

Microfluidic Chip Simulation GUI

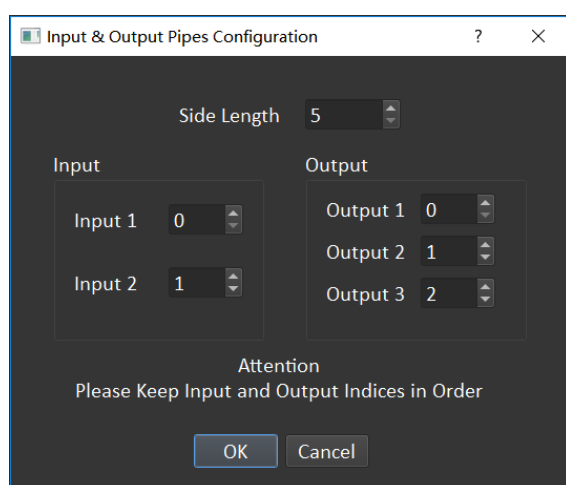
Designer: 计 73 李家昊

程序功能

基础功能

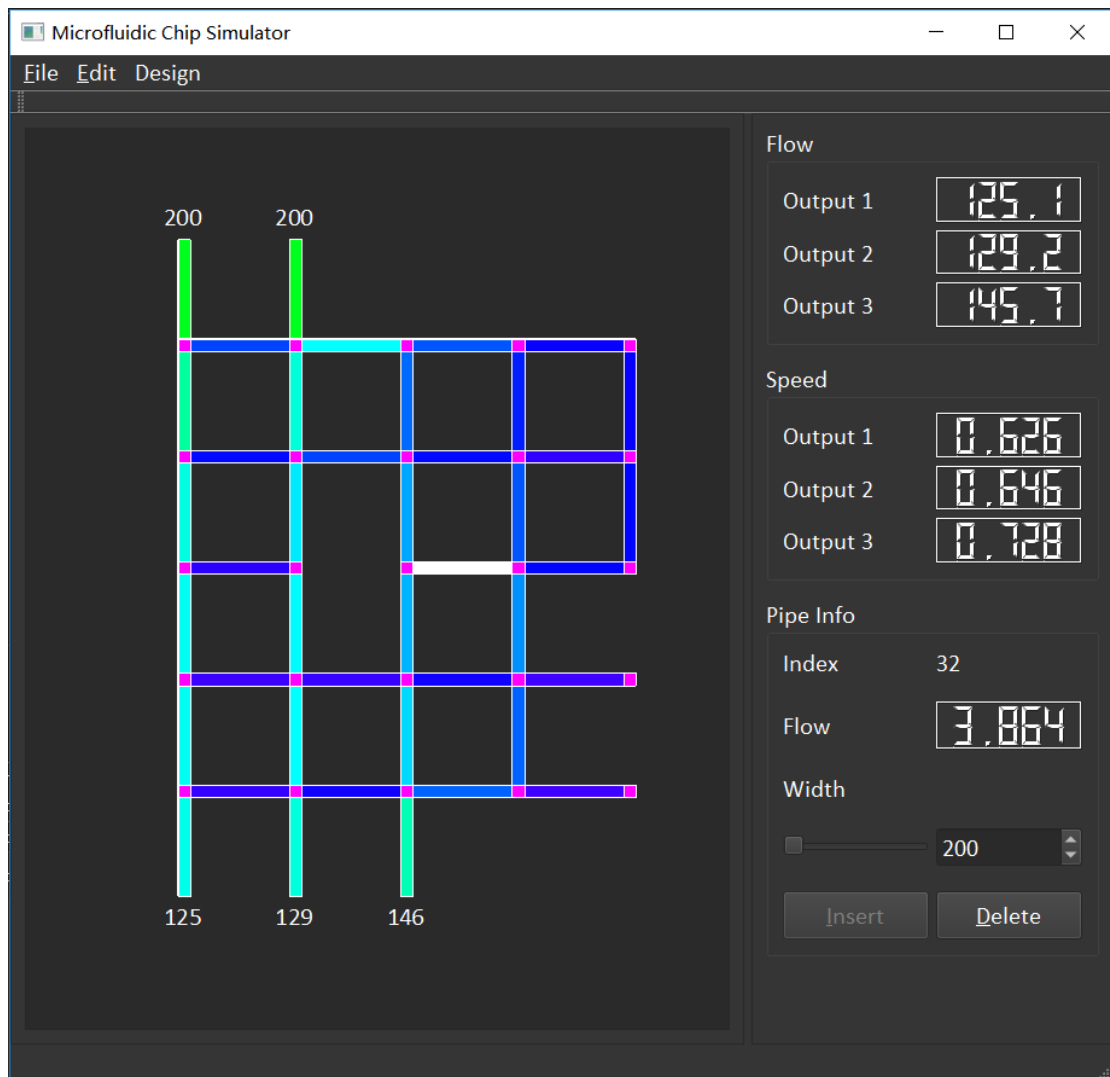
设置行列数

程序启动时，弹出对话框获取用户设置，用户可设置芯片行列数、输入管道位置和输出管道位置。对话框设有输入数据检查功能，当用户输入的数据不合法（如输入管道的位置发生重合），对话框的 ok 按钮会被禁用。