南 开 大 学

高 级 语 言 程 序 设 计

实验报告

专　　业：工科试验班

姓　　名：郭大玮

学　　号：2112052

班　　级：0972

日　　期：2022 年 5 月 7 日

目录

一、 作业题目

1.1 选题内容

基于 Qt 实现节点式图像处理软件

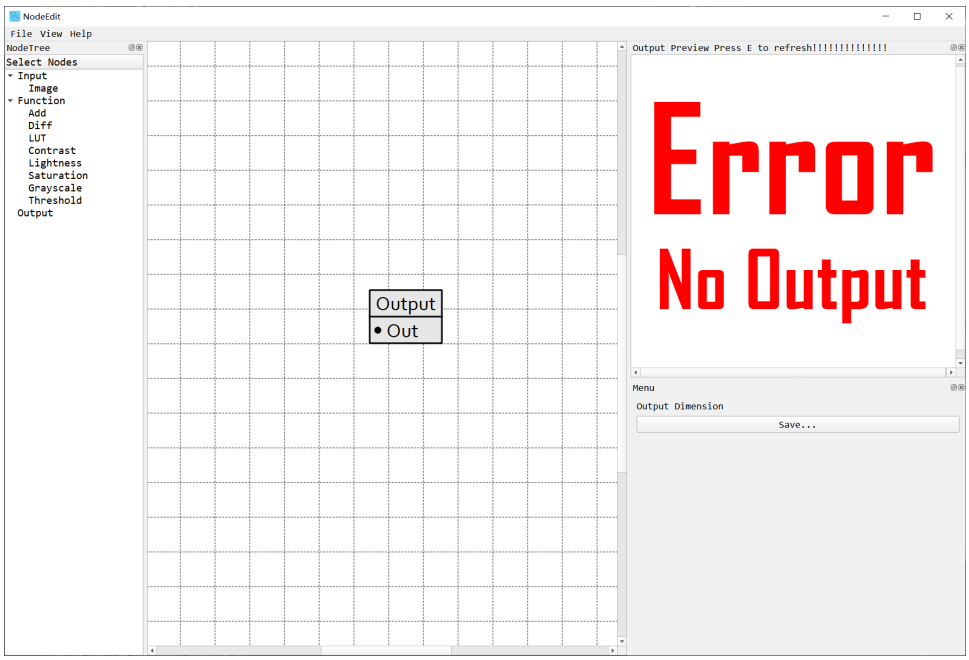


图 1: 主界面

1.2 节点式编辑器简介

对于传统的层级式的操作界面（如 Adobe Photoshop），如果在多项操作后忽然发现需要更改之前操作的某些参数，如一级调色的某些参数、色彩配置文件的映射等时，往往需要一直撤销后重新制作，不利于美术人员的实时修改。而节点式操作界面保留了所有计算过程与计算参数，可以根据需要任意调节，解决了这一问题。

在节点式操作界面中，任意函数（非隐函数）都可以抽象成一个节点，称作函数节点，一个函数节点可以有多个输入、多个输出。同时导入与导出这一操作也都抽象为只有一端的节点，称作输入节点和输出节点，由连线代表输入输出关系。如下图表示的就是一个二输入一输出节点，其数学表达为

