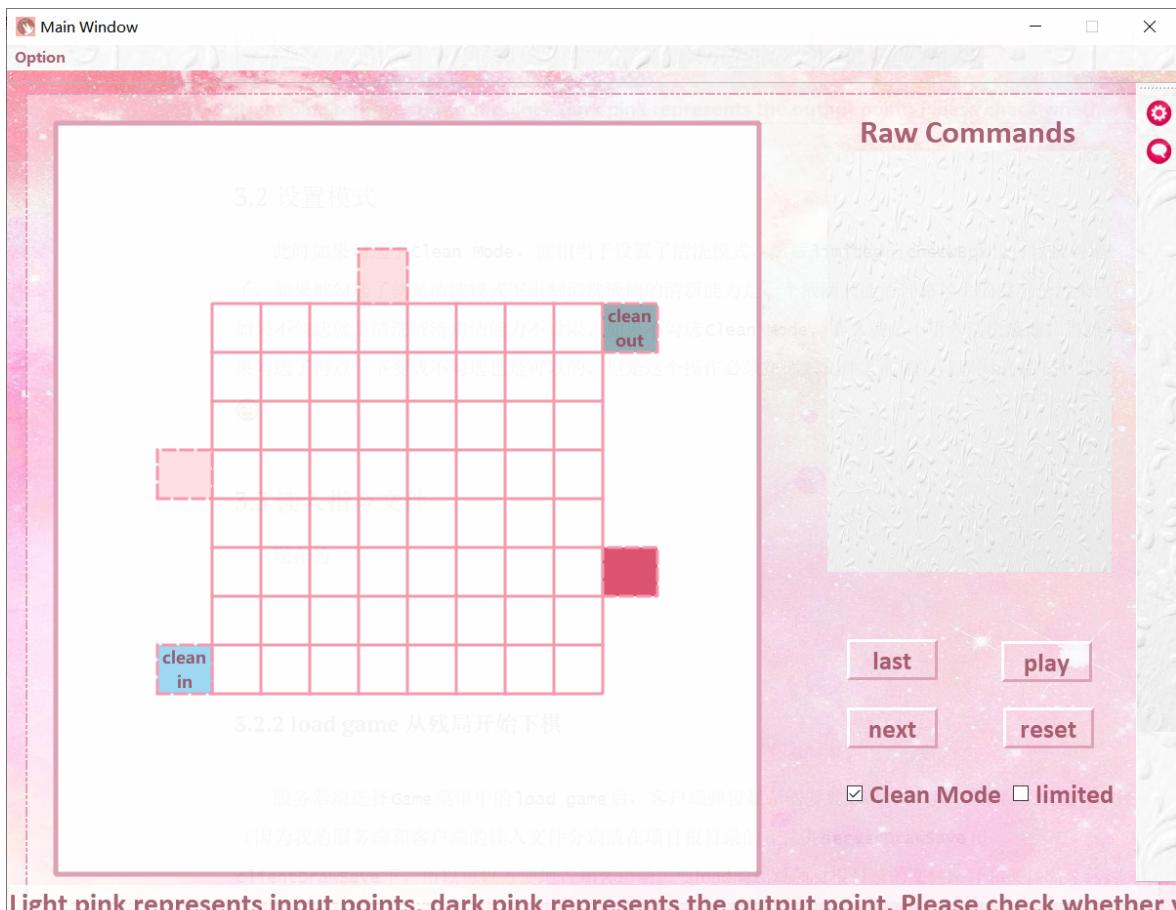
等到全部都合法了，点击 **save**，设置完成，主界面显示出对应的芯片：



### 3.2 设置模式和障碍

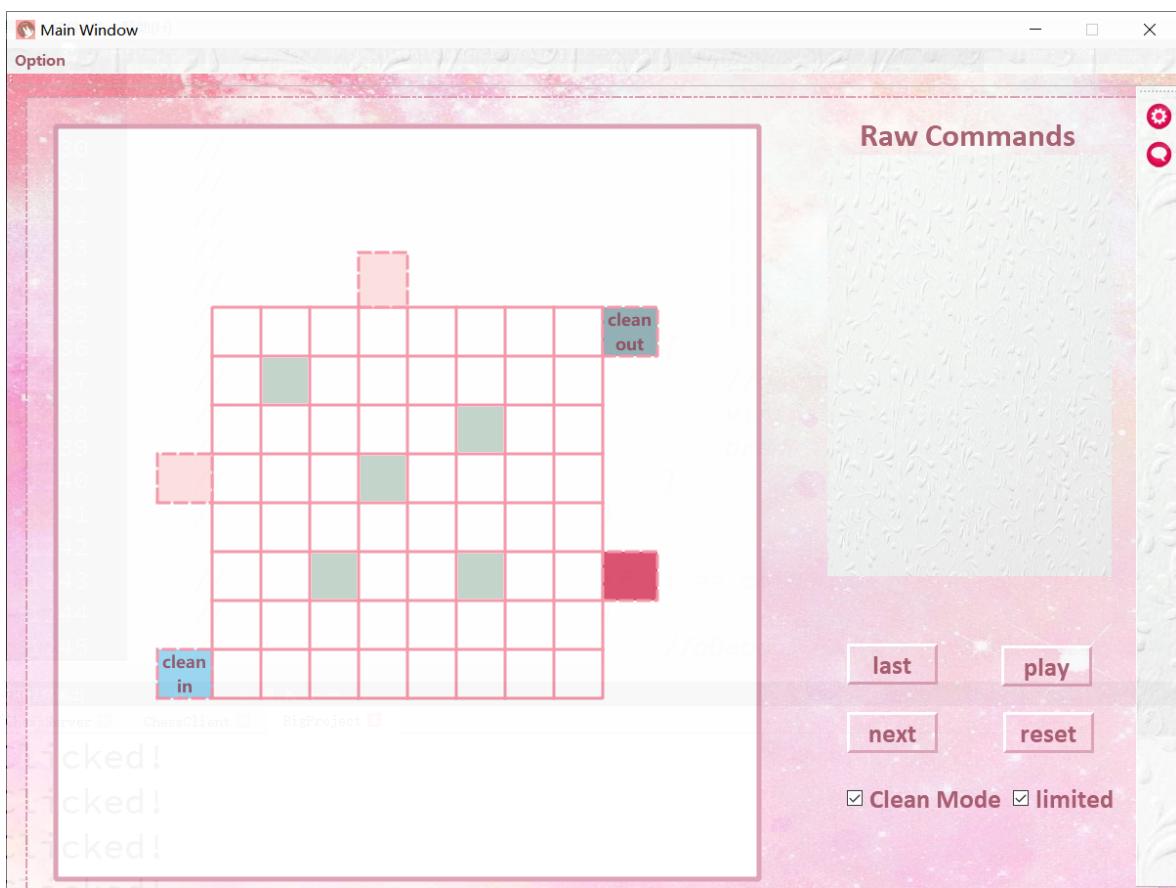
此时如果勾选了 **Clean Mode**，就相当于设置了清洗模式，然后 **limited** 的 **checkbox** 也可以被勾选了，如果被勾选了就是清洗模式下限制清洗液滴的清洁能力是一个液滴只能清理路径上的前三个污染点，如果不勾选就是清洗液滴清洁能力不设限。