当用户点击确认键时，程序根据用户输入的边长和输入输出管道信息，产生一个正方形网格芯片，每条边存在的概率为 10%，输入输出管道除外。



增删管道

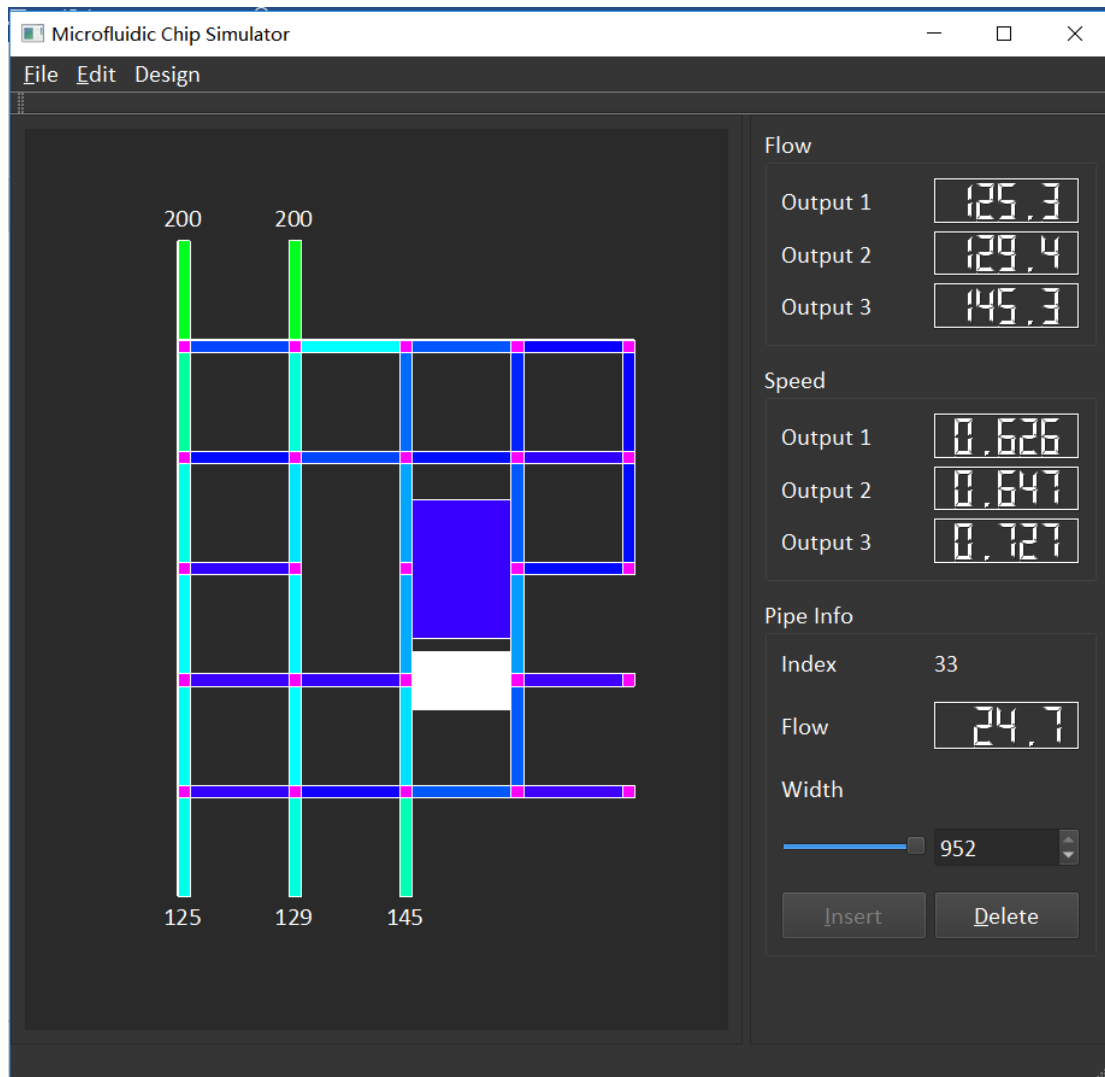
用户选中某一管道时，管道高亮，若管道不存在，则用白色虚线边框表示高亮，若管道存在，则将其填充为白色。选中管道时，界面右侧的功能区中能显示管道信息，Insert 和 Delete 按钮其中之一会被启用，当用户点击 Insert 或 Delete 按钮时，管道会立刻显示或消失在屏幕中，同时后台开始重新计算每条管道的流量和流速，并实时更新在屏幕上。