$$out = Add(in1, in2)$$

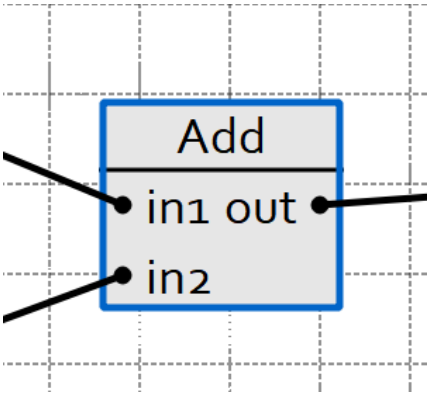


图 2: Add 节点

在触发运行后，软件会逐一读取输入节点的数据，按照流程线上经过的函数节点进行计算，直到遇到输出节点进行输出。

这种表达方式更加接近写代码时的思维，给与使用者更加低层的编辑能力，而又不需要使用者关心函数内部的实现。如下图等效的数学表达为

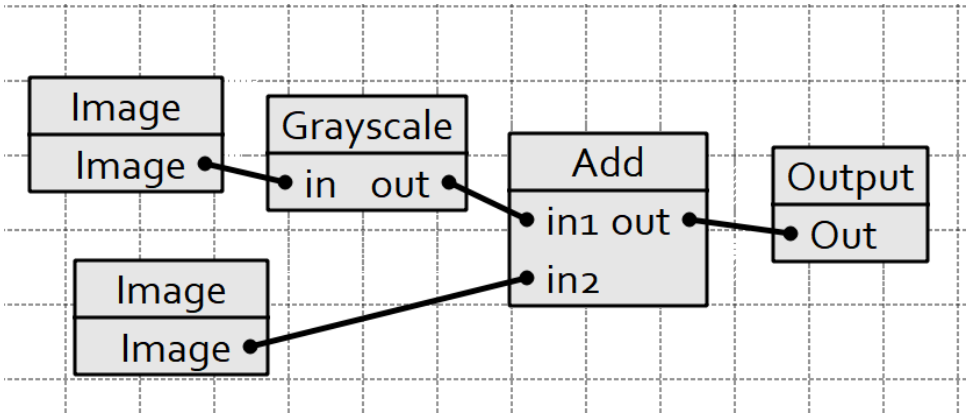


图 3: 节点图示例

$$out = Add(Grayscale(image), image)$$

等效的 C++ 语言（仅为示意，非实际实现）为

```
Image run(Image image1, Image image2){
    Image image3 = Grayscale(image1);
    Image image4 = Add(image3, image2);
    return image4;
}
```

大型专业软件中有很多都采用了节点式的编辑方式，如 3D 软件 Blender 中使

用节点来构建材质、几何、后期编辑，专业视频合成软件 Nuke 使用节点合成视频，工业特效软件 Houdini 使用节点合成特效，在游戏引擎 Unreal Engine 中也引入了节点式的蓝图系统来减少代码需求，控制角色运动。

二、 已实现功能

2.1 操作界面

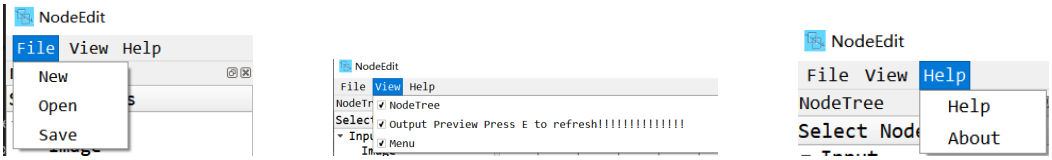
一个简单的操作界面，包括菜单栏、主窗口、节点选择树、图像预览界面和节点属性界面。

所有窗口都可以改变大小（有最大最小限制）、拖动、弹出、关闭。关闭后可以在菜单栏的 View 中重新打开。

在任意处按 E 键都会刷新图像预览窗口，详见 2.1.4。

2.1.1 菜单栏

菜单栏包含三项：文件 (File)、视图 (View)、帮助 (Help)



