EasyPaint设计报告

简介

仿照photoshop制作的一个极简绘图工具，提供最基本的绘图功能，重点关照用户绘图时的体验。

制作过程中，我发现了photoshop很多不引人注意，但却能极大改善体验的小细节，例如：通过滑块调节画笔尺寸时，尺寸变化先慢后快；按Alt+滚轮放大缩小时，会自动以鼠标位置为中心缩放，让用户不用放大后不需要再调整位置；鼠标不松开地连续绘制透明颜色时，前后两笔的颜色不会叠加，保持透明度；这些细节恰恰是我在制作了一个最最最简陋的绘图工具后发现的问题，ps都非常妥善地将其改善了。

最终的这份东西中，绘图工具仅有最基本的画笔、橡皮、颜料桶。但上述photoshop的优秀细节，我都以自己的方式实现了。

此设计文档中，涉及代码的部分基本只包含头文件中的声明，主要讲述其负责的功能，具体定义可以查看附件。

操作手册

菜单栏操作

- 单击菜单栏/文件/新建 或Ctrl+N——创建一个新的空画布；
    - 在弹出的对话框中设置画布尺寸，点击锁按钮可以锁定宽高比
    - 可以创建多画布
- 单击菜单栏/文件/打开 或Ctrl+O——打开一张本地图片
- 单击菜单栏/文件/保存 或Ctrl+S——将当前绘制的内容（除隐藏图层外）保存为图片
- 单击工具/工具箱 或工具箱上方按钮——呼出或关闭工具箱
- 单击编辑/撤销 或Ctrl+Z——撤销上一次操作，可以撤销的内容包括绘制、创建和删除图层操作

绘制操作

- 初始界面左侧为工具箱，可以拖出来作为浮窗或放到右边，可关闭，也可以单击 菜单栏/工具/工具箱 来呼出；
    - 工具箱中有三种工具（画笔，橡皮，颜料桶），单击选中工具后，下方设置栏将切换为对应工具的设置选项
    - 在设置面板中，可以调整尺寸，颜色，容差，稀疏度等；
        - 通过滑动滑条调整尺寸等数值
        - 点击颜色块，弹出一个调色盘以调整颜色，透明度也在此调整
- 在打开一个画布后，用鼠标即可在中央画布上用所选工具绘制
- 画布右侧为图层面板，选中哪个图层就是在哪个图层上绘制
    - 单击图层上方按钮可以创建、删除、隐藏/显示图层
    - 拖动图层可以调整顺序，上面的图层显示在上方。
    - 双击图层可以重命名
- 画布左侧有一竖直滚动条，滑动可以放大缩小画布（只是缩放显示，并不改变画布实际大小），也可以按住Alt+鼠标滚轮缩放，通过Alt+鼠标滚轮缩放时，将以鼠标当前指向的画布作为缩放中心。

计划功能

- 菜单（文件，编辑，工具）
    - 文件：新建画布，打开和保存图片文件
    - 编辑：撤销
    - 工具：显示/关闭工具箱
- 工具箱（画笔，橡皮擦，颜料桶）
- 主界面（支持多文档）
- 子文档内：
    - 图层面板（添加，删除，隐藏，拖拽调整顺序）
    - 画布（绘制）
    - 缩放条（缩放显示）
- 可能会做的功能：历史记录（√），图的移动，选框操作

设计结构

结构总览

其中，括号内为该层部件对应的类，自定义类用粗体标识，自网络引用的类用斜体标识，后跟 - 继承自的类

- MainWindow (QWidget)
    - 菜单栏，工具栏，状态栏等部件
    - 多文档管理器 mdiArea (QMdiArea)
        - 子文档 subWindow1 (**PaintDoc** - QWidget)
            - 图层操作按钮 addLayerButton等 (QPushButton)
            - 图层面板 layerList (**LayerList** - QListWidget)
                - 图层 layer1 (**Layer** - QListWidgetItem)
                - 其他图层 layer2...
            - 绘画区域 scrollArea (QScrollArea)
                - 横竖滑动条，视口等 scrollArea自带部件
                - 内部填充 scrollAreaWidgetContents(QWidget)
                    - 画布 canvas (**Canvas** - QWidget)
            - 尺寸滑动条 scaleSlider (**MySlider** - QSlider)
            - 历史记录 hist (**history**)
        - 其他子文档 subWindow2...
    - 工具箱 ToolBox (QDockWidget)
        - 工具列表 ToolList (QListWidget)
            - 画笔，橡皮，颜料桶 (QListWidgetItem)
            - 设置面板 (**setting_pen setting_eraser setting_bucket** - QWidget)
                - 尺寸，稀疏度，容差设置项 sizeSlider (**MySlider** - QSlider)
                - 颜色设置项 colorwidget (**ColorSettingWidget** - QWidget)
                    - 按钮，背景等
                    - 颜色对话框 (*ColorPickDialog* - QDialog)

