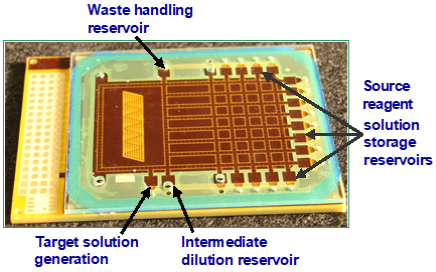
微流控生物芯片模拟程序设计文档

程序介绍

现实背景

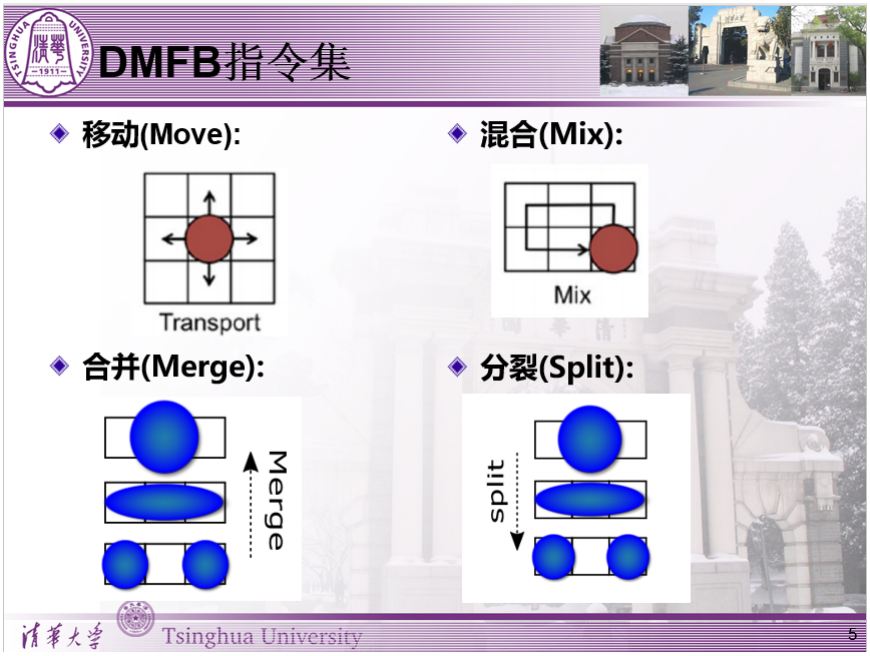
微流控生物芯片是将生化医分析过程的样品制备、反应、分离、检测等基本操作单元集成到一块微米尺度的生物芯片上，自动完成分析的全过程。



在分析的过程中，数字微流控芯片(DMFB)通过控制微电极阵列上的电压变化，操纵液滴进行移动(Move)、合并(Merge)、分裂(Split)和混合(Mix)的操作。

设计目的

利用QT的图形化界面技术，读入指令模拟DFMB上液滴的上述操作。在程序中将微流控芯片显示为俯视状态下的二维网格，其中每个独立的网格单元均为电极。四种基本操作的示意图如下



模拟过程的时序是以单位时间离散进行的。在0时刻全场为空，在t时刻执行的指令在t时刻局面的基础上进行，即效果的显示至少要到下一时刻。具体来说，对于t时刻的指令，状态转移为：

Move：t+1时刻完成移动

Merge：t+1时刻成为椭圆，t+2时刻合并完成

Split：t+1时刻成为椭圆，t+2时刻合并完成

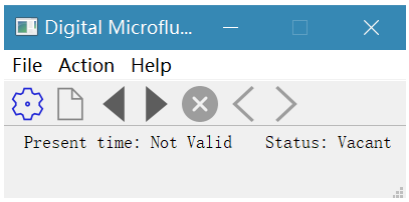
不同的液滴的种类不同，同时认为合并和分裂过程产生的都是完全不同种类的液滴，即Merge：1+2->3，Split：1->2+3。液滴的种类与污染挂钩。

液滴移动过程还必须满足的条件是静态约束和动态约束。静态约束是说，除了要合并的两个液滴，任意时刻任意两个液滴间的曼哈顿距离必须大于2，否则会导致液滴意外污染（较严重）；同时由于移动过程中速度的不确定性，前一时刻和后一时刻的任意两个不同液滴间的曼哈顿距离也必须大于2（合并和分裂过程例外）。

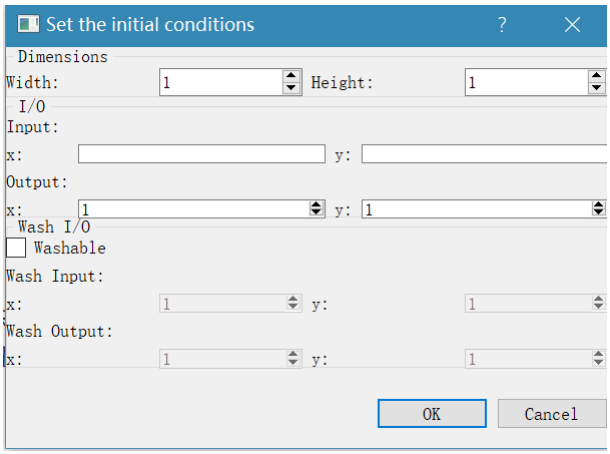
设计的基本目标为：基于给定的液滴规划数据，模拟数字微流控芯片上液滴的输入、输出、移动、混合、合并、分离的过程，设计一个基于QT的DFMB的用户界面系统。