(a) 菜单栏一 (b) 菜单栏二 (c) 菜单栏三

图 4: 菜单栏

文件 (File)-新建 (New) 删除所有除输出节点外已有节点，新建工程。

文件 (File)-打开 (Open) 呼出文件选择窗，新建工程并打开所选的节点图。

文件 (File)-保存 (Save) 保存当前的的节点图。

视图 (View) 关闭/开启对应的悬浮窗口。

帮助 (Help)-帮助 (Help) 跳转到 Github 显示帮助文档。

帮助 (Help)-关于 (About) 显示关于窗口。

2.1.2 节点树

实现节点树显示所有可选节点，按输入节点、函数节点、输出节点分类排列，并可以从节点树拖拽到主窗口

关于详细的节点功能介绍，参见 2.2 节点部分

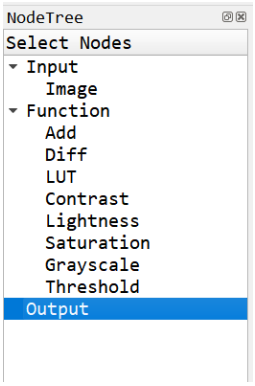


图 5: 节点树

2.1.3 主窗口

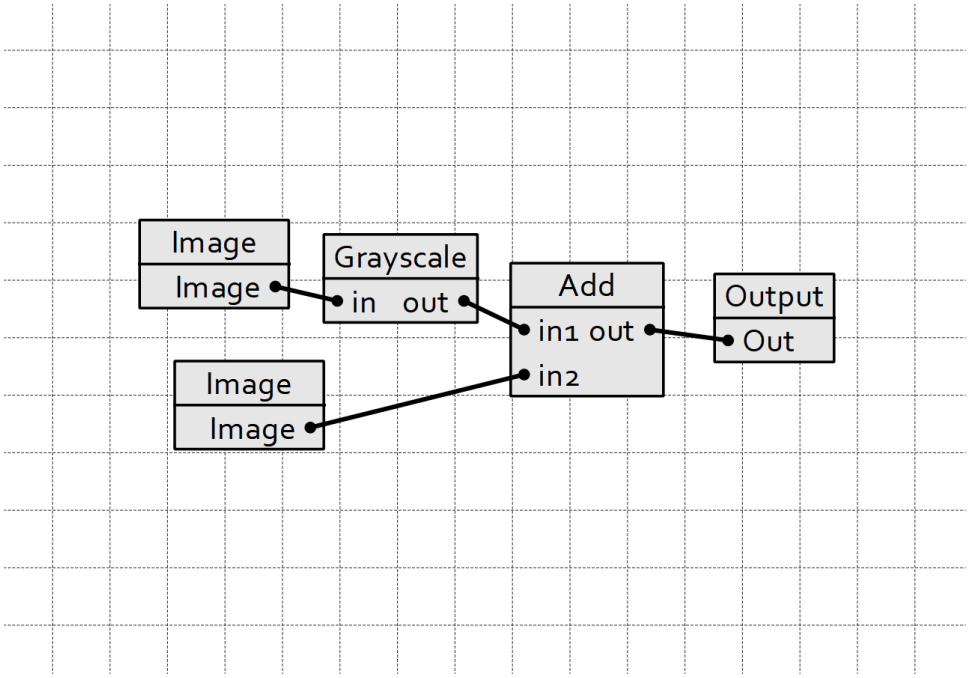


图 6: 主窗口

单击鼠标左键可以选择节点和连线。

选中节点后按住鼠标左键可以移动节点。

滚动鼠标滚轮可以缩放视图，背景网格会随视图一起缩放。

按住鼠标中键（滚轮）移动鼠标可以拖动视图。

从节点树拖拽到主窗口可以新建节点，但输出节点不能新建。

按住节点的输出端口拖拽可以呼出连线，拖拽至另一节点的输入端口可以创建连线，连线不合法时会无法连接。

按住已经连好的连线的尾部（一般连接在后一节点的输入端口）进行拖拽可以修改连线位置。拖拽到空地可以删除连线。

按住已经连好的连线的头部（一般连接在前一节点的输出端口）并不会移动连线，而是呼出新连线，这样做是为了实现一个输入节点作为多个节点的输入。

选中节点或连线后按 Delete 键可以删除节点和连线，但输出节点无法删除。

有关合法连线的内容参见 4.6 节点图求解的实现。

2.1.4 图像预览界面

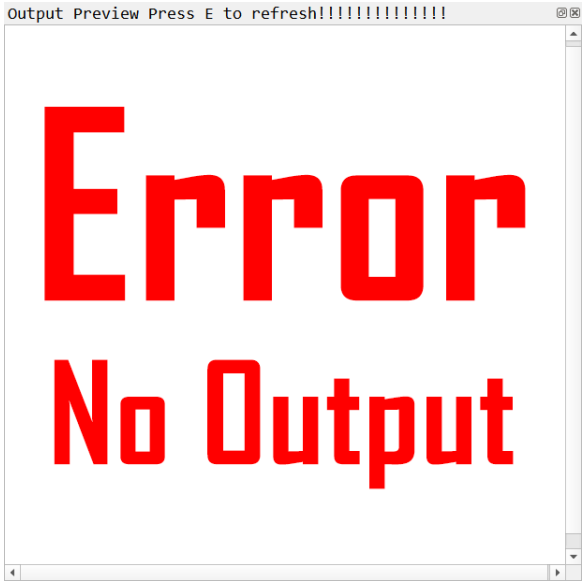


图 7: 图像预览界面

在任何位置按 E 键都可以刷新预览界面，此时程序会运行整个节点图，并将 Output 节点的数据绘制在该窗口中。

当节点图无法计算或无输出时，此窗口会显示预置的 “Error No Output” 图片。

2.1.5 节点属性界面

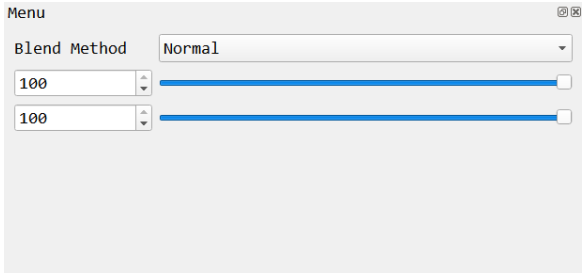


图 8: 节点属性界面

此处会显示正在选中节点的属性界面，可以对节点参数进行修改。

有些节点的打开文件按钮和保存文件按钮会显示在这里。

有关节点参数的详细介绍，参见 2.2 节点部分。

2.2 节点

截止 v0.0.3，已实现的节点共有 10 种

2.2.1 图片输入 (Input::Image)