扩展功能

管道宽度可调

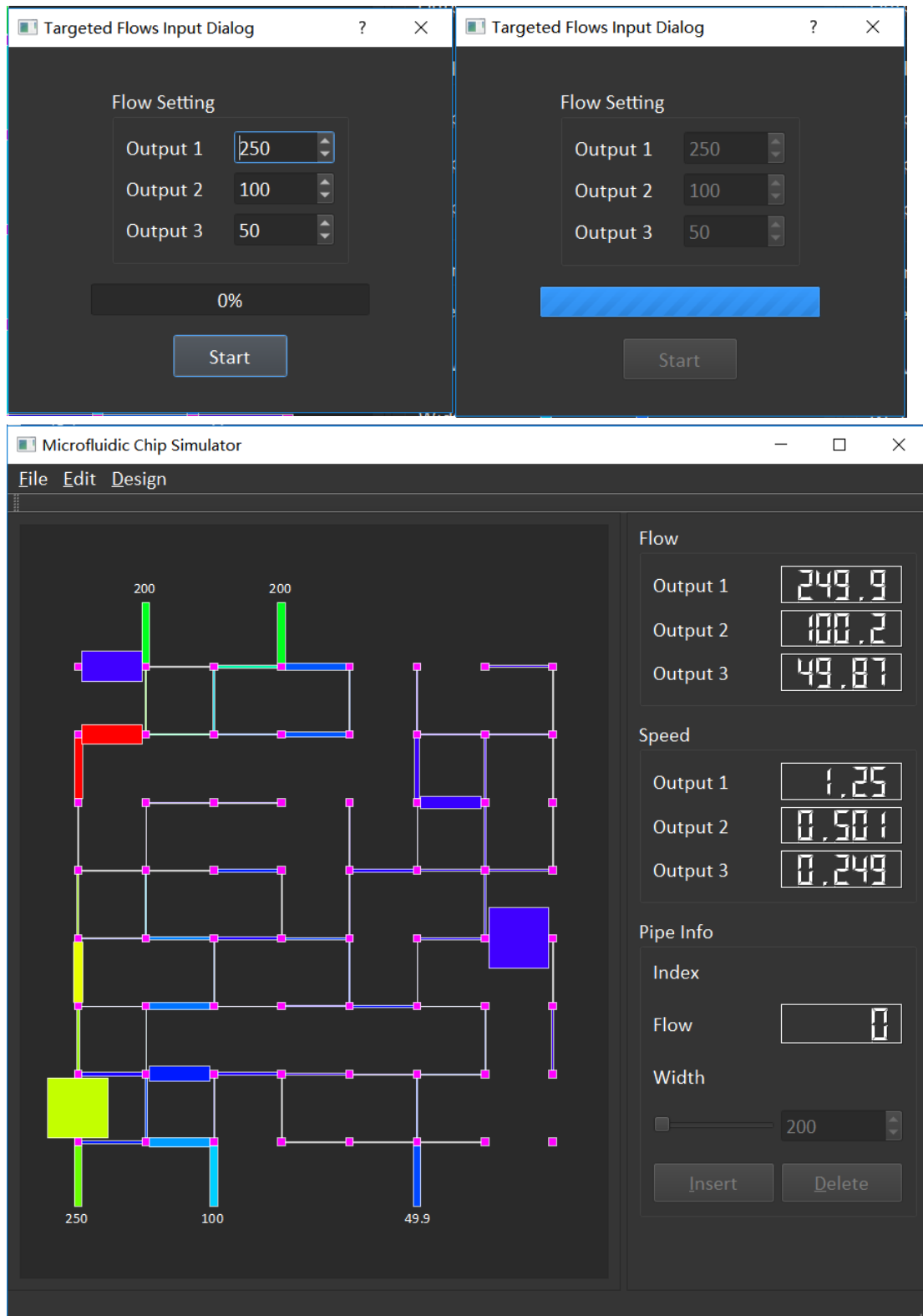
当用户选中某一管道时,若该管道存在且可操作,右侧功能区的宽度滑块和输入框即被启用,用户可通过拖动滑块,调节管道宽度,此时程序检测到宽度改变的信号,会实时在绘图区中更新该管道的形状,并重新计算各个管道的流量和流速,实时更新在功能区和绘图区上。滑块和输入框带有合法性检查功能,在用户点击管道时,程序计算得出该管道的宽度能达到的最大值,并将此最大值设为滑块和输入框的最大值,以保证输入数据的合法性。



