



Qt 画图软件设计教程

前面学习了 Qt 2D 绘图的基本知识，现在我们将所学习的知识进行一下综合，设计一个简单的画图软件。因为前面已经有了记事本的例子，所以这里对一些细节知识将不会多讲。而且，这只是个例子，说是软件，其实只是个较大点的程序而已，并不是实际的项目，所以通过这个例子，我们主要为了掌握一个方法和两个知识点。

一个方法：写较复杂程序的方法。

两个知识点：文件菜单的功能实现和利用双缓冲进行绘图。

这个例子共分为三部分进行介绍：

第一部分：进行软件界面的设计。其中一些知识可以参考我博客中的 Qt 教程四。

第二部分：实现基本的功能，这里会再次详细介绍文件菜单中新建，打开，保存，另存为，关闭等菜单的功能实现。这里你可以参考我博客中的 Qt 教程六。

第三部分：我们在这里会详细讲解所谓的双缓冲绘图，利用它实现图形（例如矩形，椭圆）的交互式绘制。

第一部分：设计界面

在 windows 上自带了画图软件，在 Qt Creator 中也有个绘图软件的例子 scribble example，你可以在帮助中进行查看。而现在我们要写的绘图程序的例子，也应该包含这些功能，如绘画涂鸦，添加直线或矩形等常用图形，可以插入图片，在图片上进行绘画，可以让它缩放，旋转，拉伸等。想到了这些功能，我们就可以设计软件的界面了。

一．设计菜单

1.在 QtCreator 中新建 Qt4 Gui Application 工程，我这里使用的工程名为 paint，使用默认的 QMainWindow 作为主窗口。

2.为了让程序中可以使用中文，我们先在 main.cpp 中加入头文件#include <QTextCodec>，并加入下面一行代码：QTextCodec::setCodecForTr(QTextCodec::codecForLocale());