类型：输入节点

输入：无

输出：[0] 读入图片

功能：

实现图像文件的读取

* 对.jpg 格式 (RGBA8888) 支持较好，对.png 格式部分支持（进行某些操作时透明通道可能会被忽略）

当图片过大 (像素数 > 1920*1080) 会发出警告。

属性界面：

[0]Open Image File <LineEdit>

读取的图片文件路径，无法直接编辑，请点击 Open... 按钮选择文件

[1]Open... <Button>

选择文件按钮，该文件会作为此节点的输出

[2]Dimension <Text>

输入图片尺寸

2.2.2 图片混合 (Function::Add)

类型：函数节点

输入：[0] 输入图片 1（置于上层）

[1] 输入图片 2（置于下层）

输出：[0] 叠加后图片

功能：

实现图像文件的叠加

输出图片尺寸为两输入的较大值

Normal 模式：直接叠加，相当于 PS 里的“正常”

Multiply 模式：乘法，相当于 PS 里的“正片叠底”

Linear Dodage 模式：加法，相当于 PS 里的“线性减淡”

属性界面：

[0]Blend Method <ComboBox>

选择混合模式

[1]<Slider>

输入一混合透明度

[2]<Slider>

输入二混合透明度

2.2.3 图片差值 (Function::Diff)

类型：函数节点

输入：[0] 输入图片 1

[1] 输入图片 2

输出：[0] 差值后图片

功能：

实现图像文件的差值

输出图片尺寸为两输入的差值乘放大倍数

属性界面：

[0]<Slider>

差值结果的放大倍数

2.2.4 图片滤镜 (Function::LUT)

类型：函数节点

输入：[0] 输入图片

输出：[0] 滤镜后图片

功能：

对图像按照 LUT(Look Up Table) 施加滤镜

支持.cube 文件，这是 Adobe 公司规定的一种 LUT 文件格式，记录了 RGB 三维空间到另一 RGB 三维空间的映射，常用来作为图片滤镜或色彩空间转换。

仅支持 3DLUT，内置了一些 LUT 供测试。

属性界面：

[0]Open LUT File <LineEdit>

LUT 文件路径，无法直接编辑，请点击 Open... 按钮选择文件

[1]Open... <Button>

选择 LUT 文件按钮，节点会对输入按照此文件运算后输出

2.2.5 对比度 (Function::Contrast)

类型：函数节点

输入：[0] 输入图片

输出：[0] 处理后图片

功能：

修改图片对比度

采用的算法对 RGB 分别计算，可能效果会不理想

属性界面：

[0]<Slider>

对比度，100 为原图，<100 降低对比度，>100 提高对比度

2.2.6 亮度 (Function::Lightness)

类型：函数节点

输入：[0] 输入图片

输出：[0] 处理后图片

功能：

修改图片亮度

属性界面：

[0]<Slider>

亮度，100 为原图，<100 降低亮度，>100 提高亮度

2.2.7 饱和度 (Function::Saturation)

类型：函数节点

输入：[0] 输入图片

输出：[0] 处理后图片

功能：

修改图片饱和度

属性界面：

[0]<Slider>

饱和度，100 为原图，<100 降低饱和度，>100 提高饱和度

2.2.8 灰阶 (Function::Grayscale)

类型：函数节点

输入：[0] 输入图片

输出：[0] 灰阶图片

功能：

生成灰阶图片

属性界面：

[0]<Slider>

预乘值，100 为原图，<100 偏暗，>100 偏亮

2.2.9 二值化 (Function::Threshold)

类型：函数节点

输入：[0] 输入图片

输出：[0] 二值化图片

功能：

图片二值化

属性界面：

[0]<Slider>

阈值，亮度低于预置的像素将被设为黑色，亮度高于预置的像素将被设为白色

2.2.10 输出 (Output::Output)