如果不勾选 **Clean Mode**，那么就是不引入清洗液滴。当然如果勾选了再点一下变成不勾选也是可以的。但是这个操作必须在读入文件之前做完，原因请看

如果勾选了 **Clean Mode**, 芯片绘制区会出现清洗液滴的输入输出口:



Light pink represents input points, dark pink represents the output point. Please check whether your board is valid.

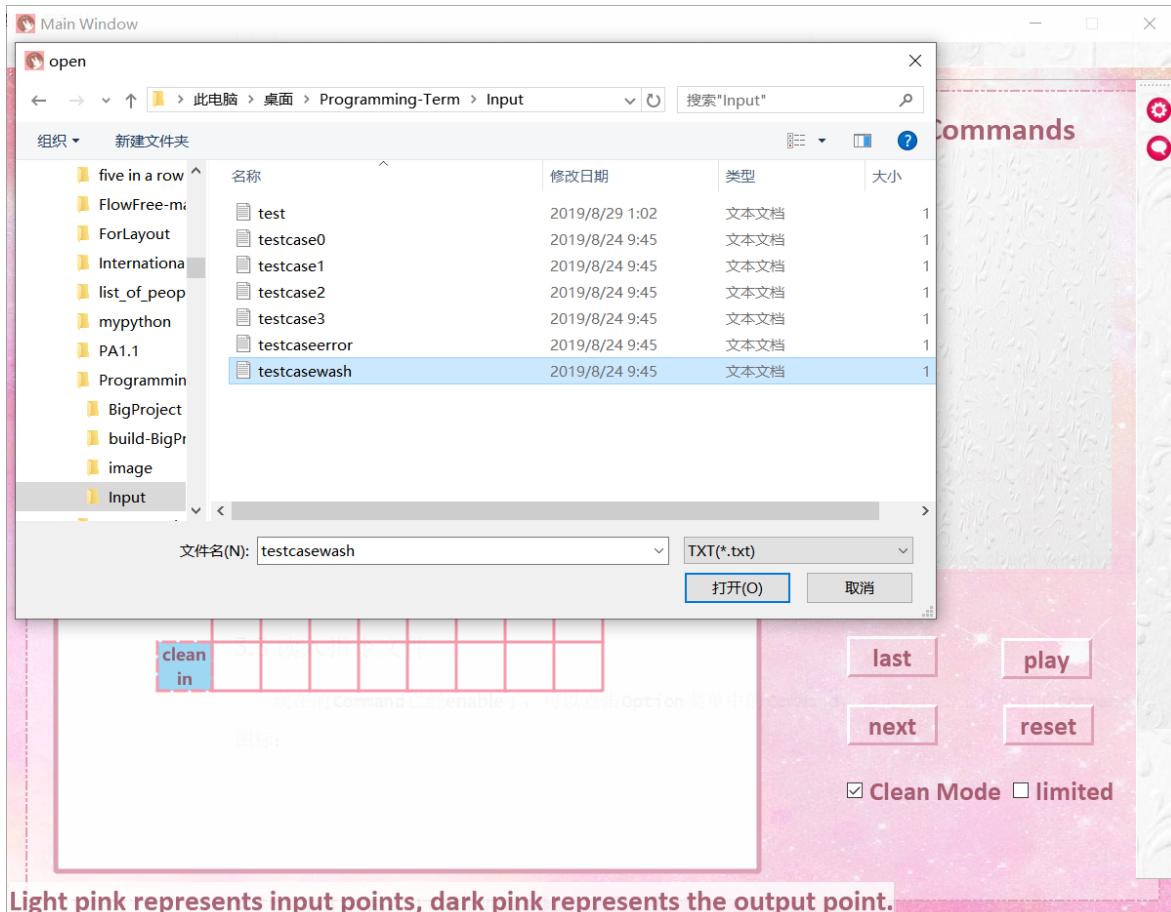
然后也可以进行清洗液滴障碍的设置:



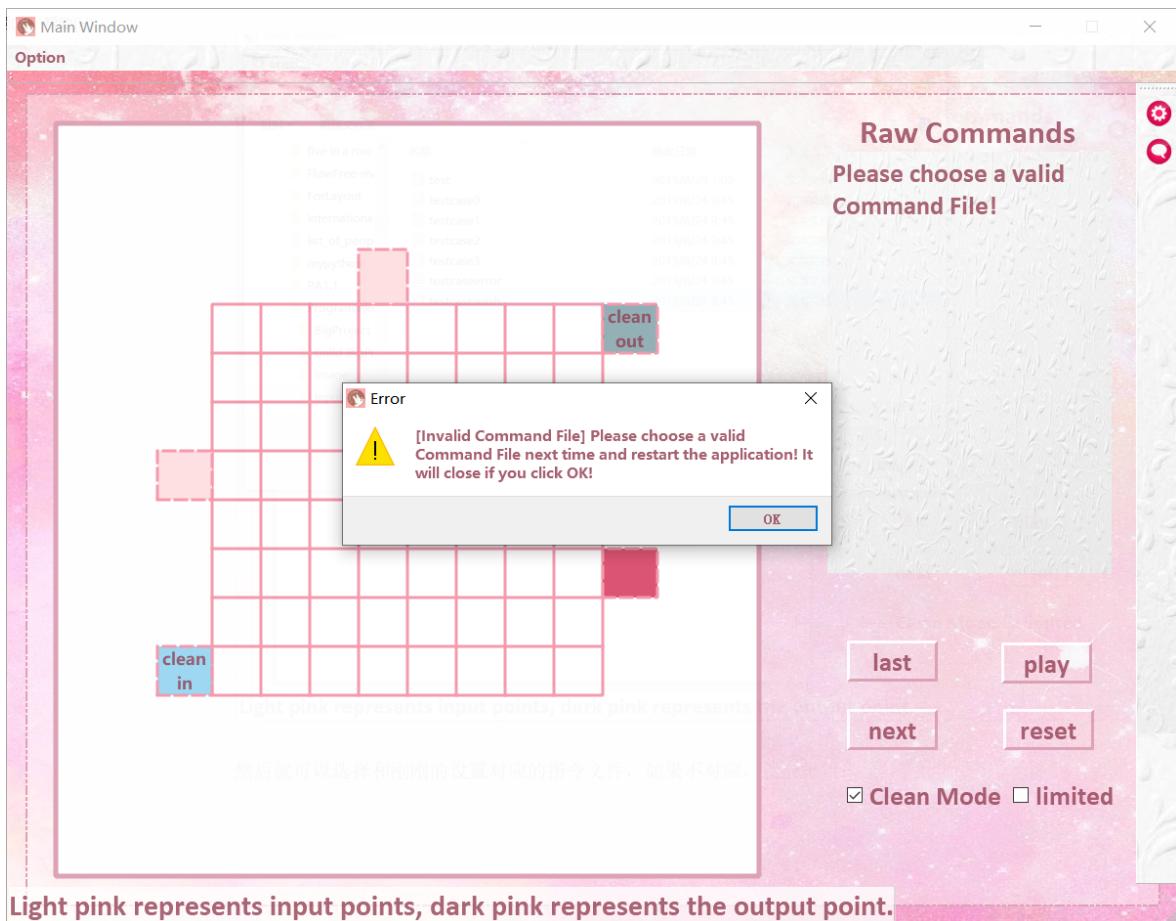
Light pink represents input points, dark pink represents the output point. Please check whether your board is valid.