芯片反向设计

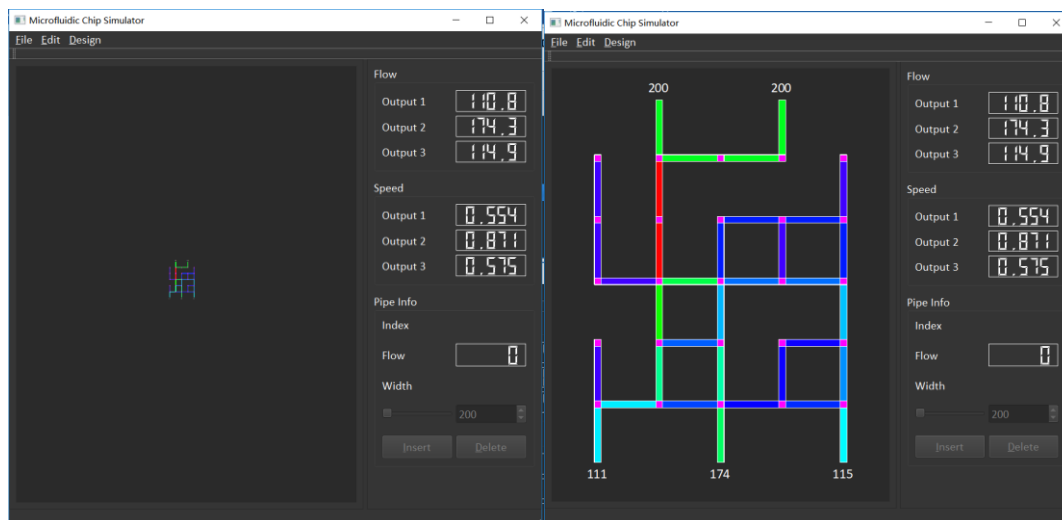
采用遗传算法，将所有管道的宽度信息编码成基因，设置种群大小为 100，每一轮淘汰中，选出前 20 个最适应个体，使他们随机交配产生下一代，并设置变异概率。对于 8*8 的芯片，能在数秒内产生结果，且误差方差不超过 3。