程序使用方法

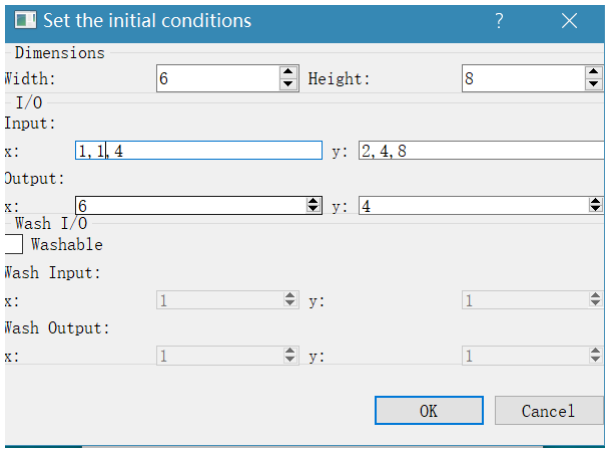
打开程序显示下列界面



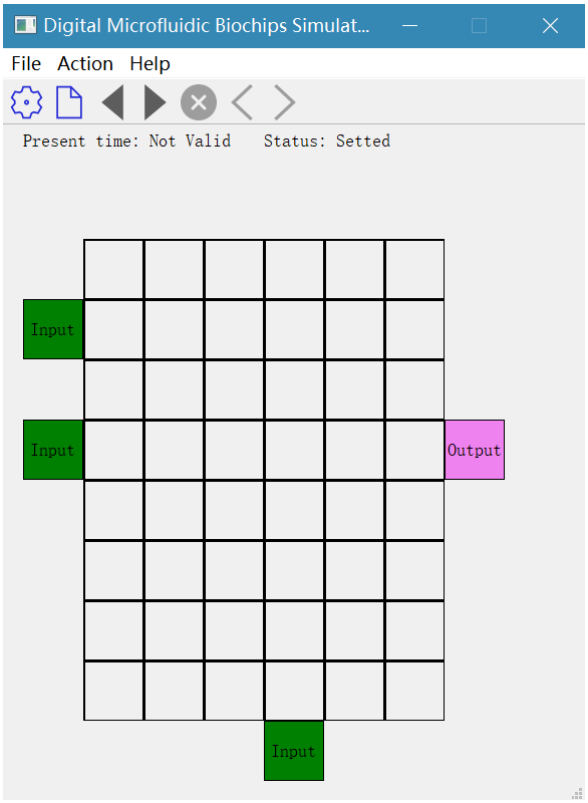
当前时刻显示不可用，状态显示空闲。此时只能进行的操作是进行初始化（查看About除外，下同）。点击菜单栏或者工具栏中的初始化按钮，弹出如下窗口



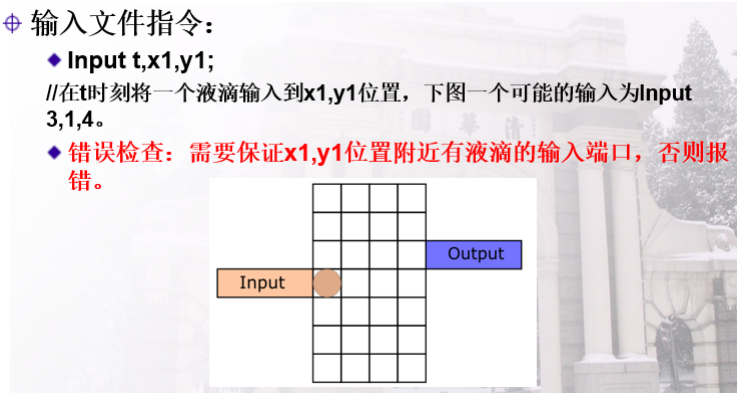
第一行设置网格的宽和高（不能同时小于等于3，不大于12），第二行设置输入口的横纵坐标（可以有多个，用逗号隔开，对应），第三行设置输出口的横纵坐标（只能有一个），第四行选择是否加入清洗功能（具体功用继续看下述说明），若加入则还需填写第五六行的清洗液滴的输入口和输出口（均只能有一个）。所有输入口和输出口都必须在网格的边界上。暂时不开启清洗功能，如下设置

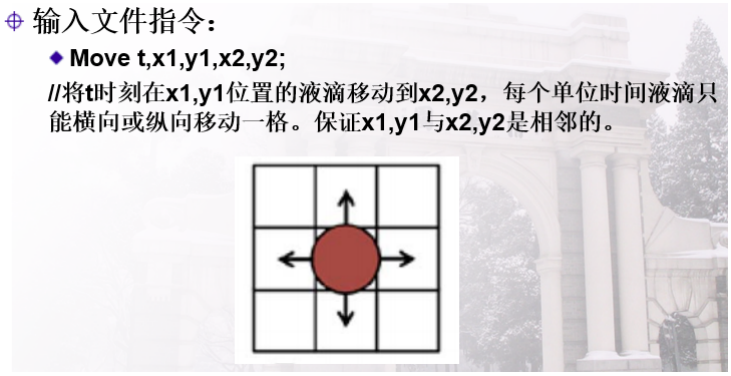


点击OK按钮后效果如下：



可以看到基本网格和输入输出口已经绘制完毕，同时状态变为已设置。在这时还可以重新设置，而模拟则需要导入文本文件。文本文件由多行自然语言化的固定格式的文本形式的指令组成（不要求按照时序），如下



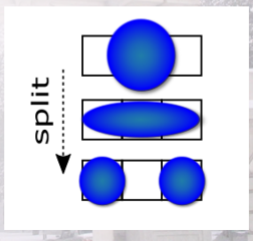


输入文件指令：

- **Split t,x1,y1,x2,y2,x3,y3;**

//将t时刻在x1,y1位置的液滴分成x2,y2、x3,y3两个液滴，分离方式如下图，输入保证x2,y2、x1,y1、x3,y3是三个相邻液滴（水平相邻或垂直相邻）。