### 3.3 读入指令文件

现在的 `Command` 已经 enable 了，可以点击 `option` 菜单中的 `Command`，也可以点击工具栏中的 `Command` 图标：



然后就可以选择和刚刚的设置对应的指令文件，如果不对应，就会报错：



Light pink represents input points, dark pink represents the output point.

如果直接把文件选择对话框给关掉了，没有选择任何文件，那么 `Clean Mode` 和 `limited` 还可以设置。

等到读入了正确的对应文件，`Clean Mode` 和 `limited` 就 disable 了，用来显示当前芯片图像对应的时刻的 `lcd` 就会出现了，那 4 个按钮此时也 enable 了。

现在可以解答 3.2 节中我提出的问题了，“为什么模式的设定一定要在读入正确的文件之前呢？”，因为这其实不仅仅是读文件，我的程序在读文件和显示文件之间的时间段里已经做了所有的数据操作，这么做好处不仅高效，也可以解决指令不按时间顺序导致不能读一句显示一次图像的问题，还可以解决计算量过大导致界面卡顿的问题，因为如果是播放芯片状态的时候卡顿是非常明显的，而读文件再显示到界面上，这个过程中有些许延迟也不影响用户体验。

那么我都对读入的指令做了什么处理呢？

首先我在 `Line` 类里面将每一句原始指令剖析成另一种更便于处理的指令，存进 `QStringList` 里，作为新的指令集。因为原始指令的时间顺序很乱，如果按照原来的顺序进行存储，不仅该动作的起始时间不是按照顺序来的，甚至指令内部的每个拆分开的动作也都和其它指令的时间错乱的。所以我做了如下处理：

```
Line::Line(QString line)
{
    curStep = 0;
    wholeLine = line;
    qDebug() << "wholeLine = " << wholeLine;
    line = line.replace(" ", ",");
}
```

```

line = line.replace(";", " ,");
QStringList parts = line.split(",");
qDebug()<<parts;

action = parts[0];
qDebug()<<"action = "<<action;

QPoint nowPoint;
for(int i=2;i<parts.size()-1;i+=2)
{
    if(parts[i].toInt()==0)
    {
        break;
    }
    else {
        nowPoint.setX(parts[i].toInt());
        nowPoint.setY(parts[i+1].toInt());
        path.push_back(nowPoint);
    }
}

beginTime = parts[1].toInt();
int interval = 0;
if(action == "Input")
{
    kind = 1;
    beginTime = parts[1].toInt();
    interval = 1;

    QStringList list2;
    list2<< "drawI"<<QString::number(beginTime+1)<<parts[2]<<parts[3];
    lineList.push_back(list2);
    qDebug()<<list2;
}
if(action == "Move")
{
    kind = 2;
    interval = 1;
    QStringList list1;
    list1<< "clear"<<QString::number(beginTime+1)<<parts[2]<<parts[3];
    lineList.push_back(list1);
    qDebug()<<list1;
    QStringList list2;
    list2<< "move"<<QString::number(beginTime+1)<<parts[2]<<parts[3]
    <<parts[4]<<parts[5];
    lineList.push_back(list2);
    qDebug()<<list2;
}

if(action == "Split")
{

```

```

kind = 3;
interval = 2;
QStringList list1;
list1<< "clear"<<QString::number(beginTime+1)<<parts[2]<<parts[3];
lineList.push_back(list1);
qDebug()<<list1;
QStringList list2;
if(path[2].y()==path[1].y())
{
    list2<< "drawLFS";
}
else {
    list2<<"drawLS";
}
list2<<QString::number(beginTime+1)<<parts[2]<<parts[3];
lineList.push_back(list2);
qDebug()<<list2;
QStringList list3;
list3<< "clear"<<QString::number(beginTime+2)<<parts[2]<<parts[3];
lineList.push_back(list3);
qDebug()<<list3;
QStringList list4;
list4<< "drawS"<<QString::number(beginTime+2)<<parts[4]<<parts[5]
<<parts[6]<<parts[7];
lineList.push_back(list4);
qDebug()<<list4;
}
if(action == "Merge")
{
    kind = 4;
    interval = 2;
    QStringList list1;
    list1<< "clear"<<QString::number(beginTime+1)<<parts[2]<<parts[3];
    lineList.push_back(list1);
    qDebug()<<list1;
    QStringList list2;
    list2<< "clear"<<QString::number(beginTime+1)<<parts[4]<<parts[5];
    lineList.push_back(list2);
    qDebug()<<list2;
    QStringList list3;
    QPoint middle = (path[0]+path[1])/2;
    if(path[0].y()==path[1].y())
    {
        list3<< "drawLFM";
    }
    else {
        list3<<"drawLM";
    }
    list3<<QString::number(beginTime+1)<<QString::number(middle.x())
<<QString::number(middle.y());
    lineList.push_back(list3);
    qDebug()<<list3;
}