另外，由于计算耗时较长，程序采用多线程技术，将遗传算法放进另一个线程中执行，避免了程序卡死的问题。



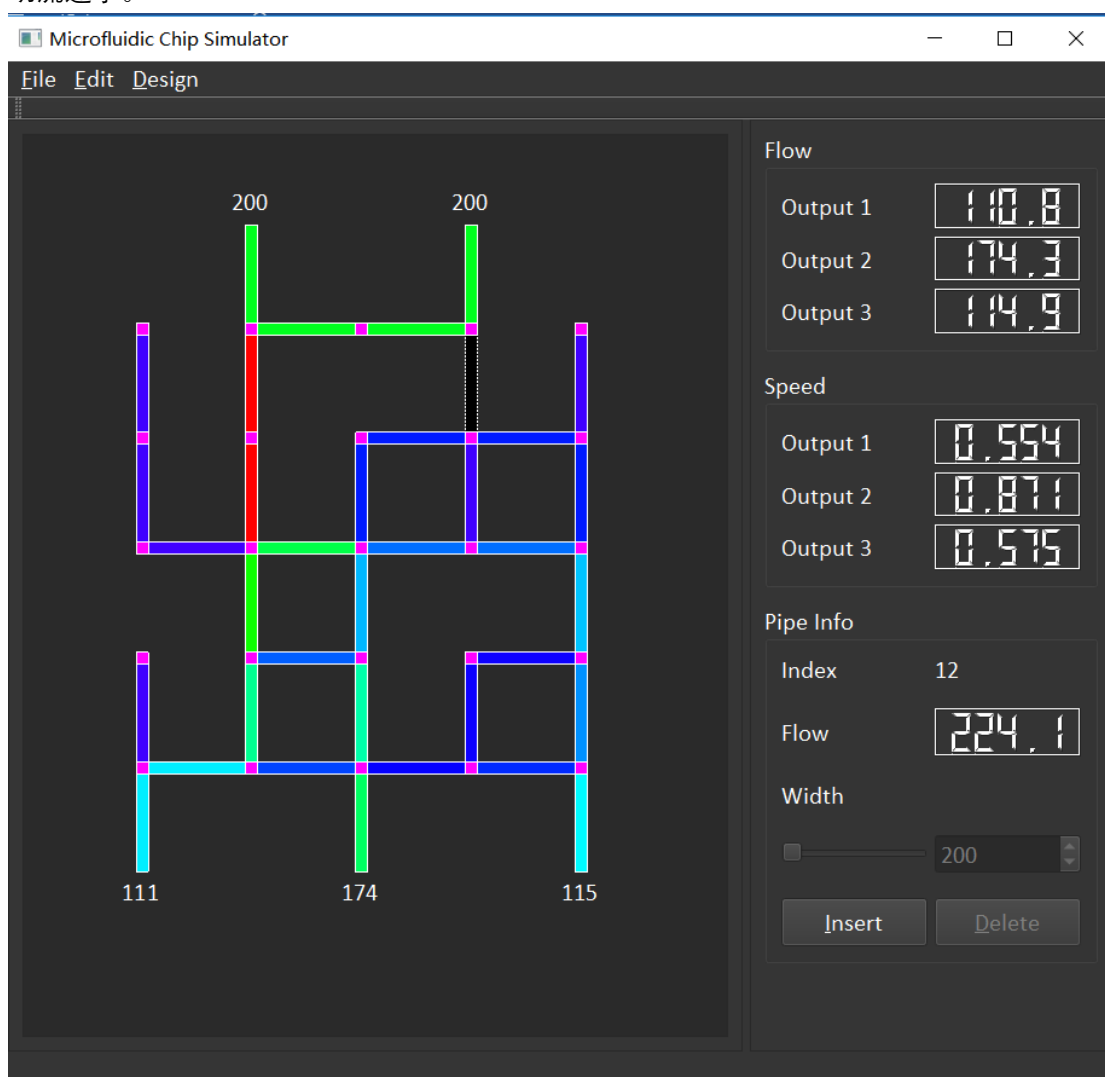
放缩绘图区

绘图区实时监听鼠标滚轮事件，当用户在绘图区滑动鼠标滚轮时，绘图区以鼠标为中心缩放图像，(类似 CAD 软件的缩放操作)，方便操作。



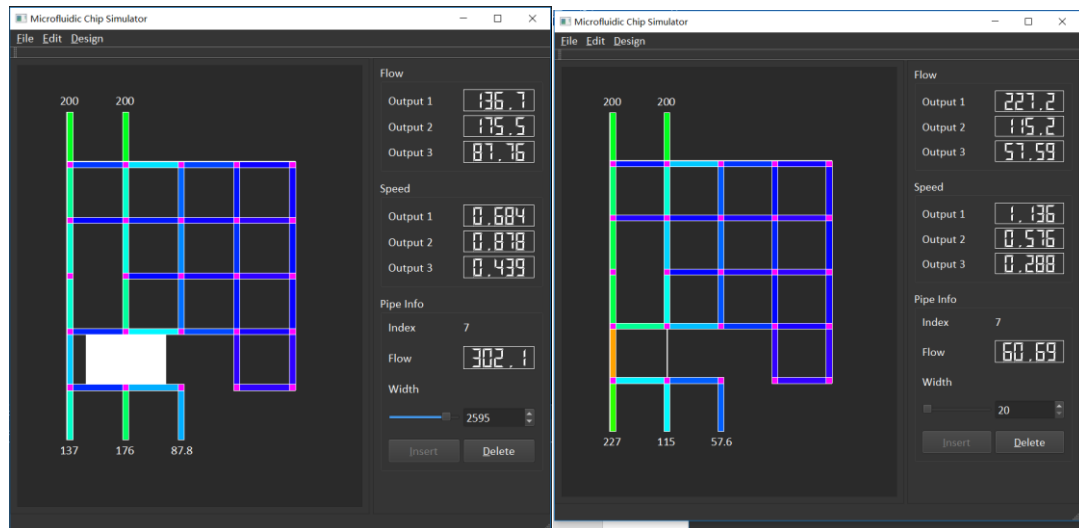
流速可视化

绘制管道时采用 HSV 颜色填充，将不同大小的流量映射到不同的 Hue 值，即可见光谱的不同位置，并实时显示在界面中。颜色可表示所有流量范围(0-400)，红色表明流速大，蓝色表明流速小。



实时计算流速

当用户增删管道，或者拖动宽度调节滑块时，无需重新手动刷新，屏幕就已经显示出更新后的流速。管道流速是动态变化的，绘图区管道的颜色也会动态改变，实时计算和更新为芯片设计者提供了极大的便利。



保存和导入文件