- 创建画布对话框 newcanvasDialog(**newcanvasDialog** - QDialog)
    - 几个输入框(QLineEdit) 和 按钮 (QPushButton)

具体设计

此处将由下向上介绍各个自定义类的功能用途，涉及到的代码仅包含头文件声明，具体实现可见项目文件。

Layer

```
class Layer: public QListWidgetItem {
public:
    bool visible; //是否可见
    QImage *img;  //图像
    QPoint onepot; //单点绘制特判
    QPainterPath buff; //绘制笔迹缓存区

    Layer(int w, int h, QListWidget *parent = nullptr);
    Layer(const QPixmap &pix, QListWidget *parent = nullptr);
    ~Layer();

    void pushBuffTo(QImage *tar); //将缓存绘制到目标图
    void updateStyle(); //根据visible更改显示样式

    bool hasBuff(); //是否有未绘制的内容（buff或onepot）

    //具体绘制命令
    void releaseEnd(QPoint p); //鼠标释放，将缓存绘制到图像上
    void clickStart(QPoint p); //鼠标按下，开始记录笔迹
    void paintBucket(QPoint p);//油漆桶绘制
    void dragPoint(QPoint p);  //鼠标拖拽，添加一个笔迹点
};
```

图层类，主要处理具体的绘制，它相对独立，我希望它不涉及所处的文档中，与各种缩放比例和文档设置均无关，只接受绘制指令。

这样的设计使得在不同文档间转移图层成为可能。

buff缓存的用途：在用笔或相比橡皮绘制时，实际是拆分成多条小线段进行绘制；**如果使用的是含透明度的颜色，分段绘制时前后两段交点处的颜色就会混合，导致透明度失效。**解决方案即使用一个缓存区，用户拖动鼠标时，并不真的进行绘制，只是将鼠标路径写入缓存，在鼠标松开时再一次性绘制到图像上。

主要用到了 `QPainterPath` 类和 `drawPath` 这个绘图方法。

当然，为了交互的合理性，我们必须实时显示用户绘制的内容，做法就是在显示图像时，将缓存区临时绘制一遍。具体见**Canvas**

LayerList

```
class LayerList: public QListWidget{
    Q_OBJECT
    bool isdragging;  //是否在拖拽
    int curInsertRow; //拖拽时，即将插入的行号
public:
    LayerList(QWidget *parent = nullptr);
protected:
    void mousePressEvent(QMouseEvent *event); //重载各类事件 都是为了实现图层拖拽
    void mouseReleaseEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void dragEnterEvent(QDragEnterEvent *event);
    void dragMoveEvent(QDragMoveEvent *event);
    void dragLeaveEvent(QDragLeaveEvent *event);
    void dropEvent(QDropEvent *event);

    void updateInsertRow(int row); //拖拽鼠标移动时 要实时更新即将插入的行

signals:
    void layerChanged(); //图层变动信号 -> Canvas刷新

};
```

LayerList基本可以当做QListWidget来看，和主体结构关联不大，主要是为了实现**拖拽图层调整顺序。**

实现思路：在鼠标点击某个Item时，先将需要拖拽的图层隐藏，产生一个新的"待插入"项标识插入位置，在拖拽过程中不断根据鼠标当前位置修改"待插入项"的位置，可以在不同图层面板间转移，松开鼠标确定插入时，用原先的图层信息替代待插入项