```

```

        QStringList list4;
        list4<< "clear"<<QString::number(beginTime+2)
<<QString::number(middle.x())<<QString::number(middle.y());
        lineList.push_back(list4);
        qDebug()<<list4;
        QStringList list5;
        list5<< "drawB"<<QString::number(beginTime+2)
<<QString::number(middle.x())<<QString::number(middle.y());
        lineList.push_back(list5);
        qDebug()<<list5;

    }

    if(action == "Mix")
    {
        kind = 5;
        interval = path.size()-1;
        qDebug()<<"path.size() = "<<path.size();
        int cnt = 2;
        int t = 1;
        while(t<path.size())
        {
            //这里小心别越界了
            qDebug()<<"cnt = "<<cnt;
            if(cnt>=2*path.size()) break;
            QStringList list1;
            list1<< "clear"<<QString::number(beginTime+t)<<parts[cnt]
<<parts[cnt+1];
            lineList.push_back(list1);
            qDebug()<<list1;
            QStringList list2;
            list2<< "move"<<QString::number(beginTime+t)<<parts[cnt]
<<parts[cnt+1]<<parts[cnt+2]<<parts[cnt+3];
            lineList.push_back(list2);
            qDebug()<<list2;
            cnt+=2;
            t++;
        }
    }

    if(action == "Output")
    {
        kind = 6;
        interval = 1;
        QStringList list2;
        list2<< "clear"<<QString::number(beginTime+1)<<parts[2]<<parts[3];
        lineList.push_back(list2);
        qDebug()<<list2;
    }

    endTime = beginTime+interval;

```

```
}
```

总而言之就是把 `Input` 翻译成在原指令的起始时间的下一个时刻在指令中那个位置画一个液滴 `Move` 翻译成在起始时间让第一个位置的液滴消失，在下一个时刻在第二个位置画一个液滴，`Mix` 就可以拆成很多的 `Move` 来翻译，`split` 就是先把第一个位置的液滴清空，然后画个椭圆，然后在下个时刻在第二个位置和第三个位置画两个小水滴，`Merge` 就是把前两个位置的液滴清空，同时在它们之间的格子里画个椭圆，下一秒再把椭圆清空，画成一个更大的圆液滴，`output` 就是在那个时刻把在这个位置上的液滴清空。

分析的关键点在于一定要一个一个时刻地分析，这样才能准确存储每个时刻的芯片状态。我使用 `operation` 里的 `Qvector<matrixcomb>` 的 `status` 存储仅根据指令文件刻画出来的每个时刻的芯片矩阵状态。

```
//先把无论是否清洗模式都要做的存状态步骤做完
status[0].isDecided = true;
for (int t = 0; t <= wholeTime; t++)
{
    if (!isClean)
    {
        if (t > 0)
        {
            status[t] = status[t - 1];
            status[t].soundDefault();
        }

        for (int k = 0; k < lineTimeList[t].size(); k++)
        {
            if (lineTimeList[t][k][0] == "drawI")
            {
                int c = lineTimeList[t][k][2].toInt();
                int r = lineTimeList[t][k][3].toInt();
                qDebug() << "c = " << c;
                qDebug() << "r = " << r;

                status[t].comb[c][r].isEmpty = false;
                colorSeed++;
                QColor nowColor = QColor((colorSeed * 220) % 255,
                (colorSeed * 20) % 255, (colorSeed * 60) % 255, 200);
                qDebug() << "nowColor = " << nowColor;
                status[t].comb[c][r].dropColor = nowColor;
                status[t - 1].comb[c][r].dropColor = nowColor;
                //status[t].comb[c]
                [r].pollutedSet.push_back(nowColor);
                //然后在paintEvent里面遍历所有颜色画出污染
            }
            if (lineTimeList[t][k][0] == "move")
            {
                int c1 = lineTimeList[t][k][2].toInt();
```

```

        int r1 = lineTimeList[t][k][3].toInt();
        int c2 = lineTimeList[t][k][4].toInt();
        int r2 = lineTimeList[t][k][5].toInt();

        status[t].comb[c2][r2].isEmpty = false;
        colorSeed++;
        QColor nowColor = status[t - 1].comb[c1]
[r1].dropColor;
        qDebug() << "nowColor = " << nowColor;
        status[t].comb[c2][r2].dropColor = nowColor;
        //status[t].comb[c1]
[r1].pollutedSet.push_back(nowColor);

        status[t].isMm = true;

        Polluted pol;
        if (status[t].comb[c2][r2].pollutedSet.size() > 0 &&
nowColor != status[t].comb[c2][r2].pollutedSet[0])
{
    for (int s = t - 1; s >= 0; s--)
    {
        if (status[s].comb[c2][r2].isEmpty == false)
        {
            pol.iniPol = s;
            break;
        }
    }
}

pol.time = t;
pol.p = QPoint(c2, r2);
pollutedInfo.push_back(pol);
}

}

if (lineTimeList[t][k][0] == "drawLS")
{
    int c = lineTimeList[t][k][2].toInt();
    int r = lineTimeList[t][k][3].toInt();
    status[t].comb[c][r].isEmpty = false;
    QColor nowColor = status[t].comb[c][r].dropColor;
    qDebug() << "nowColor = " << nowColor;
    status[t].comb[c][r].dropColor = nowColor;
    status[t].comb[c][r].isLongDrop = true;
    status[t].isStr = true;
}
if (lineTimeList[t][k][0] == "drawLFS")
{
    int c = lineTimeList[t][k][2].toInt();
    int r = lineTimeList[t][k][3].toInt();
    status[t].comb[c][r].isEmpty = false;
    QColor nowColor = status[t].comb[c][r].dropColor;
    qDebug() << "nowColor = " << nowColor;
    status[t].comb[c][r].dropColor = nowColor;
}