自定义了 AllData 类，其中存放了边长、管道宽度、长度、进出管道位置等信息，并重写了 QDataStream 的 operator<<()和 operator>>()函数，实现通过 QDataStream 写入和读取二进制数据的功能。保存时将 AllData 数据写入后缀为.sim 的二进制文件中，读取时通过 QDataStream 将.sim 内数据读入 AllData，并重新初始化界面和模拟器。

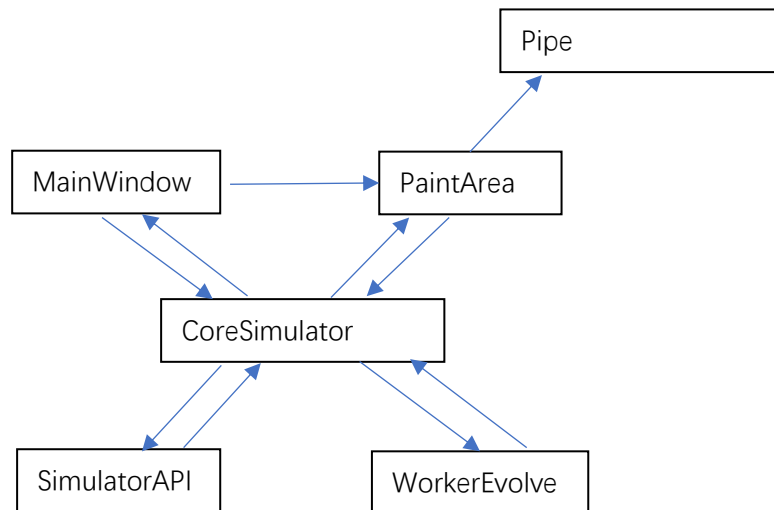
界面风格

在 Qt 官网下载了基于 CSS 设计的 DarkStyle 风格，将其作为全局风格，同时设置了 Calibri 全局字体，使界面更加美观。风格预览页面如下：

https://wiki.qt.io/Gallery_of_Qt_CSS_Based_Styles

程序逻辑

整个程序以 MainWindow 为入口，以 CoreSimulator 为中心控制，显示部分为 MainWindow 和 PaintArea，核心的流量模拟运算则由 CoreSimulator 调用给定的算法 API，遗传算法在自定义的 WorkerEvolve 线程中运行，各个类通过与 CoreSimulator 通信，从而彼此连接起来。