类型：输出节点

输入：[0] 要输出的图像

输出：无

功能：

将图像输出到图片预览界面、保存图片。

属性界面：

[0]Output Dimension <Text>

输出图片尺寸

[1]Save... <Button>

保存按钮，将当前要输出的图像保存到指定文件（仅支持.jpg）

2.3 工程文件

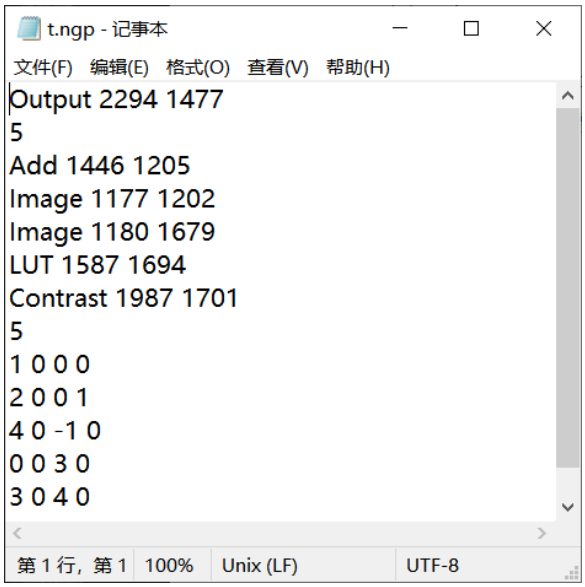


图 9: .ngp 工程文件

我们定义一种新的文件格式 (.ngp) 用于储存节点图文件

该文件按照纯文本形式存储。可以存储所有节点的位置信息和连线信息，暂时不支持存储节点属性，将会在后续更新中加入

在菜单栏 File-Save 可以保存工程文件，File-Open 可以读取工程文件

关于文件内容的详细定义，见 4.7 ngp 工程文件保存与打开的实现

三、 开发环境

编程语言：C++

使用框架：Qt5.9

使用 IDE：Qt Creator 4.3.0

四、 实现方式

4.1 类图

图 10: 类图

4.2 节点视图窗口的实现

定义一个自定义类 NodeView 继承 QGraphicsView 类，并重写鼠标点击、滚轮事件、鼠标拖入事件，用于实现视图的缩放、移动，同时存储所有节点的指针，定义添加节点函数。

```
class NodeView : public QGraphicsView
{
public:
    explicit NodeView(NodeScene *scene, QLabel *imagePreview, QWidget
        *parent = nullptr);
    ~NodeView();
    void wheelEvent(QWheelEvent *event) override;
    void mousePressEvent(QMouseEvent *event) override;
    void mouseMoveEvent(QMouseEvent *event) override;
    void mouseReleaseEvent(QMouseEvent *event) override;
    void dropEvent(QDropEvent *event) override;
    void dragMoveEvent(QDragMoveEvent *event) override;
    void dragEnterEvent(QDragEnterEvent *event) override;
    void keyPressEvent(QKeyEvent *event);
    void appendNode(QString name, QPointF point);
    GraphSolver *_solver;
    NOutput *_outNode;
    QLabel *_imagePreview;
    QList<NodeItem *> _items;
private:
    NodeScene *_nodeScene;
    QPointF _centerAnchor;
    QPoint _posAnchor;
    bool _isMousePressed = false;
};
```

定义一个自定义类 NodeScene 继承 QGraphicsScene 类，并重写鼠标点击、按键事件、定义连线函数（已连接和未连接），并定义槽函数用于实现同一时间只有一个节点被选中，并定义函数实现和 NodeView 的共享节点列表

```
class NodeScene : public QGraphicsScene
{
    Q_OBJECT
public:
```

```
    explicit NodeScene(QStackedWidget *menus, QWidget *parent = nullptr);
    void mousePressEvent(QGraphicsSceneMouseEvent *mouseEvent);
    void mouseMoveEvent(QGraphicsSceneMouseEvent *mouseEvent);
    void mouseReleaseEvent(QGraphicsSceneMouseEvent *mouseEvent);
    void dropEvent(QDropEvent *event);
    void dragEnterEvent(QDragEnterEvent *event);
    void dehangConnection();
    void hangConnection(Connection *connection);
    void keyPressEvent(QKeyEvent *event);
    void syncItems(QList<NodeItem *> *items);
    void drawBackground(QPainter *painter, const QRectF &rect);
    Connection *_hangingConnection;
    QStackedWidget *_menus;
    QList<NodeItem *> *_items;
private slots:
    void _clearSelected();
};
```

4.3 节点对象的实现

定义一个抽象基类 NodeItem 继承 QGraphicsItem 类，用于作为所有节点的父类，并重写 paint 方法，通过多态实现绘制不同的节点，封装公用函数，方便后续增加节点。