```

```

        status[t].comb[c][r].isLongDrop = true;
        status[t].comb[c][r].isFat = true;
        status[t].isStr = true;
    }
    if (lineTimeList[t][k][0] == "drawS")
    {
        int c1 = lineTimeList[t][k][2].toInt();
        int r1 = lineTimeList[t][k][3].toInt();
        int c2 = lineTimeList[t][k][4].toInt();
        int r2 = lineTimeList[t][k][5].toInt();
        status[t].comb[c1][r1].isEmpty = false;
        status[t].comb[c2][r2].isEmpty = false;
        colorSeed++;
        QColor nowColor = QColor((colorSeed * 220) % 255,
        (colorSeed * 20) % 255, (colorSeed * 60) % 255, 200);
        qDebug() << "nowColor = " << nowColor;
        status[t].comb[c1][r1].dropColor = nowColor;
        status[t].comb[c1][r1].isSmaller = true;
        status[t].comb[c1]
[r1].pollutedSet.push_back(nowColor);
        colorSeed++;
        nowColor = QColor((colorSeed * 220) % 255,
        (colorSeed * 20) % 255, (colorSeed * 60) % 255, 200);
        status[t].comb[c2][r2].dropColor = nowColor;
        status[t].comb[c2][r2].isSmaller = true;
        status[t].isAp = true;
        //status[t].comb[c2]
[r2].pollutedSet.push_back(nowColor);

    }
    if (lineTimeList[t][k][0] == "drawLM")
    {
        int c = lineTimeList[t][k][2].toInt();
        int r = lineTimeList[t][k][3].toInt();
        status[t].comb[c][r].isEmpty = false;
        colorSeed++;
        QColor nowColor = QColor((colorSeed * 220) % 255,
        (colorSeed * 20) % 255, (colorSeed * 60) % 255, 200);
        qDebug() << "nowColor = " << nowColor;
        status[t].comb[c][r].dropColor = nowColor;
        status[t].comb[c][r].isLongDrop = true;
        //status[t].comb[c]
[r].pollutedSet.push_back(nowColor);

    }
    if (lineTimeList[t][k][0] == "drawLFM")
    {
        int c = lineTimeList[t][k][2].toInt();
        int r = lineTimeList[t][k][3].toInt();
        status[t].comb[c][r].isEmpty = false;
        colorSeed++;
    }
}