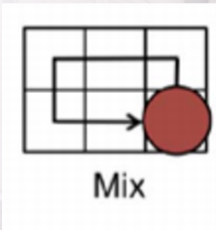
- 一次Split操作消耗两个单位时间。



输入文件指令：

- **Mix t,x1,y1,x2,y2,...,xn,yn;**

//将t时刻在x1,y1位置的液滴，沿着x1,y1,x2,y2,...,xn,yn的路线进行混合操作，到达xn,yn时混合完成，耗时为n-1个单位时间，保证每次移动只移动一格。下图对应的一个可能的输入为Mix 6,3,1,3,2,2,2,1,2,1,1,2,1,3,1;

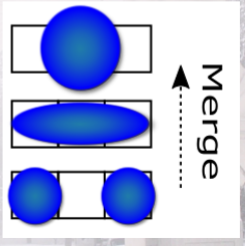


输入文件指令：

- **Merge t,x1,y1,x2,y2;**

//将t时刻在x1,y1位置和x2,y2位置的两个液滴液滴合并到两个坐标的中间位置。

- 一次Merge消耗2个单位时间。

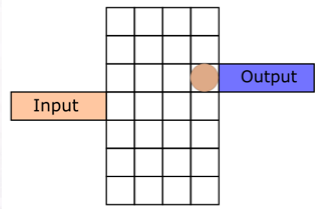


输入文件指令：

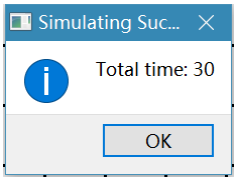
- **Output t,x1,y1;**

//在t时刻将一个液滴从x1,y1位置移出芯片，下图一个可能的输入为Output 3,4,5。

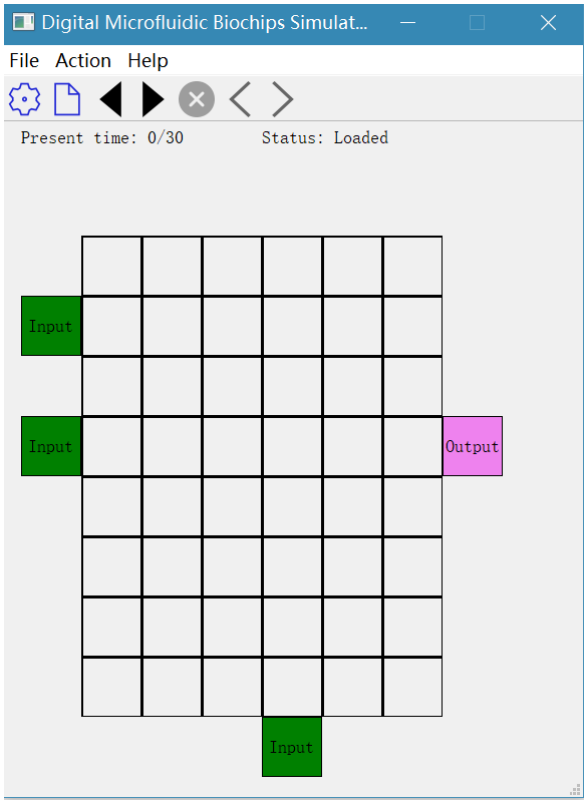
- 错误检查：需要保证x1,y1位置附近有液滴的输出口，否则报错。



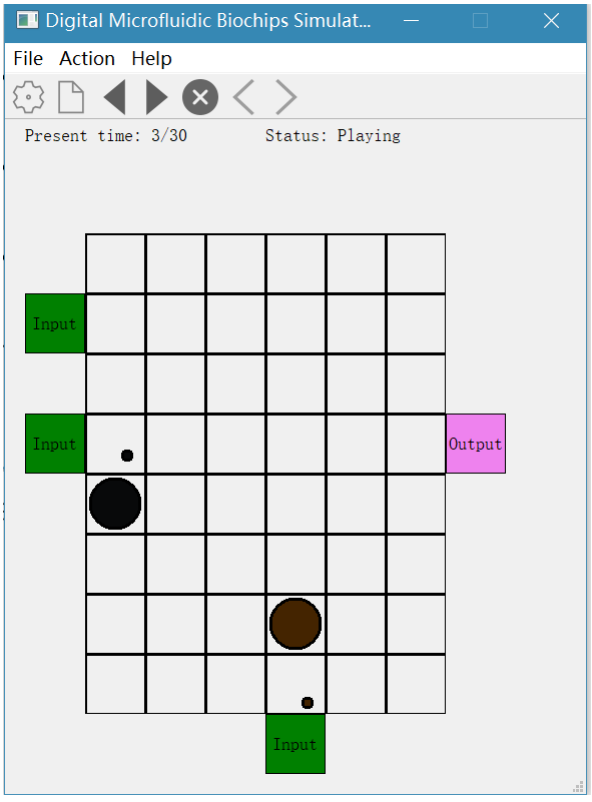
我们载入testcase2.txt，弹出如下窗口，说明文件指令已经解析完毕，没有错误，且总时间为30