构建枚举类型记录 Type，方便从 QGraphicsItem* 指针获取子类类型，并启用静态类型转换。

构建菜单 widget 指针，多态实现每个节点不同的属性界面。

定义信号函数，用于实现同一时间只有一个节点被选中。

```
class NodeItem : public QObject, public QGraphicsItem
{
    Q_OBJECT
    Q_INTERFACES(QGraphicsItem)
public:
    NodeItem(NodeScene *nodeScene, QGraphicsItem *parent = nullptr);
```

```
    ~NodeItem();
    QRectF boundingRect() const override;
    void paint(QPainter *painter, const QStyleOptionGraphicsItem
        *option, QWidget *widget) override;
    void mousePressEvent(QGraphicsSceneMouseEvent *event) override;
    void updateMenu() const;
    NodeData* importData(int nodeNum);
    void exportData(int nodeNum, NodeData* data);
    virtual void initializeNode();
    virtual void initializeBoundingRect();
    virtual void initializePort();
    virtual void initializeMenu();
    virtual void execute();
    enum { Type = 65542 };
    int type() const{
        return Type;
    }
    NodeScene *_scene;
    QRectF _boundingRect;
    QString _title;
    QFont _font;
    QList<InPort *> _in;
    QList<OutPort *> _out;
    int _id;
    Nodetype _nodetype;
    QWidget *_menu;
    QGridLayout *_layout;
    QStackedWidget *_pmenus;
signals:
    void clearSelected();
};
```

这样再需要增加节点时就只需要继承 NodeItem，并重写 initializeNode(); 初始化节点内容

initializePort(); 初始化节点端口

initializeMenu(); 初始化节点属性界面

execute(); 节点计算

四个函数就可以快速地创建出新节点，方便程序功能的增加。

4.4 连线的实现

定义一个类 Port 作为基类，继承 QGraphicsItem 类，重写 paint() 函数。构建枚举类型记录 Type，方便从 QGraphicsItem* 指针获取子类类型，并启用静态类型转换。

定义两个类 InPort、OutPort 继承 Port 类，分别作为输入端口和输出端口。

定义一个类 Connetion 继承 QGraphicsLineItem 类，存储连线两边端口 Port 的指针。构建枚举类型记录 Type，方便从 QGraphicsItem* 指针获取子类类型，并启用静态类型转换。实现已有连线和悬空连线的绘制。

```
class Connection : public QGraphicsPathItem
{
public:
    Connection(OutPort *in, InPort *out, Datatype inType, Datatype
        outType, QGraphicsItem *parent = nullptr);
    ~Connection();
    void redrawHanging(QPointF mousePos);
    void redraw();
    QPointF getPosInScene();
    OutPort *_in;
    InPort *_out;
    Datatype _inType, _outType;
    enum { Type = 65543 };
    int type() const{
        return Type;
    }
private:
    QRectF _boundingRect;
    QGraphicsItem *_parent;
};
```

4.5 节点储存自定义数据类型的实现

定义一个 NodeData 类储存所有可能的数据类型的指针，并定义各种构建函数，这样只需要在节点输出时，只存储一个 NodeData 的指针便可以存储各种数据类型。

虽然这一点在现在没有得到真正的应用，但在后续加入不同类型的输入节点时，可以实现互相转换，并能够检查类型不匹配的错误，也能更好地拓展节点数量。

```
enum Datatype{
    imageData, intData, stringData, unknownData
};
class NodeData
{
public:
    NodeData();
    NodeData(QImage *image);
    NodeData(QString &string);
    NodeData(int i);
    QImage *_image;
    int _int;
    QString *_string;
    Datatype _datatype;
};
```