```

```

        QColor nowColor = QColor((colorSeed * 220) % 255,
        (colorSeed * 20) % 255, (colorSeed * 60) % 255, 200);
        qDebug() << "nowColor = " << nowColor;
        status[t].comb[c][r].dropColor = nowColor;
        status[t].comb[c][r].isLongDrop = true;
        status[t].comb[c][r].isFat = true;
        //status[t].comb[c]
[r].pollutedSet.push_back(nowColor);

    }

    if (lineTimeList[t][k][0] == "drawB")
    {
        int c = lineTimeList[t][k][2].toInt();
        int r = lineTimeList[t][k][3].toInt();
        status[t].comb[c][r].isEmpty = false;
        QColor nowColor = status[t - 1].comb[c]

[r].dropColor;
        qDebug() << "nowColor = " << nowColor;
        status[t].comb[c][r].dropColor = nowColor;
        status[t].comb[c][r].isBigger = true;
        qDebug() << "Draw Bigger!";

        status[t].isMer = true;
        qDebug() << "status[" << t << "].isMer = true";
    }
    if (lineTimeList[t][k][0] == "clear")
    {

        int c = lineTimeList[t][k][2].toInt();
        int r = lineTimeList[t][k][3].toInt();
        QColor nowColor = status[t].comb[c][r].dropColor;
        qDebug() << "nowColor = " << nowColor;

        status[t].comb[c][r].setDefault();
        status[t].comb[c]

[r].pollutedSet.push_back(nowColor);

    }

    //只是轨迹变化，其它并不变，等做了一次清洗操作后，如果后面的之前会
    //发生污染液滴事件的点也被清洗了，那就跳过它，再去清洗下一个
    //如果pollutedInfo已被赋值，就去寻找每一个该去clean的之前的时间
    //说不定前面的清理已经将后面的一个polluted给去除了
}
}

```

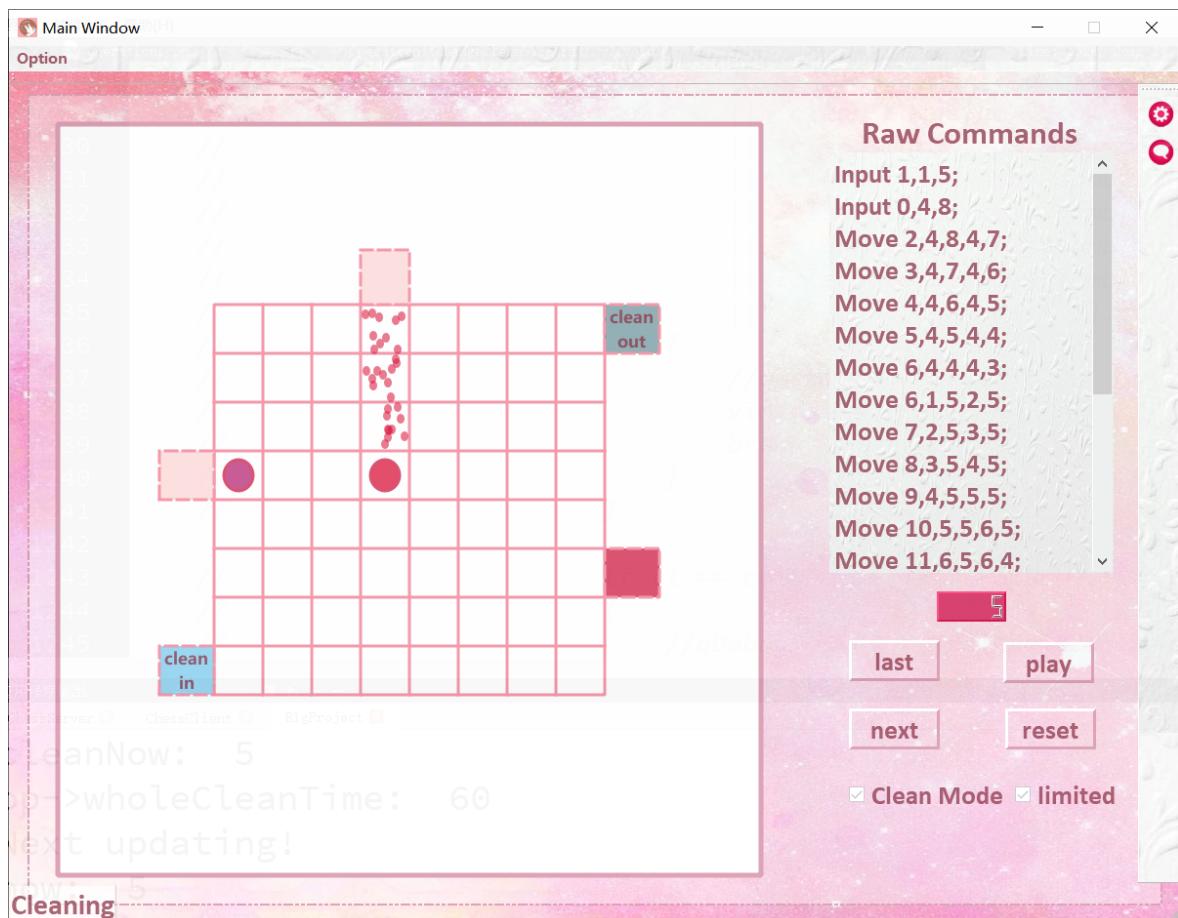
然后在普通模式下，将时刻的变化和主界面的按钮和lcd关联，`paintEvent`函数根据时刻和对应时刻的`satus[i]`来画图，就像放幻灯片那样`update`主界面就可以了。