3.打开 mainwindow.ui，先设计菜单。依次是文件菜单，编辑菜单，工具菜单和帮助菜单。其内容分别如下：



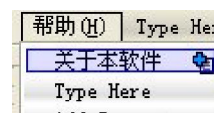
文件菜单



编辑菜单



工具菜单



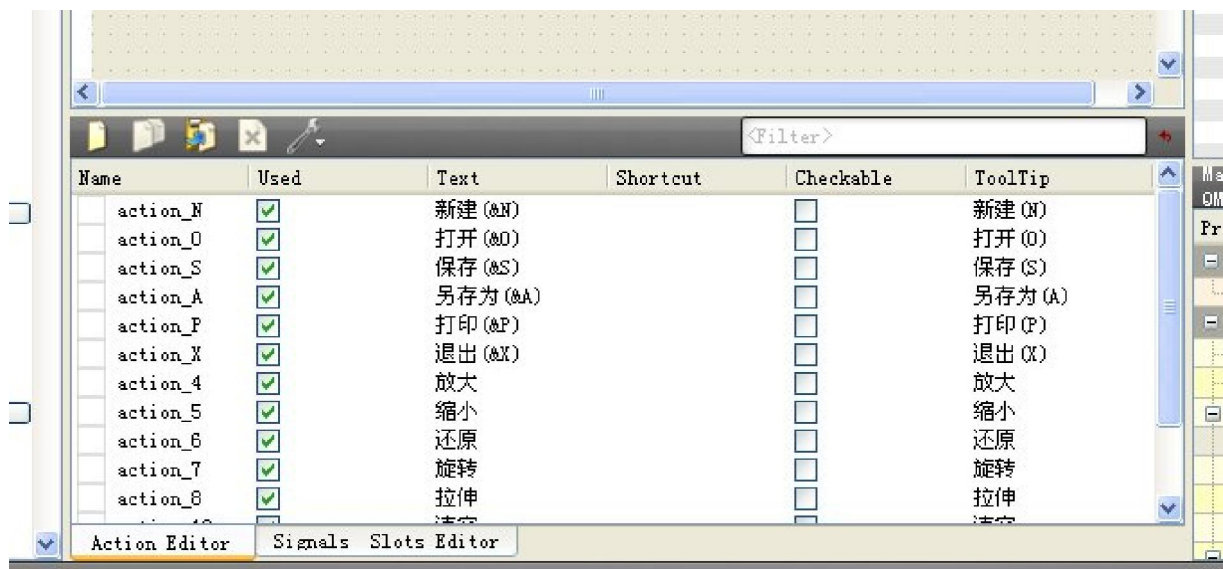
帮助菜单



4.向工程中添加资源文件，向其中添加要使用的菜单图标。

添加完后记着保存一下资源文件，不然的话，在资源管理器中可能看不到添加的图标。

5.打开动作编辑器 Action Editor，编辑已添加的菜单动作。

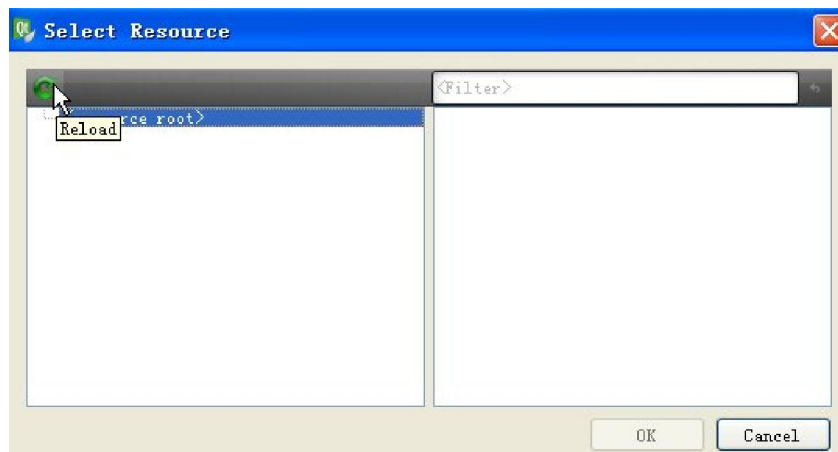


我们下面只介绍其中“新建”菜单的编辑，其余菜单照做就行了。

双击 action_N，弹出如下对话框：

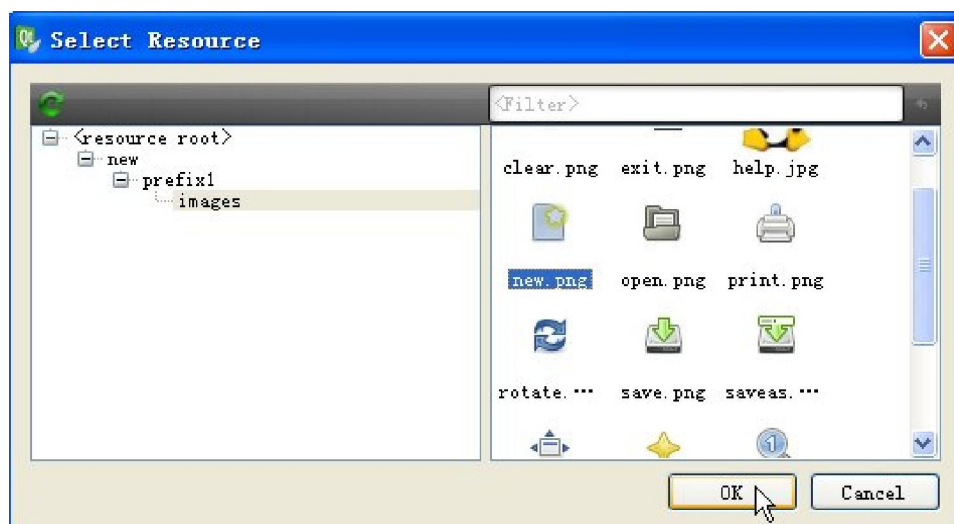


单击 Icon 后面的按钮，弹出资源管理器：





我们点击上面的 Reload，便能显示出我们已经添加的图标，我们选中其中的新建菜单的图标，点击 Ok 确认，如下图所示：



添加完图标后，我们单击一下快捷键 Shortcut 后面的输入框，然后同时按下 Ctrl 键和 N 键，这样就将新建菜单的快捷键设为了 Ctrl+N，如下图所示：



然后我们再在右面的属性窗口中更改 statusTip 的内容为“新建文件”，这样在鼠标放在新建菜单上是，状态栏就会显示“新建文件”。如下图：



我们按照同样的方法设置其它菜单，设置完成后如下图所示：



<Filter>					
Name	Used	Text	Shortcut	Checkable	ToolTip
 action_N	<input checked="" type="checkbox"/>	新建 (&N)	Ctrl+N	<input type="checkbox"/>	新建 (N)
 action_O	<input checked="" type="checkbox"/>	打开 (@O)	Ctrl+O	<input type="checkbox"/>	打开 (O)
 action_S	<input checked="" type="checkbox"/>	保存 (@S)	Ctrl+S	<input type="checkbox"/>	保存 (S)
 action_A	<input checked="" type="checkbox"/>	另存为 (@A)		<input type="checkbox"/>	另存为 (A)
 action_P	<input checked="" type="checkbox"/>	打印 (@P)	Ctrl+P	<input type="checkbox"/>	打印 (P)
 action_X	<input checked="" type="checkbox"/>	退出 (@X)		<input type="checkbox"/>	退出 (X)
 action_4	<input checked="" type="checkbox"/>	放大	Ctrl+Up	<input type="checkbox"/>	放大
 action_5	<input checked="" type="checkbox"/>	缩小	Ctrl+Down	<input type="checkbox"/>	缩小
 action_6	<input checked="" type="checkbox"/>	还原	Ctrl+Home	<input type="checkbox"/>	还原
 action_7	<input checked="" type="checkbox"/>	旋转	Ctrl+Left	<input type="checkbox"/>	旋转
 action_8	<input checked="" type="checkbox"/>	拉伸	Ctrl+Right	<input type="checkbox"/>	拉伸
 action_10	<input checked="" type="checkbox"/>	清空	Ctrl+Del	<input type="checkbox"/>	清空
 action_11	<input checked="" type="checkbox"/>	绘图工具栏		<input type="checkbox"/>	绘图工具栏
 action_12	<input checked="" type="checkbox"/>	关于本软件		<input type="checkbox"/>	关于本软件

6.我们把其中的一些图标放到工具栏上，如下图所示：



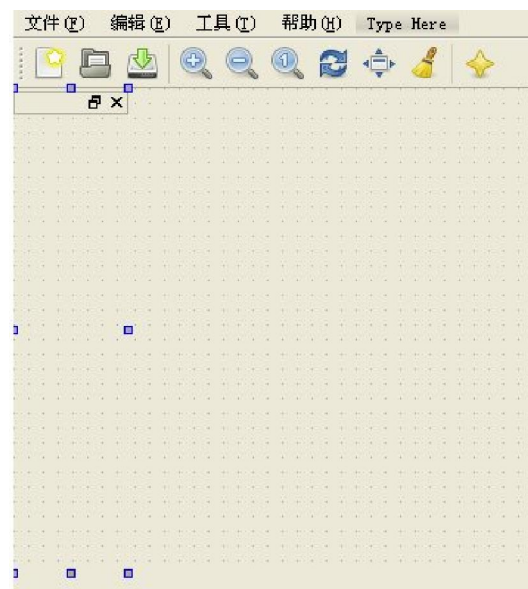
这样就完成了菜单的设计。

二．添加绘图工具栏

我们这里的绘图工具栏使用的是一种叫做 Dock 的窗口，它与其它窗口的不同就是它可以在其父窗口中浮动，也可以停靠在父窗口的边界，就像一个工具栏一样。

1.我们在左边的部件栏中找到 Dock Widget，将其拖入到设计区。


添加后它默认在左边框上停靠着。如下图：





2.我们在其属性栏里将其 windowTitle 改为 “ 画图工具 ”。

3.向其中拖入相关部件，效果如下图：



其中，“画笔线宽”下的部件为 Spin Box，其属性中的 objectName 为 penWidthSpinBox，属性栏最下面的 minimum 属性改为 1，即最小值为 1。其余部件均为组合框 ComboBox，objectName 依次为：

选择图形：shapeComboBox 画笔类型：penStyleComboBox
 画笔颜色：penColorComboBox 填充颜色：brushColorComboBox

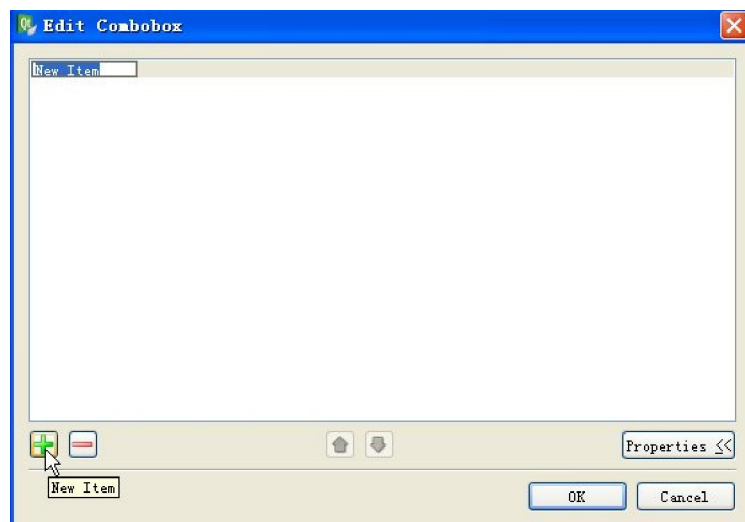
dockWidget	QDockWidget
dockWidgetContents	QWidget
brushColorComboBox	QComboBox
label	QLabel
label_2	QLabel
label_3	QLabel
label_4	QLabel
label_5	QLabel
penColorComboBox	QComboBox
penStyleComboBox	QComboBox
penWidthSpinBox	QSpinBox
shapeComboBox	QComboBox