4.6 节点图求解的实现

先定义几个概念

我们将下图这种一个输出端口连接多个输入端口的环称为第一类环

对于第一类环，我们可以通过将连接多个输入端口的节点进行拆分，将其复制一份。

这样就从图结构变化为树结构。

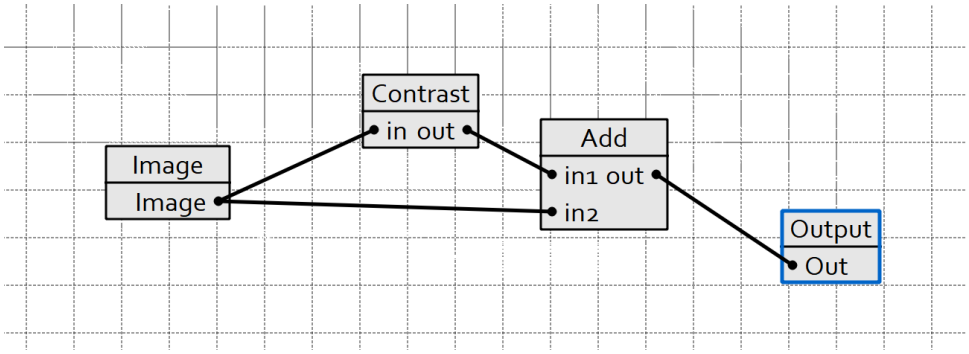


图 11: 第一类环

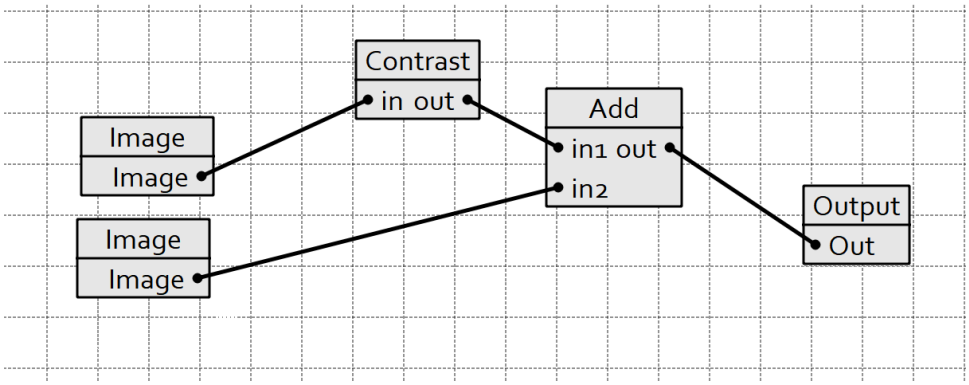


图 12: 第一类环的解环法

我们将下图这种某个输出端口连接到其前端依赖节点的输入的环称为第二类环

这种环不合法，无法计算。后续我会实现这种环的检测，现在版本中请避免这种连接

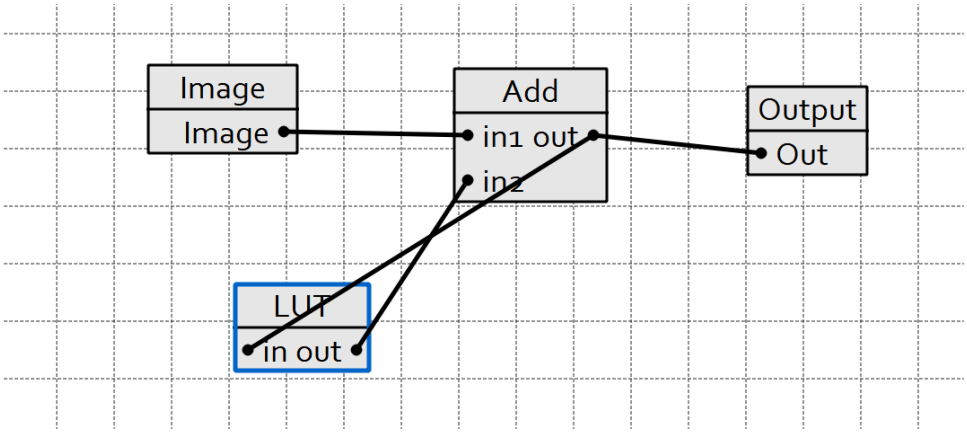


图 13: 第二类环

在对第一类环进行拆解后，不考虑第二类环下，节点图变为一个纯指针存储下树的遍历问题。

这里求解使用深度优先遍历方法，以输出节点作为根节点，将节点图中深度

最深的节点先进行计算，逐级向根节点更新，这样可以保证后一节点计算时所依赖的节点数据都已经计算完成。

4.7 ngp 工程文件保存与打开的实现

一个典型的工程文件如下

```
Output 2294 1477
5
Add 1446 1205
Image 1177 1202
Image 1180 1679
LUT 1587 1694
Contrast 1987 1701
5
1 0 0 0
2 0 0 1
4 0 -1 0
0 0 3 0
3 0 4 0
```

ngp 文件定义为

```
//第一行 输出节点的坐标x, y
Output x y
//第二行 除输出节点外节点个数N
N
//接下来N行 每行一个节点的名称、坐标
title x y
//...
//第N+3行 连线个数M
M
//接下来M行 每行一个连线
//表示从上面给出的N个节点中的第inNode个节点(下标0开始)的第inPort端口
//连接到第outNode个节点(下标0开始)的第outPort端口
//其中当outNode=-1时，表示连接到输出节点
```

```
inNode inPort outNode outPort
//...
//END
```