Qt的QDrag类非常强大，它不仅能实现QListWidget内部项的拖拽，还什么都能拖。其主要原理是用到一个QMimeData对象。

QDrag：我理解为它是一个拖拽的全过程，在鼠标按下时创建QDrag对象，设置好它包含的内容，拖拽时的动画表现等，然后执行exec（它并不会阻塞主事件循环）。

和它密切相关的是**drag，drop**开头的几大事件函数，在拖拽的exec中会触发这些事件，事件的event当中也包含了该QDrag存储的信息，可以通过event设置exec的返回值（可以返回`Qt::MoveAction`、`Qt::CopyAction`等，表达此次拖拽是移动命令还是拷贝命令），可以执行`event->accept()`或`ignore()`。

- 在dragEnterEvent中，调用`accept`或是`ignore`仅代表此处是否允许落点，不会直接返回
- dropEvent中为拖拽落点事件，调用`accept`或是`ignore`会返回，结束exec
- dragMoveEvent 和dragEnter同理，更加细化到所有移动过程都判断是否允许落点，如无需这么细，直接accept即可，但必须也要重写，否则将执行默认的dragMoveEvent，可能会出事。

QMimeData：一个信息类，结合**QDataStream**，可以将各种我们想要通过拖拽传输的数据保存为一个QMimeData，QDrag传输的信息就是一个QMimeData。它还能给数据打上标签，让接收方判断拖进来的是不是自己需要的内容，从而accept或Ignore。

问题：重载mousePressEvent后，双击事件被覆盖？ 图层原来的双击重命名功能失效。

原因：

1. 在mousePressEvent中需要调用父类的mousePressEvent，否则默认的处理全被覆盖了
2. 猜测可能是鼠标一点击就判断为开始拖拽，然后进入drag的exec循环，无法检测到下一次点击？

解决：并不在鼠标点击事件中执行拖拽的exec，而是在鼠标点击后打上一个标记。松开时删除标记，在mouseMoveEvent中判断有无标记，开启拖拽。

Canvas

```
class Canvas: public QWidget {
    Q_OBJECT;
    PaintDoc *fromDoc; //所属文档，从此获取显示比例等设置属性
    LayerList *layers; //源图层列表
public:
    Canvas(PaintDoc *_fromDoc, LayerList *_layers, QWidget *parent = nullptr);

protected:
    void paintEvent(QPaintEvent *event);         //重载绘图事件
    void mousePressEvent(QMouseEvent *event);    //重载各种鼠标事件，主要要在画布阶段检测用户操作，
    void mouseReleaseEvent(QMouseEvent *event); //应用缩放和一些必要的处理后直接把具体的绘图指令传递给Layer
    void mouseMoveEvent(QMouseEvent *event);

    void enterEvent(QEvent *event);  //重载鼠标进入离开事件，实现鼠标在画布内时改变鼠标样式
    void leaveEvent(QEvent *event);
};
```

Canvas画布类，主要处理显示给用户的内容。

paintEvent中将所有图层的图像按顺序绘制到画布上，另外还要将图层的缓存内容也一并绘制上去（见**Layer**）

此处遇到了一个问题：**在图像（显示比例）放的很大时，paintEvent绘制时很会卡**，我的优化方案有两点：

- 先不顾显示比例，将各个Layer的图像混合，最后再根据显示比例放大。
- 让所属文档返回一个**canvasShowRect**，为该图层在**视口**中的截面（具体见**PaintDoc**），paintEvent中仅刷新这个截面
    - 主要用到QPainter的 `setClipRegion` 方法，设置裁剪区域

优化后明显顺滑了很多。

鼠标样式是根据工具来设置的，在用画笔或橡皮时，我希望它显示一个同笔触大小的圆。

一开始使用一张圆的图片作为鼠标指针，出现了一个问题：随着尺寸放大缩小，这个圆的边缘会变得太粗或太细，而我希望圆的边缘保持不变，同时在笔触尺寸极小时，适当放大鼠标指针使用户能看清。