4.我们给选择框添加条目。

右击“选择图形”下面的组合框，弹出菜单，如下：

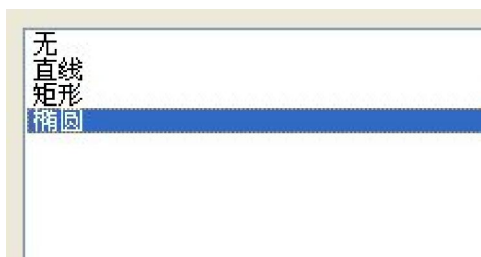


我们点击 Edit Items 菜单，弹出下面的条目编辑框，我们点击 “+” 按钮，添加新的条目：





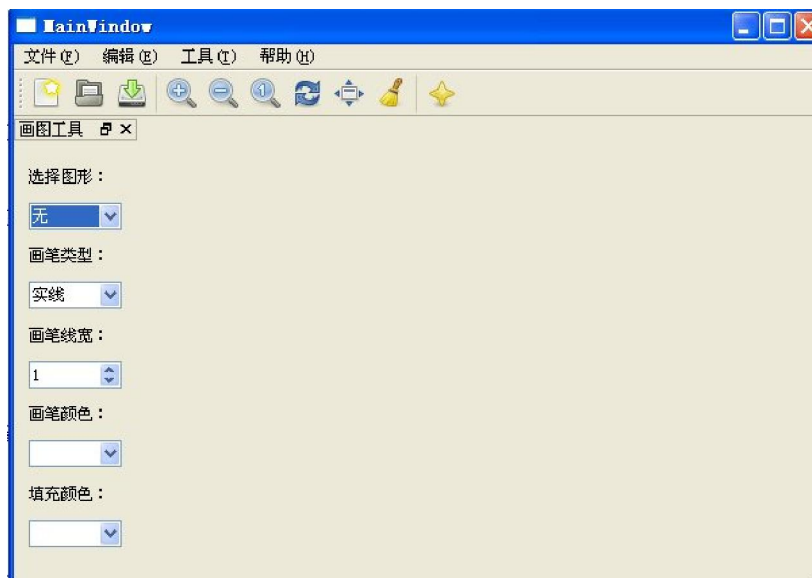
我们依次添加“无”“直线”“矩形”“椭圆”四个条目，如下所示：



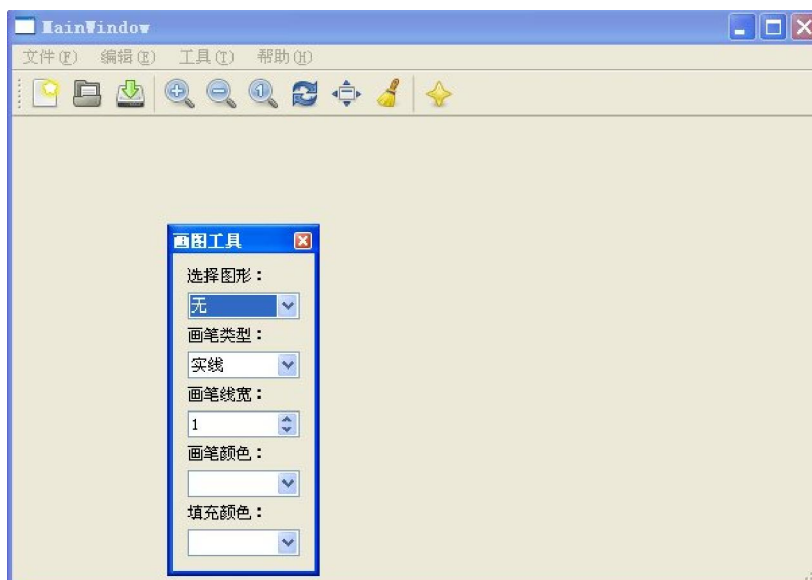
同样的我们给“画笔类型”下的组合框添加两个条目“实线”和“点线”。两个颜色组合框的条目以后再添加。