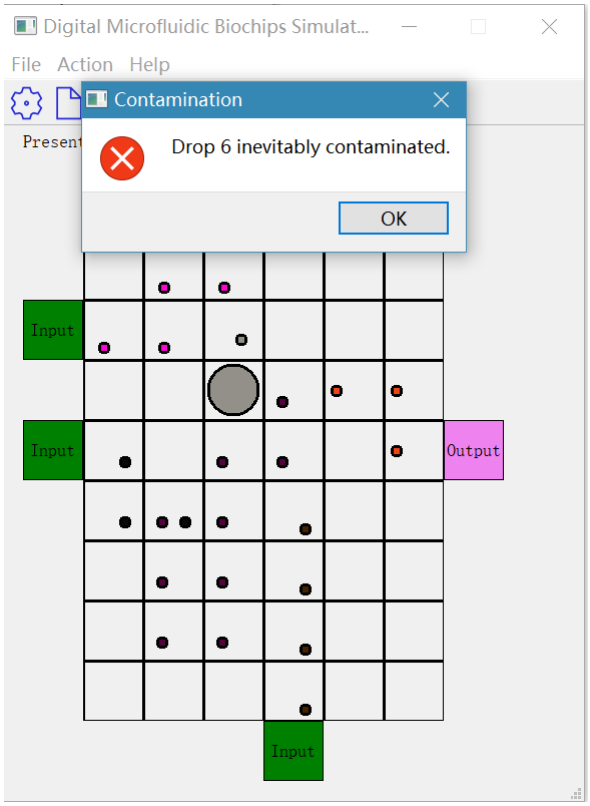
此时页面变为



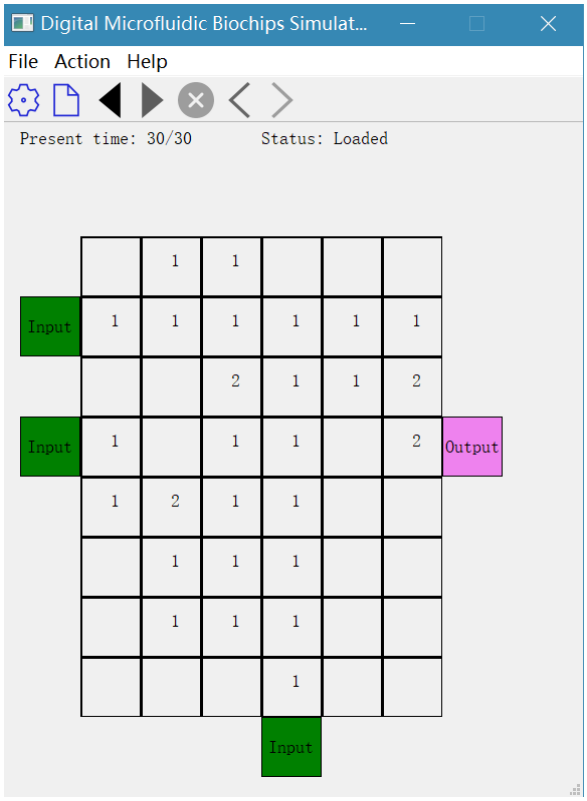
上方标签显示了当前时刻，状态显示为已加载。此时可以点击下一步（由于停留在第一步不能点击第一步，但向后播放后就可点击）、播放、重置。播放状态下状态变为正在播放



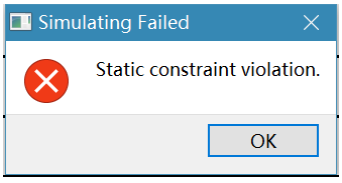
此时可以点击停止按钮来终止播放，而其余按钮均变为灰色无法点击。液滴的不同动作附带有不通音效。不同液滴的颜色是随机的不同色，可以看到液滴走过的格子会留下同颜色的污点，同一种污点在格子内的位置相同，而不同种的污点位置随机生成。当液滴走上留下有其它液滴污点的格子时，会弹出对话框提示污染，如

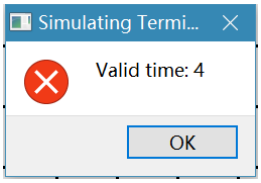


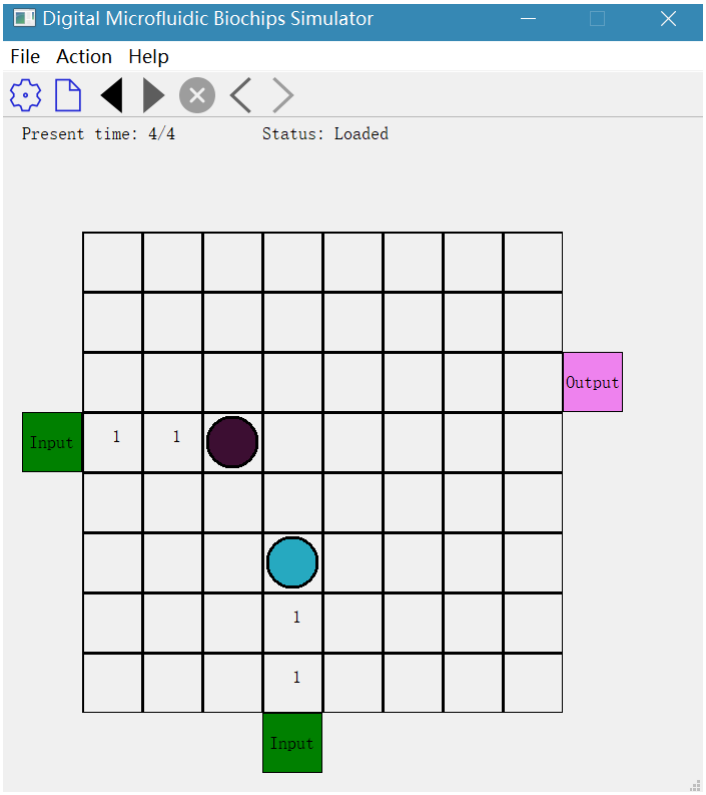
在Loaded状态下仍可进行重新初始化操作（此时所有文件载入状态重置），重新载入文件指令操作（保留原初始配置的基础上），且不会出现bug。到最后一个时刻（正常状态下场上液滴清空，均已输出），每个格子（电极）上显示被污染次数。