最后采取的方案是放弃使用图片，在绘制鼠标指针时，临时开一个QPixmap，先在这个QPixmap上绘制一个圆，在应用到鼠标指针上，这样就做到边缘粗细不变了。

PaintDoc

```
class PaintDoc : public QWidget {
    Q_OBJECT
    Ui::PaintDoc *ui;
    Canvas *canvas;       //画布
    QList<history> hist; //历史记录列表
public:
    float view_scale;     //显示比例
    int width, height;    //画布大小
    int default_num;      //图层自动名称编号值

    PaintDoc(int w, int h, QWidget *parent = nullptr);
    PaintDoc(QImage sor, QWidget *parent = nullptr);
    ~PaintDoc() override;

    void revoke();               //撤销
    void addHistory(history _h);//新增历史记录

    void autoFixSize();      //自动调整显示比例以符合画布大小
    QSize scaledSize();      //返回按显示比例缩放后应有的尺寸
    QRect canvasShowRect(); //canvas在视口中截面的rect
    QImage combinedImage(); //将所有图层组合得到一个img
protected:
    void wheelEvent(QWheelEvent *ev) override;          //重载滚轮事件 实现Alt+滚轮
定焦缩放
    bool eventFilter(QObject *obj, QEvent *ev) override;//scrollArea自带滚轮事件的
处理，将其过滤掉

public slots:
    //几个图层按钮的槽函数
    void on_addLayerButton_clicked();   //创建图层
    void on_delLayerButton_clicked();   //删除图层，应用于当前图层
    void on_hideLayerButton_clicked();  //隐藏图层，应用于当前图层
    void on_scaleSlider_valueChanged(int val); //滑动条改变值的槽函数，处理缩放和定焦缩
放
};
```

PaintDoc 作为一整个子文档类，其中包括一个QListWidget作为图层面板和一个scrollArea存放画布。

主要包含文档设置，定焦缩放（可能是我自己造的词），图层修改，历史记录，以及作为子文档的“全局变量”提供各种函数。

关于“**定焦缩放**”，即文章开头提到的ps优秀特性之一：缩放时以用户鼠标指针为缩放中心，让用户放大图像后不需要再调整位置。

具体而言，Alt按下且鼠标滚动时，首先获取鼠标在视口中的位置 Pv，若鼠标指针不在视口内则设置 Pv 为视口中点，然后计算Pv 在画布上指向的像素位置 Pc

然后进行缩放，我不太清楚QScrollArea这个类在其内容放大时，滑块采取什么样的移动策略，也不希望关注如此繁琐的机制，只需要定义一点原则：**在缩放之后，要移动滑块使得 Pc 这一像素仍在视口的 Pv 位置。**

道理应该不难理解，那么缩放之后，计算该像素在视口中的新位置 Pv_2，让画布整体做出 $Pv - Pv_2$ 的移动即可，缩小时可能可移动的余量补足，无法准确对焦，就尽量移动即可。

另外，由于QScrollWidget自带滚动事件的处理，，我使用一个事件过滤器，在按下Alt时过滤其滚轮事件，自然顺延到PaintDoc的滚轮事件处理上。

history

```
//历史记录类
class history{
    char type;     //记录类型，'+', '-', '*'
    Layer *target; //操作图层
    QImage oriImg; //对于绘制操作，暴力存储绘前图像
public:
    LayerList *from; //所操作图层来自的layerlist

    history(char _t, Layer* _tar);
    history(char _t, Layer* _tar, QImage *_ori);
    ~history();    //析构函数中分情况处理图层的销毁
    void revoke(); //撤销此条记录
};
```

历史记录部分做的比较简单，只记录绘制和增删图层操作。绘制操作在每次缓存添加到图像时记录，符合人们的主观直觉，记录也只是非常暴力地将绘制前的图像整个存下来...此处理论上可以仅记录包含绘制区域的最大矩形，细节比较麻烦，因为时间原因没有搞了。

增删图层操作方面，我记录操作图层的指针。也就是说在删去一个图层时，并不真的将其删去，仅是从LayerList中移除，加入到一个history中。在history的析构函数中真正地delete此图层。

在PaintDoc中，用一个链表存储历史记录，且仅存储10条，超过10条时，最早的记录将被删除，无法撤销。

MainWindow

```
const int TOOL_PEN = 0;
const int TOOL_ERASER = 1;
const int TOOL_BUCKET = 2;

class MainWindow : public QMainWindow {
    Q_OBJECT
    Ui::MainWindow *ui;
    QWidget *settingWidget[3], *curWidget; //三种工具的设置面板，当前面板指针
    QLabel *statusLabel; //下方状态栏内容
public:

    static MainWindow *ins;   //单例模式
    static int currentTool(); //返回当前所用工具编号

    MainWindow(QWidget *parent = 0);
    ~MainWindow();

    void showSetting(int p);  //设置面板切换为工具p的

public slots:
    //各种菜单栏中的action触发槽函数
    void on_action_New_triggered();  //新建画布
    void on_action_Open_triggered(); //打开图片
```

```
    void on_action_Save_triggered(); //保存图片
    void on_action_Tool_triggered(bool); //开/关工具箱
    void on_action_Revoke_triggered(); //撤销按钮
};
```