设计感想

本次设计中遇到了很多问题，通过查资料 and 上机实践我已经解决了绝大部分。下面列举几个比较有意义的问题。

Qt 图形视图框架的坐标变换问题

Qt 图形视图框架主要有三个类：QGraphicsView, QGraphicsScene, QGraphicsItem。

对于 Item，要实例化一个 Item，必须重写 paint() 和 boundingRect() 两个虚函数，paint() 函数负责绘制这个 Item，boundingRect() 函数返回一个 QRectF 对象（通常是传给 paint 绘制），那么这个 QRectF 到底是在哪个坐标系中定义的呢？查资料发现，如果一个 Item 没有父对象，那么默认这个 Item 的坐标系是自己的坐标系，如果 Item 没有调用过 setPos() 函数，则将自己的坐标系原点设置在 Scene 的 (0,0) 处，如果调用过，则将自己的坐标原点固定在了 Scene 上的某个点处，接下来 Item 处理的坐标都是基于自己的坐标原点的，正因为这一点，除了一些必要的坐标变换之外，每个 Item 处理起来非常方便。

对于坐标变换，Qt 提供了几个常用的函数，如

QGraphicsView::mapFromScene()	// 将 Scene 下的坐标转换到 View 下的坐标
QGraphicsView::mapToScene()	// 将 View 下的坐标转换到 Scene 下的坐标
QGraphicsItem::mapFromScene()	// 将 Scene 下的坐标转换到 item 下的坐标
QGraphicsItem::mapToScene()	// 将 Item 下的坐标转换到 Scene 下的坐标
QGraphicsScene::itemAt()	// 检测 Scene 坐标下对应的是哪个 Item

实际应用：当需要检测一个管道时，可以在 view 的函数中，先从 View 下坐标转到 Scene 下的坐标，再动态转型检查是否是一个管道 Item，再判断需要做的事情，最后 update Scene 即可触发 paint() 函数。

```

QPointF pos = this->mapToScene(event->pos());
if(QGraphicsItem *item = scene.itemAt(pos, QTransform()))
{
    if(Pipe* pipeItem = dynamic_cast<Pipe*>(item))
    {
        //do something
    }
}
  
```

```

    }
    else
    {
        // do something
    }
    scene.update();

```

UI 设计问题

以前我使用 Qt 时是不会用 Qt Designer 创建界面的，都是纯代码控制，现在发现使用 Designer 能节省很多时间，修改起来也很方便，还能预览界面。比如下面 MainWindow 的界面，如果用代码方式控制，将会非常麻烦。当然也可以用代码和 Designer 搭配，设计出更美观的界面。

Object	Class
▼ MainWindow	QMainWindow
▼ centralWidget	QWidget
▼ layoutMain	QHBoxLayout
frameL	QFrame
▼ frameR	QFrame
▼ groupBoxFlow	QGroupBox
labelFlowOut1	QLabel
labelFlowOut2	QLabel
labelFlowOut3	QLabel
lcdFlow1	QLCDNumber
lcdFlow2	QLCDNumber
lcdFlow3	QLCDNumber
▼ groupBoxPipeInfo	QGroupBox
btnDelete	QPushButton
btnInsert	QPushButton
labelIndex	QLabel
labelIndexNum	QLabel
labelPipeFlow	QLabel
labelPipeWidth	QLabel
lcdPipeFlow	QLCDNumber
sliderWidth	QSlider
spinBoxWidth	QSpinBox
▼ groupBoxSpeed	QGroupBox
labelOut1_2	QLabel
labelOut2_2	QLabel
labelOut3_2	QLabel
lcdSpeed1	QLCDNumber
lcdSpeed2	QLCDNumber
lcdSpeed3	QLCDNumber
verticalSpacer	Spacer
▼ menuBar	QMenuBar
▼ menu_File	QMenu
actionNew	QAction
actionOpen	QAction
actionSave	QAction
separator	QAction
actionClose	QAction
▼ menu_Edit	QMenu
actionPipeIO	QAction
▼ menuDesign	QMenu
actionFlowDesign	QAction
mainToolBar	QToolBar
statusBar	QStatusBar

MVC 模式的尝试

模型(model) – 视图(view) – 控制器(controller)

本次设计尝试使用 MVC 模式，基本保证了 MVC 的分离，M 为给定的 API，充当核心模型，C 为 CoreSimulator，它基本不参与显示计算任务，只负责调用 Model，读取计算数据，再

传递参数给 View(MainWindow, PaintArea), 而 MainWindow 和 PaintArea 除了坐标、尺寸等显示相关的计算, 也同样没有参与任何核心计算。这种模式让整个程序结构更加清晰, 增强了可维护性和可拓展性。