其中新液滴的颜色绘制我使用了

```
QColor nowColor = QColor((colorSeed * 220) % 255, (colorSeed * 20) % 255,
                           (colorSeed * 60) % 255, 200)
```

这样调整 `colorSeed` 的值，能产生和我的界面颜色比较搭的颜色，因为它的最初值(220,20,255)就是粉紫色系的，而且加上了透明度，不仅不会相互遮盖，还很美观。

而每一个旧液滴就继承它之前那步的液滴颜色就可以了，至于污染的绘制，我使用了随机数，在方格范围内随机选取小圆的横纵坐标，就可以看到很形象的污染了，而且每刷新一下，随机数都不同，就像看到污染液滴在流动一样。



### 3.4 清洗模式

在清洗模式里，在时停的基础上，我先根据之前存储的 `status` 找出了第一个会出现污染的地方（因为可能清理这个污染点的时候，后面的污染点也被清掉了，所以先考虑第一个），然后往之前的时刻去回溯它处在不受其它液滴静态约束的最晚的时刻（因为时刻越晚留下的污染越多，清理液滴就越能充分清理这些污染），然后根据那个时刻的芯片状态，找出清理液滴可以行走的节点（不受其它液滴的静态约束，并且没有被设成障碍），把它们存到节点集里。

我使用了**floyd**算法，并且用如下方法把每两个可通节点的最短路径存了下来，这样就直接能得到清洗液入口到未来的第一个污染点所在位置的最短路径，和未来的第一个污染点所在位置到清理液出口的最短路径：

```
int** dist = new int*[pointNum];
for (int i = 0; i < pointNum; i++)
{
    dist[i] = new int[pointNum];
}

qDebug() << "pointNum = " << pointNum;

for (int i = 0; i < pointNum; i++)
{
    qDebug() << "points[" << i << "]" << points[i];
}

path = new int*[pointNum];
for (int i = 0; i < pointNum; i++)
{
    path[i] = new int[pointNum];
}

int middle = -1;

for (int i = 0; i < pointNum; i++)
{
    if (points[i] == mid)
    {
        middle = i;
    }
}

qDebug() << "middle: " << middle;

if (middle == -1)
{
    emit cannotCleanInCleanMode(poL.cleanTime);
    return;
}

for (int i = 0; i < pointNum; i++)
{
    for (int j = 0; j < pointNum; j++)
    {
        if (i == j)
        {
            dist[i][j] = 0;
```

```

        }
        else if (qAbs(points[i].x() - points[j].x()) + qAbs(points[i].y() -
points[j].y()) == 1)
        {
            dist[i][j] = 1;
            //qDebug() << "Origin: dist[" << i << "][" << j << "] = 1";
        }
        else {
            dist[i][j] = MAXNUM;
        }
    }

for (int i = 0; i < pointNum; i++)
{
    for (int j = 0; j < pointNum; j++)
    {

        if (dist[i][j] < MAXNUM)
        {
            path[i][j] = i;
        }
        else {
            path[i][j] = -1;
        }
    }
}

for (int k = 0; k < pointNum; k++)
{
    for (int i = 0; i < pointNum; i++)
    {
        for (int j = 0; j < pointNum; j++)
        {
            if (dist[i][k] + dist[k][j] < dist[i][j])
            {
                dist[i][j] = dist[i][k] + dist[k][j];
                path[i][j] = path[k][j];
            }
        }
    }
}

```

```

void Operation::out(int i, int j)
{

```

```

if (i == j)
{
    //qDebug("%d->", i);
    return;
}
int k;
k = path[i][j];
if (k == i)
{
    return;
}
//qDebug() << "path[" << i << "][" << j << "]: " << path[i][j];
if (k == -1)
{
    qDebug() << i << " and " << j << " can't be connected!";
    emit cannotCleanHere();
    return;
}
out(i, k);
road.push_back(k);
qDebug("%d->", k);
}

```

然后我就把新的芯片矩阵存到新的`Qvector<matrixComb> cleanstatus`里面了，然后也像之前那样关联，就像放幻灯片那样update就可以了。

如果不能清理，要么是没有路径，要么是污染点一直处于会违反静态约束的地方，就会弹窗报错说不能清洗，然后可以以非清洗模式继续往下播放。

如果之前勾选了**limited**选项，就让每个清洗液滴只能清理路径上的前3个点，如果预判时发现不能清理到目标污染点，就在放出该清洗液滴的下`i (i = 1, ....)`秒再放出足够个清洗液滴，知道清理到目标点即可。

我的这种方法在时停的时候应当是用时最少也最省清洁液滴的。