MainWindow 主界面，主要处理工具箱中的设置面板切换，各种菜单项action的触发以及文件层面的交互。

使用**单例模式**，主要是为了方便下层Layer等类中能直接获取到当前使用的工具（使用 `MainWindow::currentTool()` ）

文件交互方面，用的是文件对话框和QImage自带的加载、保存函数，实现存取图片都比较自然，略去不谈。

各个动作的触发槽函数也都比较简单，基本都是调用上下层已实现的函数，略去不谈。

setting

```
//画笔设置类
class setting_pen : public QWidget {
Q_OBJECT
public:
    static int size;        //设置属性作为静态变量
    static int sparse;
    static QColor color;

    explicit setting_pen(QWidget *parent = nullptr); //绑定函数都写在构造函数中
    ~setting_pen() override;
private:
    Ui::setting_pen *ui;
};
```

三种工具的设置面板均为UI设计的Widget，主界面上，可以直接通过掉包Widget的方式快捷切换设置面板，还能顺便保存设置内容。同时三种工具由各自有一个设置类（以 `setting_pen` 为例），设置面板中的控件绑定对应类中的静态变量，如画笔的设置属性就保存在setting_pen的静态变量中。

```
//setting_pen的构造函数
setting_pen::setting_pen(QWidget *parent) :
        QWidget(parent), ui(new Ui::setting_pen) {
    ui->setupUi(this);

    ui->sizeSlider->valuefunc = [](int x){
        if(x <= 50) return x;
        else return int(pow(1.0589, x)) + 33;
    }; //size增长函数

    connect(ui->sizeSlider, &QSlider::valueChanged, [&](int val){size = ui-
>sizeSlider->realVal();});
    connect(ui->sparseSlider, &QSlider::valueChanged, [&](int val){sparse =
val;});
    connect(ui->colorWidget, &ColorSettingWidget::colorChanged, [&](const QColor
&col){
        color = col;
    }); //用匿名函数的方式绑定控件的修改和值的修改
}
```

工具设置面板中均使用这种匿名函数的方式绑定控件修改和值修改，能节省代码，避免出现大量冗杂无意义的函数。

同时本项目中还有大量其他地方的小绑定使用了匿名函数，略去不谈。

其中用到的是自定义滑动条，它允许设置一个值映射函数，具体见**MySlider**

在使用不同工具绘制时，主要利用的是QPainter类提供的多种**复合模式**，具体参考https://doc.qt.io/qt-5/qpainter.html#composition-modes

- SourceOver模式下，绘制的内容叠加在原内容之上，可以实现笔刷功能
- Clear模式下，绘制区域清除，与原内容和绘制内容均无关，可实现最简单的橡皮
- 我希望给橡皮也加入一个强度属性，可以用到DestinationOut模式，它令原内容减去绘制内容。

使用颜料桶即以点击到的像素为中心广搜，相邻像素颜色差值（我采取 $\sqrt{d_r^2+d_g^2+d_b^2+d_a^2}$ 作为颜色的差值，d_r 表示 r 分量上的差值）在容差范围内的都允许扩展。同时颜料桶也是先绘制到一个缓存区中，再一并叠加到原像素上，实现带透明度的颜色填涂。

其他设计

MySlider

```
class MySlider: public QSlider {
    Q_OBJECT
    QLabel *hint;
public:
    int (*valuefunc)(int); //值映射函数

    MySlider(QWidget *parent = nullptr);
    int realVal(); //应用值映射函数的真实值
protected:
    void mousePressEvent(QMouseEvent *event); //几种事件处理，一方面优化Slider本身的补足
    void mouseMoveEvent(QMouseEvent *event);  //另一方面制作了一个值显示功能
    void mouseReleaseEvent(QMouseEvent *event);
};
```