5.最后可以让所有部件处于一个网格布局管理器中。

此时运行程序，效果如下：



拖动画图工具栏后效果如下：

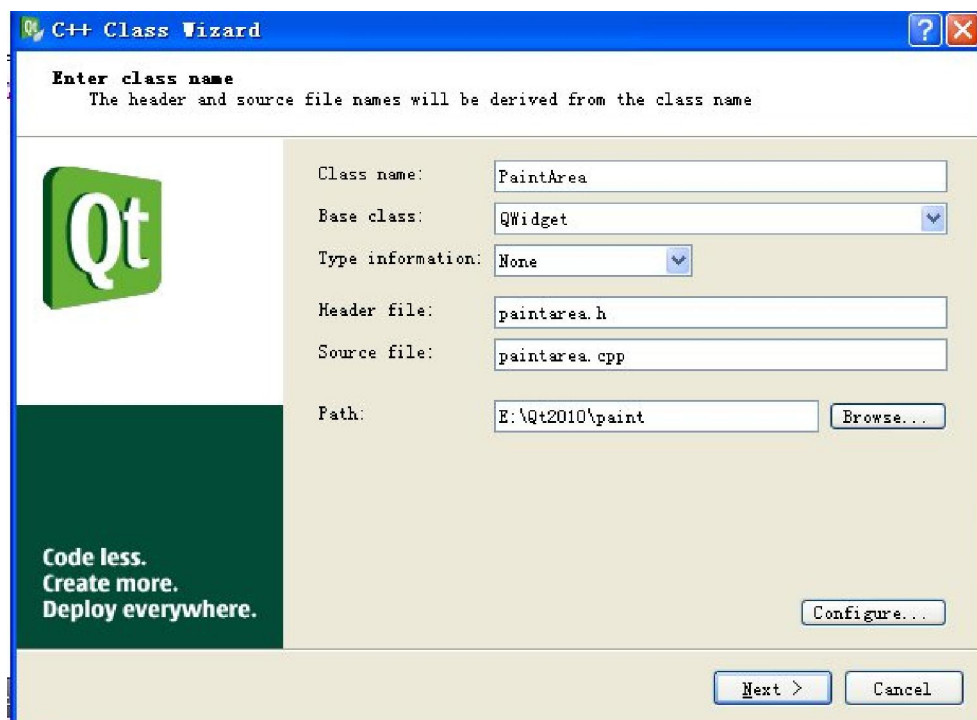
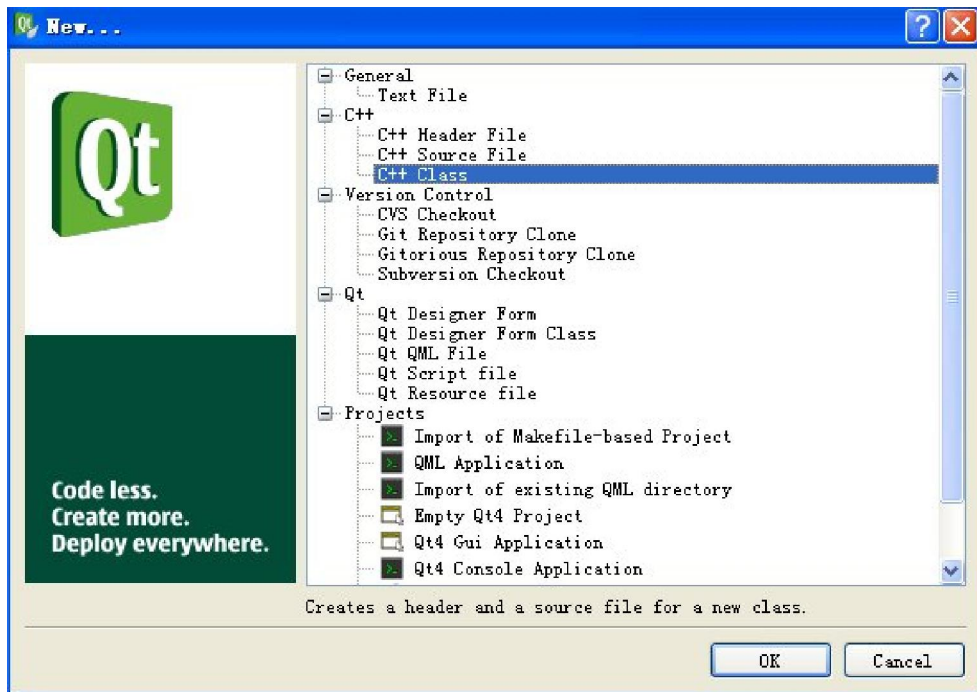




三．添加画布

因为画布是真正实现绘图功能的，所以我们新建一个类来实现所有跟绘图有关的功能。这里先进行操作，对于一些内容到后面我们会详细解释的。

1.往工程中添加新的 C++类，类名为 PaintArea，以 QWidget 作为基类，如下所示：



2.在 paintArea.h 中声明对象和函数。



```
class PaintArea : public QWidget
{
public:
    PaintArea();
protected:
    void paintEvent(QPaintEvent *);    //重绘事件
private:
    QImage image;    //QImage 类对象，用于在其上绘图
    QRgb backColor; //QRgb 颜色对象，存储 image 的背景色
};
```

我们这里使用了 QImage 类对象进行绘图，其实使用以前讲过的 QPixmap 类对象也是可以的。

3.在 paintarea.cpp 中的构造函数里初始化对象。

先加入头文件声明：`#include <QPainter>`

再更改构造函数：

```
PaintArea::PaintArea()
{
    image = QImage(400,300,QImage::Format_RGB32);    //画布的初始化大小设为
400*300，使用 32 位颜色
    backColor = qRgb(255,255,255);    //画布初始化背景色使用白色
    image.fill(backColor);
}
```

4.在 paintarea.cpp 中定义重绘函数。

```
void PaintArea::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.drawImage(0,0,image);
}
```

5.将画布添加到主界面的中心区。

因为以后可能要使用很大的画布，这样为了使画布很大时还能显示，我们需要加入滚动条，所以这里使用了 QScrollArea 类对象，它提供了横向和纵向的滚动条，如果你有兴趣，可以在帮助中查看它的介绍。

首先，在 mainwindow.h 文件中进行如下操作：

添加头文件声明：

```
#include "paintarea.h"
#include <QScrollArea>
```



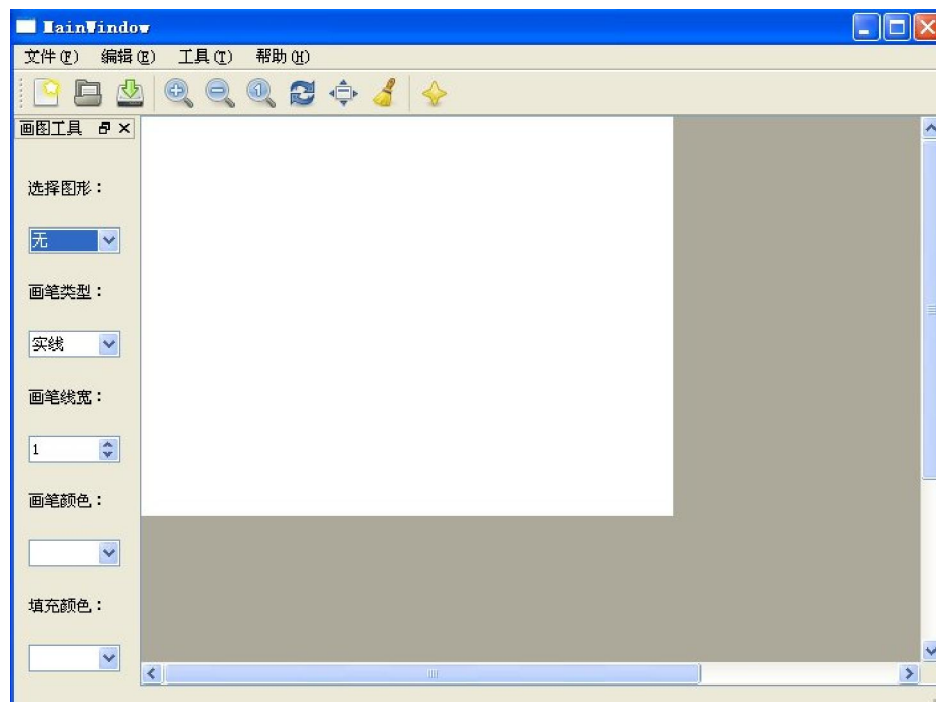

在下面的 private 中添加对象的声明：

```
PaintArea *area;  
QScrollArea *scrollArea;
```

然后在 mainwindow.cpp 文件中的构造函数里添加代码：

```
MainWindow::MainWindow(QWidget *parent) :  
    QMainWindow(parent),  
    ui(new Ui::MainWindow)  
{  
    ui->setupUi(this);  
    resize(700,500); //主窗口大小设为 700*500  
    area = new PaintArea;  
    scrollArea = new QScrollArea;  
    scrollArea->setBackgroundRole(QPalette::Dark); //scrollArea 对象的背景色设为 Dark  
    scrollArea->setWidget(area); //将画布添加到 scrollArea 中  
    scrollArea->widget()->setMinimumSize(800,600); //scrollArea 初始化大小设为  
800*600  
  
    setCentralWidget(scrollArea); //将 scrollArea 加入到主窗口的中心区  
}
```