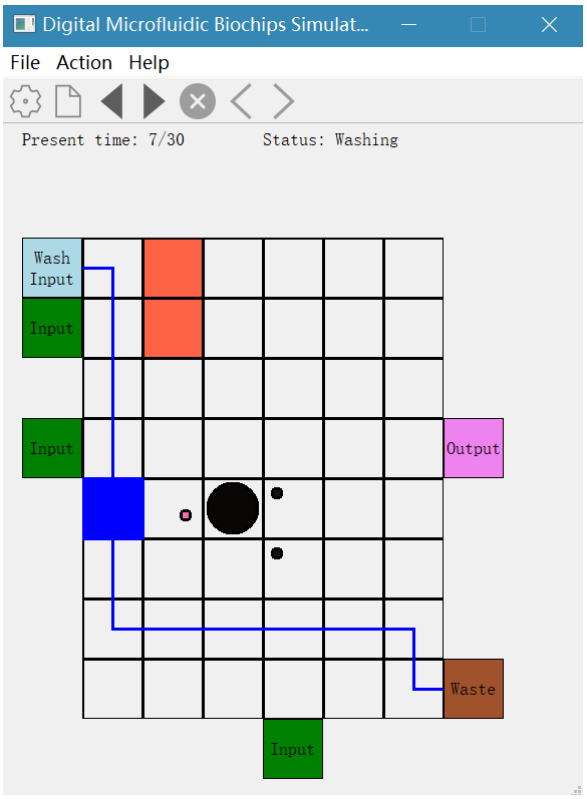
而在载入的文件出现问题（输入输出位置不再输入输出口，宽高不符，违背静态约束或动态约束）的情况下，加载完成的瞬间会弹窗报错，并保留了出现无可救药错误之前的所有时刻（有效时间）。可使用testcaseerror.txt测例自行测试，效果为







而开启清洗功能，则会实现清洗液滴的路径自动规划，**尽量**防止液滴被污染。程序采取的策略是，每个时刻后进行时停（在锁定时刻使其它液滴静止），并是清洗液滴在满足约束条件的情况下清洗尽可能多数量的污迹。而在清洗开启时，可右键点击某个电极，此时电极会变成红色，标明了清洗液滴无法经过此电极。在红色电极上再次点击则可以取消对于此电极的锁定。在清洗过程中会显示清洗液滴的经过路径，并以50毫秒每步的速度演示清洗液滴从清洗入口到废液出口的移动过程。下图为效果演示



程序未实现附加功能，对其分析见程序设计部分的说明。

程序设计

技术基础

本程序在Qt Creator中编写测试，共约1300行代码，使用自带编译器(mingW)编译。

使用了Qt自带的`QString`文本库，`QFile`文件库，`QMap`、`QSet`、`QVector`、`QList`等容器库存储数据，`QSound`播放wav音效，`QTimer`进行延时，`QPaintEvent`、`QMouseEvent`处理鼠标和绘图事件，`QMessageBox`、`QFileDialog`标准消息框和文件对话框，`QPainter`绘图方法。

类的职责

utils.h中包含了定义在`namespace utils`中通用的模板方法

```
namespace utils{
    template <typename T>
    T max(T a,T b)//取最大值
    {
      return a>b?a:b;
    }
    template <typename T>
    T abs(T a,T b)//取绝对值
    {
      T c=a-b;
      return c>0?c:-c;
    }
}
```

结构体

```
struct Pos{//坐标二元组
    int x,y;
    bool operator == (const Pos & value) const{
        return (value.x==x&&value.y==y);
    }
    bool operator < (const Pos & value) const{
        if(x<value.x)return true;
        if(x>value.x)return false;
        if(y<value.y)return true;
        if(y>value.y)return false;
        return false;
    }
};

struct Setting{//初始设置
    int w,h;//宽高
    QVector<Pos> in;//输入口
    Pos out;//输出口
    bool washable;//是否开启清洗功能
    Pos w_in,w_out;//清洗液滴输入/输出口
};
struct DropStatus{//液滴状态
    Pos pos;//当前位置
    bool elliptical;//是否是椭圆
    bool vertical;//椭圆走向
};
```

```
struct Scene{//某时刻的场上液滴、污染状态
    QMap<Drop*,DropStatus> dropstatus;//液滴指针到状态的映射
    QMap<Pos,QSet<int>> newcontaminations;//每时刻相对上一时刻新增的污染
};

struct Movement{//液滴动作
    int type; //0~6 corresponding to
Input/Output/Move/Split1/Split2/Merge1/Merge2
    Pos pos;//起点
    Pos destination;//终点
    bool upward;//走向
};
```

此外还有一个全局的内联函数`uint qHash(const Pos)`，作用之后说明。

drop.h是`class Drop`的头文件。`Drop`是一个很简单的类，同一种液滴对应唯一的编号和颜色（在读入文件的解析过程中构建完毕，不更改不覆盖只丰富）。

initiatedialog.h中的`InitiateDialog`继承自`QDialog`，是进行初始化设置的对话框，调用其`Setting getSetting()`函数将执行`exec()`并阻塞，窗口销毁后返回设置（不合法的数据会之间弹出窗口报错）。

lattice.h是重要的绘图组件，全权负责电极、液滴、输入输出口等所有的图形绘制，使用最基本的`QPainter`方法。内置三种状态

```
int status=0;//0/1/2 Empty/Scene/WashScene
```

`paintEvent`内重载绘图时会根据此变量决定绘制的部分。其构造函数需传入`Setting`和两个`int`参数（标明了清洗液滴的输入输出口相对网格的上/下/左/右）。此类提供了公有函数

```
void load(int vtime,QVector<Scene> ss,QMap<int,Drop*> ds);//负责在文件解析完毕后加载到Lattice中、状态变为1
void setScene(int t);//设置显示的时刻
void wash();//短暂阻塞、显示清洗液滴移动的路径和动画、状态变为2
void getContaminations(int t);//使Lattice动态解析某个时刻的污染物分布（由于清洗液滴的引入导致污染物分布不与时刻一一对应，处理方法之后详述）
void refresh();//清洗液滴移动后进行刷新显示、恢复状态1时的画面
```

mainwindow.h放置主窗口，统筹各类之间的交互。其含有下列槽

```
public slots:
    void Initiate();
    void Load();
    void nextstep();
    void laststep();
    void reset();
    void play();
    void stop();
```