保存和读取工程文件部分代码

```
void NGPHandler::save(QString filename){
    if(!filename.isNull()){
        QFile file(filename);
        bool isReady = file.open(QIODevice::WriteOnly);
        if(isReady){
            QList<QGraphicsItem *> itemList = _scene->items();
            QList<NodeItem *> nodeitems;
            QList<Connection *> connections;
            NOutput *output = nullptr;
            for(QGraphicsItem *item : itemList){
                if(item->type() == NodeItem::Type){
                    nodeitems.push_back(static_cast<NodeItem *>(item));
                }else if(item->type() == Connection::Type){
                    connections.push_back(static_cast<Connection *>(item));
                }else if(item->type() == NOutput::Type){
                    output = static_cast<NOutput *>(item);
                }
            }
            QTextStream filestream(&file);
            if(!output){
                file.close();
                return;
            }
            filestream << output->_title << ' ' << output->pos().x() << '
                ' << output->pos().y() << endl;
            filestream << nodeitems.size() << endl;
            for(NodeItem *item : nodeitems){
                filestream << item->_title << ' ' << item->pos().x() << '
                    ' << item->pos().y() << endl;
```

```
            }
            filestream << connections.size() << endl;
            for(Connection *con : connections){
                filestream << nodeitems.indexOf(con->_in->_parent) << ' '
                    << con->_in->_parent->_out.indexOf(con->_in) << ' ' <<
                    nodeitems.indexOf(con->_out->_parent) << ' ' <<
                    con->_out->_parent->_in.indexOf(con->_out) << endl;
            }
        }
    file.close();
    }
}
void NGPHandler::open(QString filename){
    if(!filename.isNull()){
        QFile file(filename);
        bool isReady = file.open(QIODevice::ReadOnly);
        if(isReady){
            QTextStream filestream(&file);
            QString title;
            int x, y;
            filestream >> title >> x >> y;
            _view->_outNode->setPos(x, y);
            int nItems, nConnections;
            filestream >> nItems;
            for(int i = 0; i < nItems; i++){
                filestream >> title >> x >> y;
                _view->appendNode(title, QPointF(x, y));
                qDebug() << _view->_items.size();
            }
            filestream >> nConnections;
            int fromNode, fromPort, toNode, toPort;
            qDebug() << _view->_items.size();
            for(int i = 0; i < nConnections; i++){
                filestream >> fromNode >> fromPort >> toNode >> toPort;
```

```
                if(toNode >= 0){
                    Connection *con = new
                        Connection(_view->_items[fromNode+1]->_out[fromPort],
                        _view->_items[toNode+1]->_in[toPort], unknownData,
                        unknownData);
                    _scene->addItem(con);
                    con->_in->_connections.push_back(con);
                    con->_out->_connections.push_back(con);
                    con->redraw();
                }else{
                    Connection *con = new
                        Connection(_view->_items[fromNode+1]->_out[fromPort],
                        _view->_outNode->_in[0], unknownData, unknownData);
                    _scene->addItem(con);
                    con->_in->_connections.push_back(con);
                    con->_out->_connections.push_back(con);
                    con->redraw();
                }
            }
        }
    file.close();
    }
}
```

4.8 节点功能的实现

由于篇幅限制，这里不过多赘述，仅介绍 LUT 节点的实现。

根据 Adobe 官方文档，逐行读取.cube 文件并将三维查找表存储于 Vector 中。

根据

$$Line = R + G * N + B * N * N$$

查找颜色映射坐标，之后对两个最近相邻的颜色坐标进行线性插值。

部分实现代码

```
QColor NLut::applyLut(QColor c){
    if(_fileName.isEmpty() || !_dataIsReady)
        return c;
    double r = c.redF() * (_lutSize - 1);
    double g = c.greenF() * (_lutSize - 1);
    double b = c.blueF() * (_lutSize - 1);
    int rH = ceil(r);
    int rL = floor(r);
    int gH = ceil(g);
    int gL = floor(g);
    int bH = ceil(b);
    int bL = floor(b);
    double frH = _lutData[rH + gH * _lutSize + bH * _lutSize *
        _lutSize][0];
    double fgH = _lutData[rH + gH * _lutSize + bH * _lutSize *
        _lutSize][1];
    double fbH = _lutData[rH + gH * _lutSize + bH * _lutSize *
        _lutSize][2];
    double frL = _lutData[rL + gL * _lutSize + bL * _lutSize *
        _lutSize][0];
    double fgL = _lutData[rL + gL * _lutSize + bL * _lutSize *
        _lutSize][1];
    double fbL = _lutData[rL + gL * _lutSize + bL * _lutSize *
        _lutSize][2];
    int fr = 255 * (frL + (r - floor(r)) * (frH - frL));
    int fg = 255 * (fgL + (g - floor(g)) * (fgH - fgL));
    int fb = 255 * (fbL + (b - floor(b)) * (fbH - fbL));
    return QColor(fr, fg, fb);
}
```

五、 测试

进行了大量的测试，受制于时间限制，有一些 bug 没有来得及修正，欢迎在 Gtihub 和 Gitee 上 Report

特别感谢 2111334 刘宗桓在测试中提供的帮助。

受制于篇幅限制，不过多赘述。

六、 总结

收获甚丰矣。

受制于篇幅限制，不过多赘述。

特别感谢 2111876 梅骏逸在程序编写中提供的帮助。

Github Repository

Gitee Repository

Bilibili Link