这时运行程序，效果如下：



四．实现涂鸦的功能

我们在画布类中添加一些代码，实现最基本的涂鸦功能。



1.在 paintarea.h 中做更改。

添加头文件：

```
#include <QMouseEvent>
#include <QPoint>
```

在 protected 中添加函数声明：

```
void mousePressEvent(QMouseEvent *); //鼠标按下事件
void mouseMoveEvent(QMouseEvent *); //鼠标移动事件
void mouseReleaseEvent(QMouseEvent *); //鼠标释放事件
```

在 private 中添加对象声明：

```
QPoint lastPoint,endPoint; //定义两个坐标对象存放鼠标指针的前后两个坐标
```

2.在 paintarea.cpp 中添加函数的定义。

```
void PaintArea::mousePressEvent(QMouseEvent *event)
{
    if(event->button() == Qt::LeftButton) //当鼠标左键按下
    {
        lastPoint = event->pos(); //获得鼠标指针的当前坐标作为起始坐标
    }
}

void PaintArea::mouseMoveEvent(QMouseEvent *event)
{
    if(event->buttons() & Qt::LeftButton) //如果鼠标左键按着的同时移动鼠标
    {
        endPoint = event->pos(); //获得鼠标指针的当前坐标作为终止坐标
        paint(image); //绘制图形
    }
}

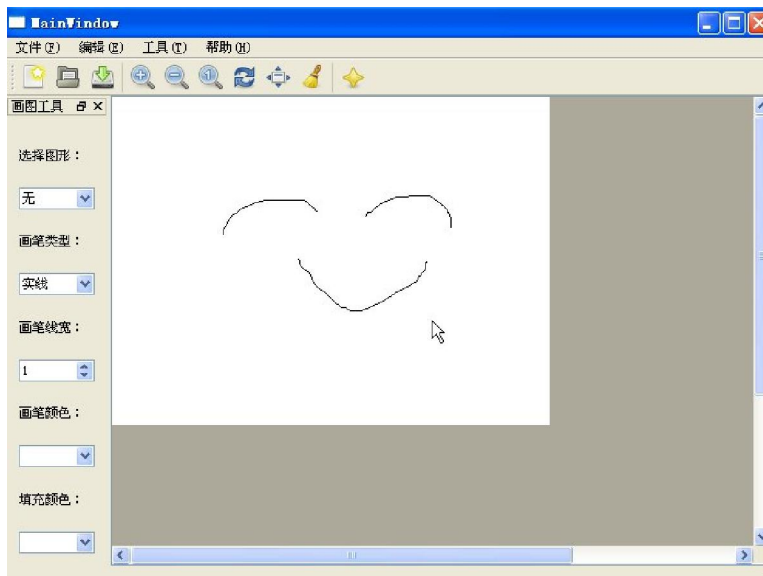
void PaintArea::mouseReleaseEvent(QMouseEvent *event)
{
    if(event->button() == Qt::LeftButton) //如果鼠标左键释放
    {
        endPoint = event->pos();
        paint(image);
    }
}

void PaintArea::paint(QImage &theImage)
{
    QPainter pp(&theImage); //在 theImage 上绘图
    pp.drawLine(lastPoint,endPoint); //由起始坐标和终止坐标绘制直线
}
```



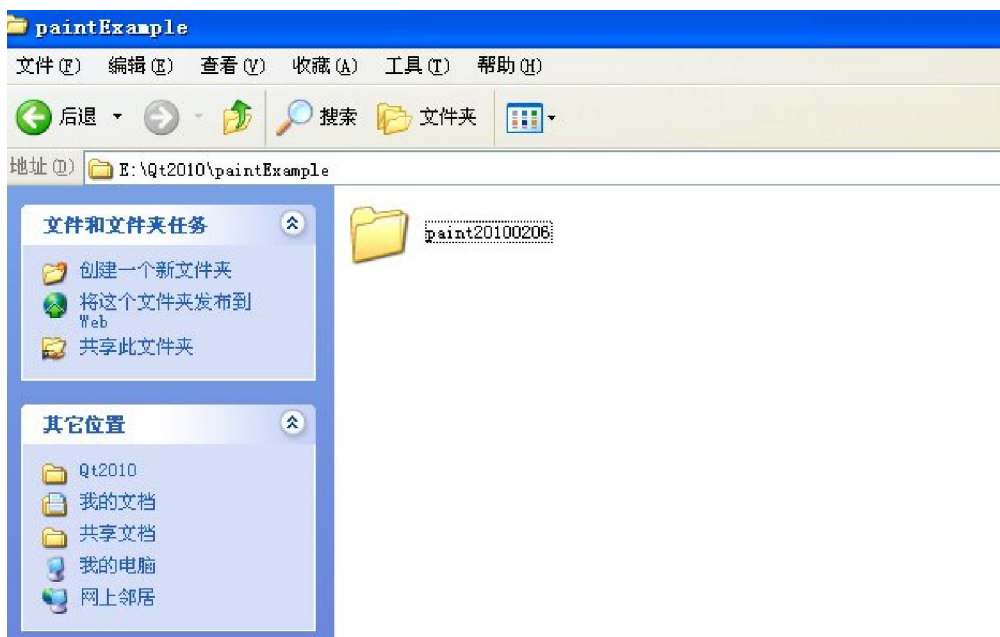
```
lastPoint = endPoint; //让终止坐标变为起始坐标
update(); //进行更新界面显示，可引起窗口重绘事件，重绘窗口
}
```