这些槽均与`QAction`的信号关联，点击菜单栏或工具栏中对应按钮会调用这些槽。MainWindow中内置状态`int status;//0~3 corresponding to Vacant/Setted/Loaded/Playing/Washing`。MainWindow中还有一些私有方法

```
void jumpto(int i);//跳转到某一时刻（联络Lattice）并更新上方标签显示时间，兼具微调QAction的功用（如在0时刻是上一步Action变为灰色）
void setStatus(int id);//设置状态，主要调整界面中QAction的disabled情况
Pos getWidgePos(Pos pos);//由坐标得到相对窗口的物理坐标
Pos transferPos(Pos pos);//调整输入输出口的附着（网格的上下左右）
void transferOrdertoMovements(QString order);//将一行指令解析并归并入保存所有Movement的数组中
bool processSimulation();//核心算法，逻辑模拟所有移动过程，判断有效性并形成模拟后的Drop*组、每一时刻的Scene
//下列是三个液滴相对位置的检测函数，用于检测静态约束、动态约束等
bool isadjacent(Pos pos1,Pos pos2);
bool farenough(Pos pos1,Pos pos2);
bool intheborder(Pos pos);
```

类的交互

类的组合关系十分明显，`Lattice`是`MainWindow`的组件，大部分逻辑处理均在`MainWindow`中完成。

下面从用户操作的**正常**流程（这里说到正常是立足于用户检验软件模拟效果而并非检验逻辑bug的角度，如用户并不会在加载文件后重新初始化。实际上本程序在逻辑的自恰性上同样下了不少功夫，除非用户使用强行修改内存等方法破坏逻辑的流畅性，许多细节的处理是严谨的！笔者也在这方面花了很多时间和精力）来解析类之间的交互和配合过程。

首先用户点击初始化，触发信号调用槽`void Initiate();`。这个槽内部新建`InitiateDialog`并阻塞，得到其返回的`Setting`。若用户直接关闭了`InitiateDialog`则维持原来状态，否则根据得到的合法配置构建`Lattice`、调整各个`QAction`的disabled状态、调整窗口大小以适应`Lattice`的大小（可对照窗口初始大小和初始化后的大小）、改变status和标签显示。

之后用户点击加载文件，弹出`QFileDialog`标准文件选择框。若用户关闭对话框则什么也不做（这样的处理之后不再叙述），若用户选择了文件则尝试以ReadOnly模式打开，若打开失败则弹出`QMessageBox::critical`标准框报错（这样的细节也不再叙述），否则使用`transferOrdertoMovements(QString)`解析每一行，全部解析后使用`processSimulation()`解析出每一时刻的状态，保留尽可能多的时刻（validetime）。这时已构建了`QVector<Scene> scenes`和`QMap<int,Drop*> drops`，利用Qt的随机数生成指定每个液滴及其污迹的颜色。之后调用`Lattice::load`装备lattice，lattice装备过程改变自身的私有变量，且生成每个液滴污迹在格子内的位置（同样使用随机数生成）。使用`jumpto(0)`调到0时刻，以及`setStatus(2)`改变`MainWindow`状态，这时装载完美结束。

装载完毕后重置、上一步、下一步、播放按钮处于可点击状态。在不开启清洗时，重置、上一步、下一步是改变`MainWindow`内置记录当前时刻的私有变量`time`并简单的调用`jumpto(time)`，播放则是使用while循环（没有到达最后时刻且没有停止），不断调用下一步并在代码块内部阻塞延时；播放状态下停止按钮处于可点击的状态，而其余按钮均变灰。

在开启清洗功能时，在上文已经提到本程序采用的策略是每步均尽最大可能清洗。开启清洗功能且不处于Washing状态时，lattice接收鼠标的右键点击事件并在私有变量`QSet<Pos>`中进行增减操作（清洗液滴移动的路径规划时会有考虑，同时`paintEvent`会对锁定的格子涂红）。此时改变的是`MainWindow::jumpto(int)`的内在逻辑：从简单的`lattice->getContaminations()`——`lattice->setScene`到`lattice->getContaminations()`——`lattice->setScene`——延时300毫秒——`lattice->wash()`——`lattice->refresh()`，途中同时改变`MainWindow`自身的状态及标签显示。这里的一个独特的亮点是，在开启清洗过程的状态下可以点击上一步和重置，细节的设定在下一部分详述。

关键部分处理

- 信号槽连接风格

- Qt中的信号槽连接有两种风格
  - Qt4风格

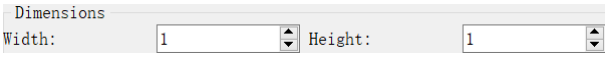
```
QObject::connect(action_next,SIGNAL(triggered()),this,SLOT(next
step()));
```

    - 槽函数必须有slots关键字
    - SIGNAL SLOT 将函数转为字符串，不进行错误检查
  - Qt5风格

```
QObject::connect(action_next, &QAction::triggered,this,
&MainWindow::nextstep);
```

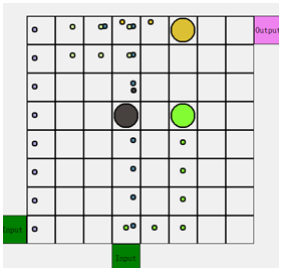
    - 槽可以是任意的成员函数，普通全局函数和静态函数
    - SIGNAL SLOT 会进行错误检查
    - 在一些功用较简易的连接中配合**lambda表达式**

```
QObject::connect(action_about, &QAction::triggered,this,[=]
(){
        QMessageBox::about(this,"About DFMB Simulator","A
rough QT product for course design.");
```

- 输入格式的控制。这是程序严谨性的体现，在输入阶段限制用户输入的随意性。
  - `SpinBox`数值输入框
    - 

    - 预期输入1-12的整数
    - 可使用`ui->spinBox->setRange(1,12);`语句/在QT Designer中修改设置来控制输入范围
  - `LineEdit`文本输入框
    - 预期输入半角逗号隔开的1-12的整数，如2,4,8
    - 使用正则表达式控制输入格式

```
QRegExp regExp("^(([1-9]|1[0-2]){1},){0,}([1-9]|1[0-2]){1}$");
QRegExpValidator *pRegExpValidator = new
QRegExpValidator(regExp,this);
ui->lineEdit->setValidator(pRegExpValidator);
```