鉴于Qt自带的Slider功能十分不完善，难以满足需求，我在其基础上制作了一个自己的Slider类

Qt的slider主要缺点有几个：

- 在鼠标单击某个位置时，只能让滑块按照“步长”逐步移动，不能设置直接跳到目标位置。
- 不能在面板上显示数值
- 值域只能是均匀的，不能实现文章开头提到的ps中“非均匀分布”的尺寸条

第一点可以简单通过重载鼠标事件完成。

第二点，也是通过重载鼠标事件，鼠标按下时在滑块上方添加一个label以显示数值，且在鼠标移动事件中持续随滑块移动。

第三点：Photoshop中的画笔尺寸也是通过滑条修改的，但它的滑条前半段值变化慢，后半段值变化快，**做到了在小尺寸下精细，范围也足够大。**MySlider的主要目的即模仿这一优秀特性。

它保存一个函数指针 `int func(int)` 作为QSlider的value到实际值的映射函数，默认为 $f_x=x$。

仔细看了ps的滑动条之后，发现它是一个分段函数，前一般的滑动条都是均匀的，后半段值域激增，于是最终设计函数如下：

$$f_x = \begin{cases} x & x < 50 \\ 1.0589^x + 33 & x \geq 50 \end{cases}$$

x 为滑块原始值，范围在$[1, 120]$，最终 f_x 值域到约993

1.0589这个值的来源是：在50的分界点处，两段函数能够平滑连接（值相等且一阶导相等）

随后在适当扩大 x 的上界，使值域在3位数内尽可能大（不到四位数是为了布局好看）

ColorSettingWidget

```
class ColorSettingWidget: public QWidget{
    Q_OBJECT
    QLabel *back;    //网格表现的透明背景
    QPushButton *button; //用于触发的按钮
    QColor color;    //当前颜色
public:
    ColorSettingWidget(QWidget *parent = nullptr);
    ~ColorSettingWidget();

    void resizeEvent(QResizeEvent *ev);  //在Widget大小改变时，带动Label和Button一起改变大小

signals:
    void colorChanged(const QColor &col); //颜色改变时发出信号

public slots:
    void buttonClicked();

};
```

此处用到了一个外部的类**ColorPickerDialog**，功能和Qt自带的颜色对话框相同，但更加美观。

这个ColorSettingWidget即设置面板中的颜色设置项，它包含一个Label作为底色背景，一个Button用于交互，点击时弹出颜色对话框，且能在用户在颜色对话框中调整时，这个设置项的颜色就随之变换，加强了交互的实时性。

newCanvasDialog

```
class newCanvasDialog : public QDialog{
Q_OBJECT

    float lockedRate; //锁定宽高比，值 = 宽/高
    int width, height; //宽高
public:

    newCanvasDialog(QWidget *parent = nullptr); //构造函数中绑定了匿名函数，检测输入是否合法
    ~newCanvasDialog() override;

    void updateText(); //更新width，height到编辑框
```

```
signals:
    void sizeConfirmed(QSize size); //成功编辑后，触发此信号传递size

private:
    Ui::newCanvasDialog *ui;
};
```

点击创建画布时，弹出的小对话框，它比较简单，仅包含两个输入框（宽，高）和一个锁按钮，可以锁定宽高比。

对两个输入框的输入，我在槽函数中做了严格限制，仅允许输入数字，且在不允许输入过大数据。

总结

此份项目本打算花几天做一个简单的绘图工具即可，但在基本实现功能后，却发现用的很难受，许多地方使用直白的逻辑并不能符合用户需要（如简介所提及的种种ps优秀特性），后续在这些小细节优化上下了很大功夫，如滑动条，缩放显示，创建时自动调整等等，有些部分甚至为了一点细节重构了整个代码（如绘制透明像素的地方，为了防止重叠改写了整个绘制逻辑，加入了缓存区），最终耗费的时间远远超出了自己的预期。

比较遗憾的是，因为期末临近不得不停止继续肝这份东西，图像移动、选框操作这些东西没能完成，历史记录功能也完成的比较草率，占用太大以至于我只能限制只存十条记录。

但总体而言搞这个东西的过程中还是学到了很多，除了学到最基本的桌面应用软件开发方法外，我也在此次项目中大量应用指针和面向对象思想，这是平日里写代码不会用到的，它们极大地锻炼了我驾驭稍大型项目的能力，收获很大。