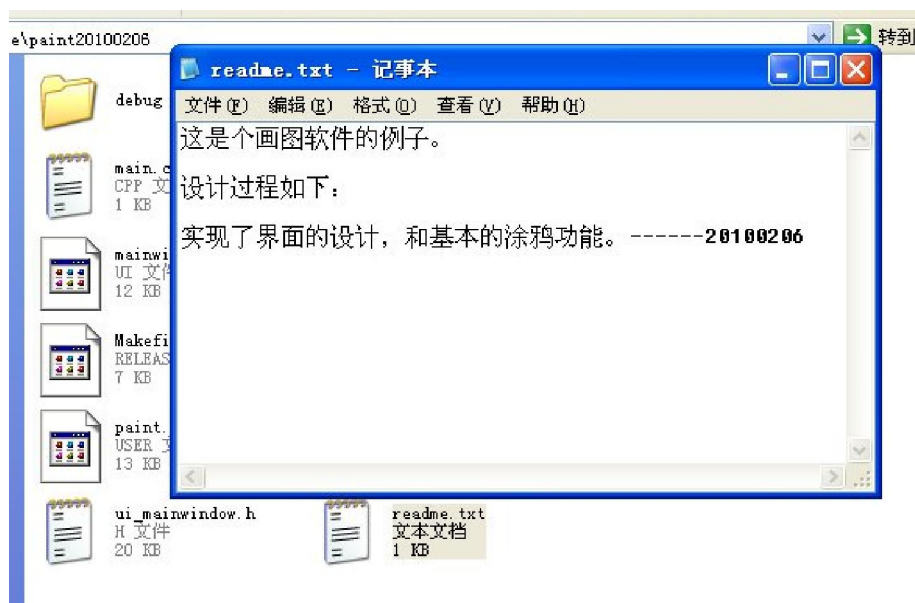
这样，再次运行程序就能实现涂鸦的功能了，效果如下：



在下一讲中我们将实现所有菜单的功能。

在退出软件之后，我们可以先将工程文件夹进行备份，比如这里命名为 paint20100206，后面就是该工程修改的日期。我们也可以在工程文件夹中新建文本文档，命名为 readme.txt，用来记录该备份对程序进行了哪些修改。





第二部分：实现基本功能

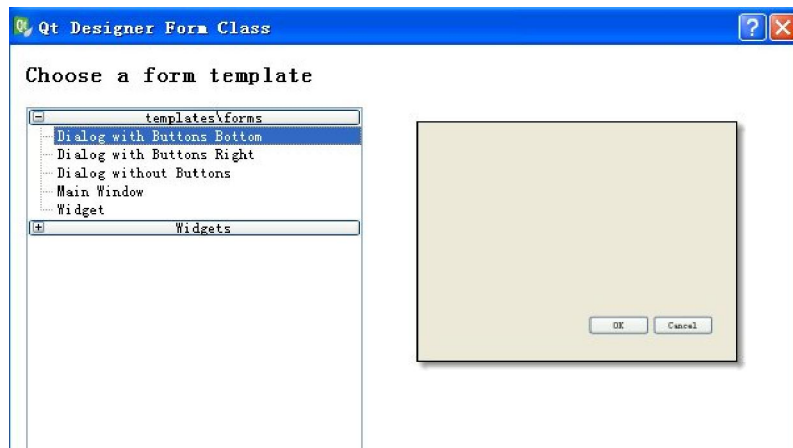
上一节我们已经将界面做好了，在这一节中我们将实现所有菜单的功能。因为文件菜单是很多软件都有的，所以我们一定要搞清楚其中几个菜单之间的逻辑关系，虽然在以前的记事本程序中已经对其进行了讲解，但是我感觉当时讲的还是不够清楚，所以这次再对其进行详细讲解，希望大家务必掌握几个函数的写法。

我们用 Qt Creator 打开上次的工程，即打开工程文件夹中的 paint.pro 文件。

一．添加设置画布的对话框

首先，因为新建画布时，我们想要可以设置画布的大小和背景颜色，所以在开始讲解之前，我们先添加一个对话框，用来让用户进行相关设置。

1.我们添加新的 Qt Designer Form Class，选择带有按钮的对话框作为模板，如下图所示。然后类名使用 DoNewDialog。



2.我们将其界面设计如下。



其中“宽”后的 Spin Box，将其 object Name 修改为 widthSpinBox，修改其 minimum 为 400，maximum 为 10000；修改“高”后的 Spin Box，将其 object Name 改为 heightSpinBox，修改其 minimum 为 300，maximum 为 10000；“背景颜色”后的部件是一个 Tool Button，点击它显示颜色对话框，和一个 Text Browser，用于显示颜色对话框返回的颜色。

3. 我们在 donewdialog.h 文件中声明函数和对象。

在 public 中添加：

```
int getWidth();    //用于返回画布的宽
int getHeight();   //用于返回画布的高
QColor getBackColor(); //用于返回画布的背景色
```

在 private 中添加：

```
QColor backColor; //用于保存画布的背景色
```

4. 然后在 donewdialog.cpp 中进行更改。

先在构造函数中对 backColor 进行初始化：

```
backColor = Qt::white;
```

然后进行函数的定义。

```
int DoNewDialog::getWidth()
{
    return ui->widthSpinBox->text().toInt();
}

int DoNewDialog::getHeight()
{
    return ui->heightSpinBox->text().toInt();
}
```



```

QColor DoNewDialog::getBackColor()
{
    return backColor;
}

```

5. 在设计器界面上右击那个 Tool Button，进入其单击事件槽函数。修改如下。

```

void DoNewDialog::on_toolButton_clicked()
{
    QColor newColor = QColorDialog::getColor(); //从颜色对话框获得颜色
    if (newColor.isValid()) //如果得到的是可用的颜色
    {
        backColor = newColor;

        QPalette palette = ui->textBrowser->palette(); //显示这个颜色
        palette.setColor(QPalette::Base, backColor);
        ui->textBrowser->setPalette(palette);
        update();
    }
}

```

我们需要加入 `#include <QColorDialog>` 的头文件。这里我们将获得的颜色在那个 TextBrowser 上显示了出来。其中 palette 对象获取了 textBrowser 的调色板，这样来更改它的显示颜色。

然后，我们需要更改 PaintArea 类的内容，让其画布可以更改大小和颜色。

1. 在 paintarea.h 中的 public 中添加函数声明。

```

void setImageSize(int width, int height);
void setImageColor(QColor color);

```

2. 在 paintarea.cpp 中进行函数的定义。

```

void PaintArea::setImageSize(int width, int height)
{
    QImage newImage(width, height, QImage::Format_RGB32);
    image = newImage;
    update();
}

void PaintArea::setImageColor(QColor color)
{
    backColor = color.rgb(); //因为 image 的背景色要用 QRgb 类型的颜色，所以这里进行
    了一下转换
    image.fill(backColor);
    update();
}