- 随机数的生成
  - 
  - 液滴颜色

```
qsrand(QTime(0,0,0).msecsTo(QTime::currentTime()));//使用当前时刻进行随机数种子初始化
    foreach(Drop *drop,drops){
        drop->color=QColor::fromHsl(qrand()%360,qrand()%256,qrand()%200);//取模控制整数范围。这里特别的一点是，使用Hsl色彩模式（色相(0-360)、饱和度(0-255)、亮度(0-255）生成颜色，且控制亮度范围小于200，可以防止生成过于亮眼/浅的颜色（如浅黄色），达到较好的视觉效果。
    }
```

- 液滴留下污迹的在格子内部的位置

```
foreach(int i,drops.keys()){
        Pos random;
        random.x=rand()%40+10;//格子为60*60，模40以防止污迹位置过于靠近格子边缘
        random.y=rand()%40+10;
        pollutionpos.insert(i,random);
}
```

- 代码内部阻塞延时
  - 对于多线程编程，在线程内部可使用`sleep(int mesc)`来阻塞休眠。
  - 对于任意代码块，可使用`QTimer`进行计时后定时调用函数/发出信号；要达到阻塞延时的效果，可以使用`QTime`或`QElapsedTimer`

```
while(time<validtime&&status==3){
    QTime t;
    t.start();
    while(t.elapsed()<300)//未达到既定时间便阻塞循环
        QCoreApplication::processEvents();//处理事件（如代码块之外的鼠标、键盘事件），防止程序未响应卡死
    nextstep();
}
```

- 结构体作容器的键值
  - 定义了结构体`Pos`存储坐标二元组

```
struct Pos{
    int x,y;
};
```

  - 作`QMap`的键值，`QMap<Pos,XX>`
    - `QMap`同STL中`Map`类似，key作索引，需具有全序关系
    - 合理重载<运算符

```
bool operator < (const Pos & value) const{
    if(x<value.x)return true;
    if(x>value.x)return false;
    if(y<value.y)return true;
    if(y>value.y)return false;
    return false;
}
```

- 作`QSet`的特化类型，`QSet<Pos>`，如记录被锁定清洗液滴无法经过的格子
  - `QSet`与STL的`set`有**本质区别**，后者是红黑树的变种，只需要具有全序关系；后者则基于哈希表，这就要求自定义的结构体必须提供

    ```
    bool operator == (const Type &b) const;
    ```

    - 一个全局的`uint qHash(Type key);`函数

      ```
      inline uint qHash(const Pos key){
          return key.x + key.y;
      }
      ```

- 文件读入与文本分割处理
  - 文本文件的按行读入——`QTextStream`

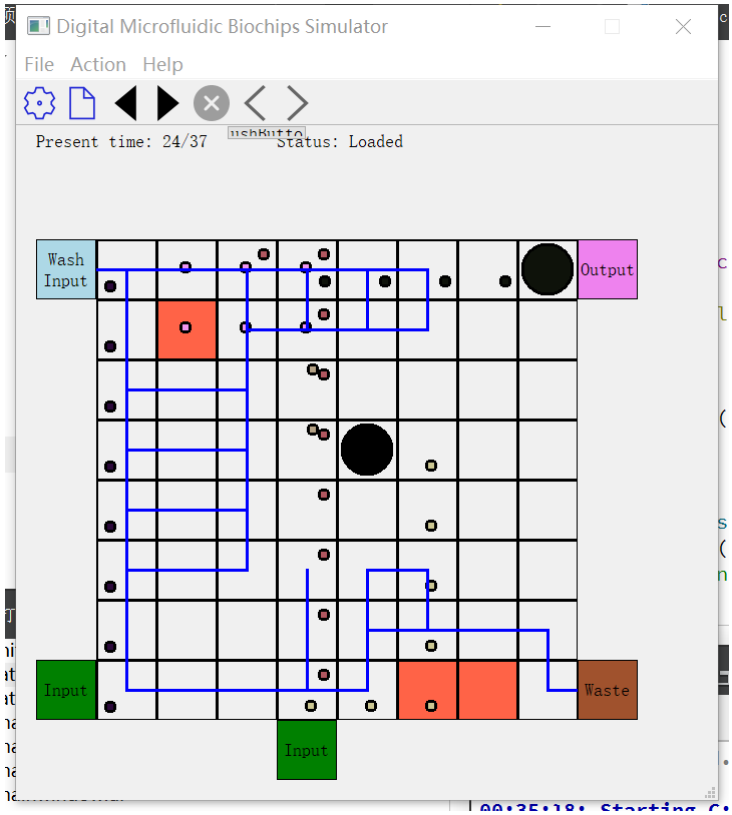
    ```
    QFile f(…);
    …
    QTextStream in(&f);

    while(!in.atEnd()){
        transferOrdertoMovements(in.readLine());//使用
    QTextStream::readLine读入每一行
    }
    ```

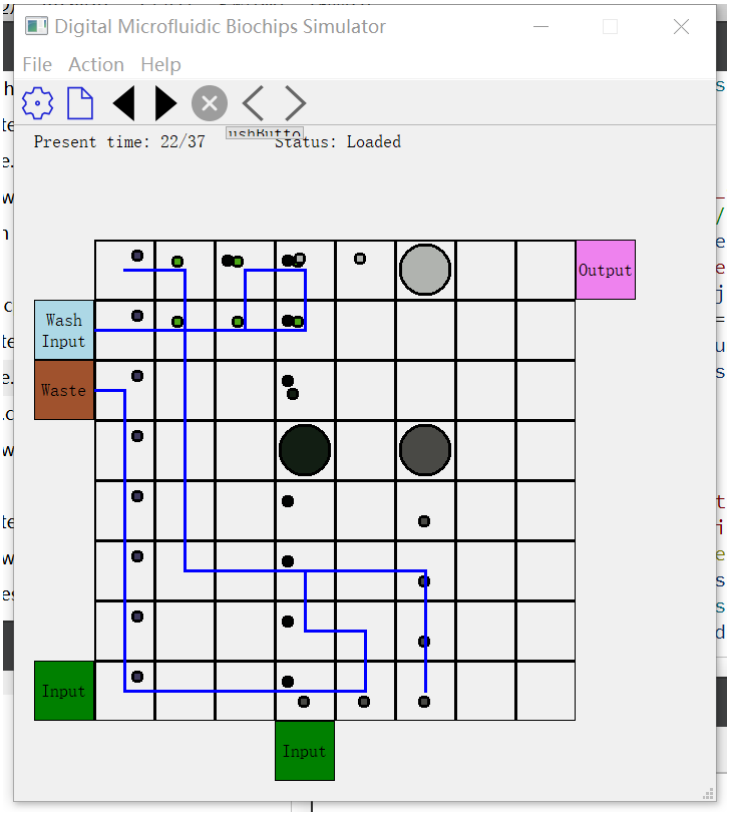
  - 行文本的分割与处理——`QStringList`

    ```
    void MainWindow::transferOrdertoMovements(QString order){
        order=order.left(order.size()-1);//去掉分号
        QStringList list1=order.split(" ");//先按空格分为两段
        QString keyword=list1.at(0);//使用QStringList::at(int)按索引获取某
    段文本
        QStringList list2=list1.at(1).split(",");//将第二段按半角逗号分为两
    段
        int time=list2.at(0).toInt();//QString::toInt()实现QString到int的
    转化
        if(keyword=="Input"){
        …
    }
    ```

- 污迹推演中的增量思想
  - 程序可以开关清洗液滴功能，而开与关在污迹的时刻分布上是质的影响。如果说清洗液滴的引入与否不会改变每个时刻场上液滴的位置和状态（不考虑附加功能），那么某个时刻场上的污迹分布状况对于开和不开启清洗液滴两种选项是完全不同的——若不开启清洗液滴，则每个时刻场上的污迹分布均可唯一确定，这样跳转到任意一个时刻都不会受到时序的影响。而若引入清洗液滴，特别是结合格子的锁定情况这一人为输入的不确定因素，污迹分布则完全随机。
  - 题目中注明了若引入清洗功能，则可以禁用上一步这样在时间上倒序的手段。然而为了保持`Lattice`中绘图事件的统一性，这样的需要特判的例外是不利于逻辑的流畅的。
  - 本程序采用增量思想，虽然一开始的文件解析可以告诉我们所有时刻的液滴和污迹情况，但无论是否引入清洗液滴，程序均不保存绝对的污迹分布而记录污迹增量（这一时刻相对于上一时刻增多的污迹的分布）。在`Lattice`切换时刻前，需在高层窗口(`MainWindow`)中手动调用`lattice->getContaminations(int)`来动态计算某时刻的污迹分布（也就是上一时刻污迹分布加上增量而已），这样便融合了开关清洗的情况，保持了统一。

- 清洗液滴路径规划——bfs/旅行商
  - 在设计目的的显式要求下，由于随时可以锁定格子，很自然的一种处理时序的方式是，对于污迹，采用上述的污迹增量法，而对于锁定的格子，不以时刻为转移，用户锁定某个格子则无论之后切换到哪个时刻，只要不手动解锁，格子的锁定状态不发生变化。
  - 由于污迹分布是动态计算产生，清洗液滴的路径规划也需要根据当前局面动态推演。其算法有两方面的考虑，一是有效性，而是效率。有效性是说，尽最大可能使场上液滴免除污染；效率是说，由于清洗液滴的移动实际也需要时间，合理规划以减少模拟过程的总时间。而有限性的优先级显然要优于效率的考量。
  - 在不重新规划**液滴**的移动的前提下，液滴的分布是确定的，而格子的锁定是不确定的，若某个算法预先模拟若干步，并提供了某种清洗液滴移动过程中时间仍自然流动（即绝对时刻适时向前推进）的清洗路径，这样当然节省了时间，但不能排除一种致命的可能——使用者在某一步堵塞了清洗液滴入口或出口。这样，不时停的清洗液滴路径规划算法的有效性无法保障。
  - 为使有效性最大化，本程序的预设下唯一的方法是每步均时停并清理尽可能多的污迹。问题转化为了在障碍的存在下经过某些必经点，从指定入口到出口的路径规划。使用bfs可以给出理论上最优的算法（在入口在左上角，出口在右下角，二者距离理论上最远的情况下）：从入口开始bfs，得到距入口最近（可经过）的有污迹的格子1，保存路径1；从格子1开始bfs，得到最近的格子2，保存路径2……遍历所有可经污迹后回到出口。
  - 而本程序的清洗液滴入口和出口位置可定制，甚至出现二者紧挨的极端情形（如一个在(1,1)，一个在(1,2)）。本程序采取的算法是bfs和便宜算法结合的处理方式。首先通过bfs搜寻出入口到每一个污迹和每一个污迹到出口的距离，建立带权图；算法的优化目标是合理调整入口和出口间格子的经过顺序，使得入口——格子1——格子2……格子n——出口的总距离最短（当然此处的距离都是曼哈顿距离）。可以发现，若最终加上一条出口——入口的边就转化为了旅行商问题，这是一个NPC问题。曼哈顿距离满足三角不等式，因此使用便宜算法可以得到较好的优化结果。效果示例（冗余路径着实少了很多）：
    - 

（优化前）


Digital Microfluidic Biochips Simulator
File Action Help
Present time: 22/37
Status: Loaded
Output
Wash Input
Waste
Input
Input

（优化后）