```




最后我们在主界面中进入新建菜单的 triggered() 事件，如下图所示。



然后在 mainwindow.h 中对其内容更改如下：

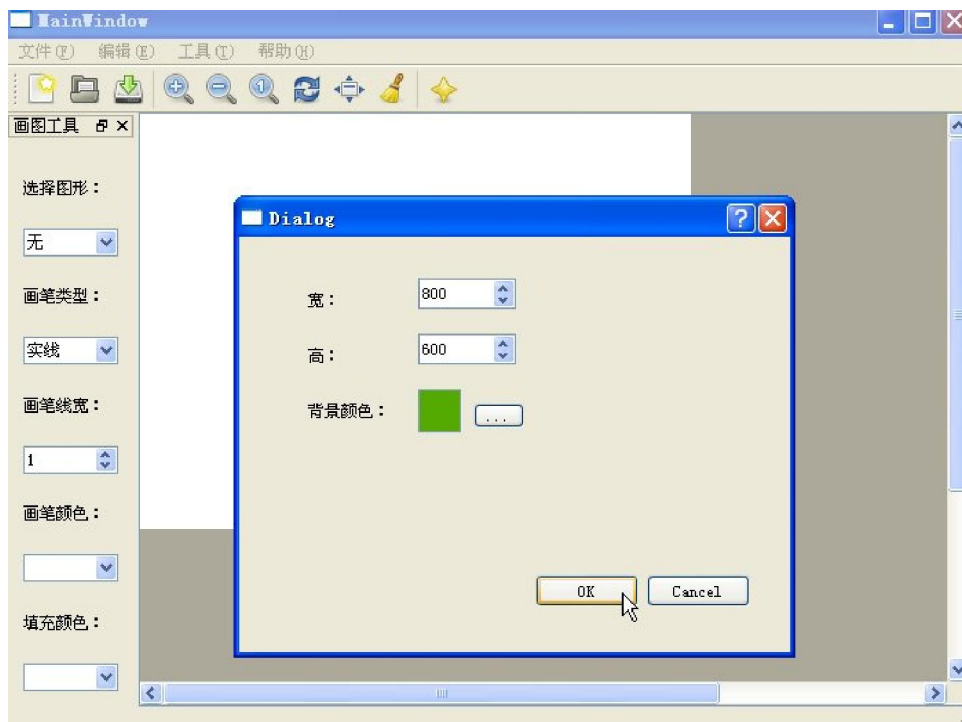
```
void MainWindow::on_action_N_triggered() //新建菜单
```

```
{
    DoNewDialog dlg;
    if(dlg.exec() == QDialog::Accepted)
    {
        int width = dlg.getWidth();
        int height = dlg.getHeight();
        area->setImageSize(width,height); //更改画布大小
        scrollArea->widget()->resize(width,height); //更改 ScrollArea 的大小
        area->setImageColor(dlg.getBackColor());
    }
}
```

并添加头文件：`#include "donewdialog.h"`

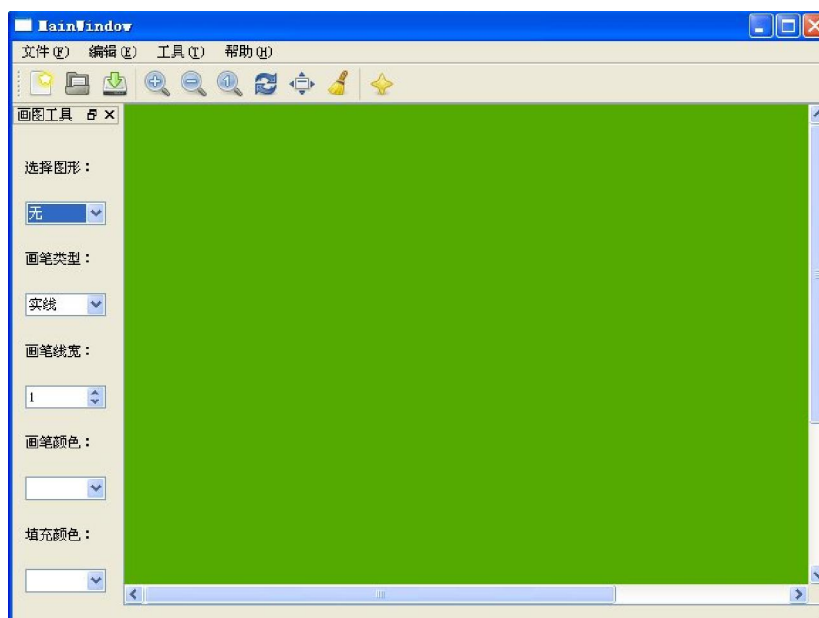
此时运行程序，效果如下。

点击新建按钮并设置画布属性：





新的画布：



二．完成文件菜单的功能

文件菜单中一般都包含新建，打开，保存，另存为和关闭或退出操作。这几个菜单之间存在着逻辑上的顺序关系，因为它们很常用，所以在这里我们再次详细地介绍一下这几个菜单的实现方法，希望能帮助大家很好地掌握。

其中利用到几个函数的原型如下：

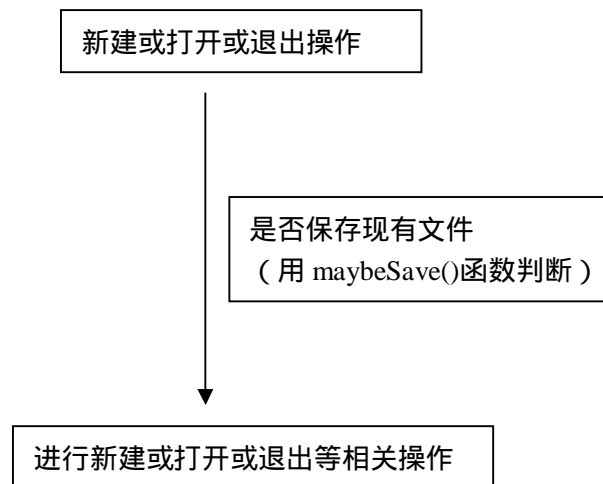
bool maybeSave()：进行是否保存的判断，当返回值为 true 时说明整个判断过程已经进行过了。

bool isModified()：进行文件是否被更改过的判断，其实就是返回一个 bool 型的变量 modified 的值。

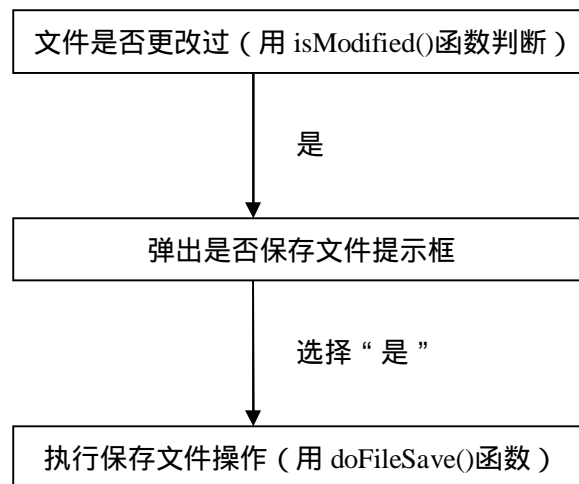
bool doFileSave()：进行文件保存操作，文件保存时先利用 bool 型变量 isSaved 判断文件是否保存过，如果保存过，就直接将现在的文件进行存储就行了；如果没有保存过，就要进行另存为操作，这样才能获得文件的保存路径。

bool saveFile(QString fileName)：以文件的路径对文件进行存储。存储成功返回 true。

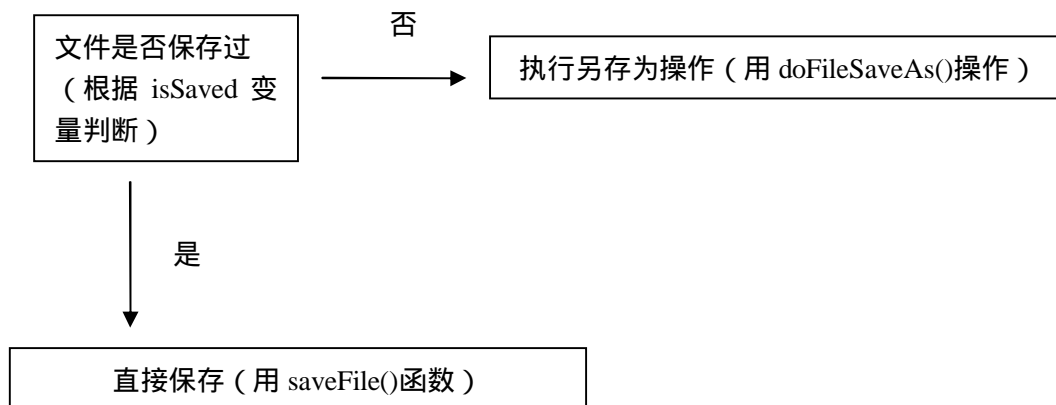
整个流程图如下：



maybeSave()函数内容：



doFileSave()函数内容：



1.在画布类中添加函数，实现具体的文件打开和保存操作。

在 paintarea.h 中的 public 中添加函数声明：



```
bool isModified() const { return modified; } //判断画布内容是否被更改过
bool saveImage(const QString &fileName, const char *fileFormat); //保存图片
bool openImage(const QString &fileName); //打开图片
```

在 private 中声明变量：

```
bool modified; //标志画布是否被更改过
```

在 paintarea.cpp 中对函数进行定义。

```
bool PaintArea::saveImage(const QString &fileName, const char *fileFormat)
{
    QImage visibleImage = image;

    if (visibleImage.save(fileName, fileFormat)) //实现了文件存储
    {
        modified = false;
        return true;
    }
    else
    {
        return false;
    }
}

bool PaintArea::openImage(const QString &fileName)
{
    QImage loadedImage;
    if (!loadedImage.load(fileName))
        return false;

    QSize newSize = loadedImage.size();
    setImageSize(newSize.width(), newSize.height());
    image = loadedImage;

    modified = false;
    update();
    return true;
}
```

在 paintarea.cpp 中相关位置设置 modified 变量。

在构造函数里添加：`modified = false;`

在 paint()函数里添加：`modified = true;`

这样就表示只有执行了 paint() 函数，才表明画布内容被更改过。



2.在主窗口类中实现文件菜单的功能。

在 mainwindow.h 中进行函数和变量的声明。

在 public 中声明函数：

```
void doNew();    //新建文件操作
void doOpen();   //打开文件操作
bool doFileSave(); //保存文件操作
bool doFileSaveAs(); //文件另存为操作
```

在 private 中声明变量和函数：

```
bool isSaved;    //标志文件是否保存过
QString curFile; //保存当前文件的路径
bool maybeSave(); //是否保存文件
```

```
bool saveFile(QString fileName); //实现文件的存储
```

在 mainwindow.cpp 中进行更改。

先添加头文件：

```
#include <QMessageBox>
#include <QFileDialog>
```

再在构造函数里进行变量初始化：

```
isSaved = false;
curFile = tr("未命名.png");
```

然后进行那几个函数的定义：

```
void MainWindow::doOpen()
{
    if (maybeSave())
    {
        QString fileName = QFileDialog::getOpenFileName(this,
                                                         tr("打开文件"), QDir::currentPath());

        if (!fileName.isEmpty())
        {
            area->openImage(fileName);
            scrollArea->widget()->resize(area->getImageSize());
            //获得图片的大小，然后更改 scrollArea 的大小
            isSaved = true;
            curFile = fileName;
        }
    }
}

void MainWindow::doNew()
{
    if(maybeSave())
```



```
{
    DoNewDialog dlg;
    if(dlg.exec() == QDialog::Accepted)
    {
        int width = dlg.getWidth();
        int height = dlg.getHeight();
        area->setImageSize(width,height);
        scrollArea->widget()->resize(width,height);
        area->setImageColor(dlg.getBackColor());
        isSaved = false;
    }
}

bool MainWindow::maybeSave()
{
    if(area->isModified())
    {
        QMessageBox::StandardButton box;
        box = QMessageBox::warning(this,tr("保存文件"),tr("图片已经改变，是否保存？"),
        QMessageBox::Yes|QMessageBox::No|QMessageBox::Cancel);
        if(box == QMessageBox::Yes)
        {
            return doFileSave();
        }
        else if(box == QMessageBox::Cancel)
        {
            return false;
        }
    }
    return true;
}

bool MainWindow::doFileSave()
{
    if(isSaved)
    {
        return saveFile(curFile);
    }
    else return doFileSaveAs();
}

bool MainWindow::saveFile(QString fileName)
```




```
{
    if(area->saveImage(fileName,"png"))
    {
        isSaved = true;
        return true;
    }
    else return false;
}

bool MainWindow::doFileSaveAs()
{
    QString fileName = QFileDialog::getSaveFileName(this,tr("另存为"),curFile);
    if(fileName.isEmpty())
    {
        return false;
    }
    else return saveFile(fileName);
}
```

这里的 doOpen()函数中用到了 area->getImageSize()，所以我们还要在画布类里添加该函数。

在 paintarea.h 中的 public 中添加函数的声明：`QSize getImageSize();`

在 paintarea.cpp 中添加其定义：

```
QSize PaintArea::getImageSize()
{
    return image.size();
}
```

3.编写菜单的 triggered 事件函数。

我们更改“新建”菜单的槽函数的内容：

```
void MainWindow::on_action_N_triggered() //新建菜单
{
    doNew();
}
```

然后按照进入“新建”菜单的方法，我们从 ui 文件依次进入其他菜单的槽函数，更改如下：

```
void MainWindow::on_action_O_triggered() //打开菜单
{
    doOpen();
}
```

```
void MainWindow::on_action_S_triggered() //保存菜单
{

```



```

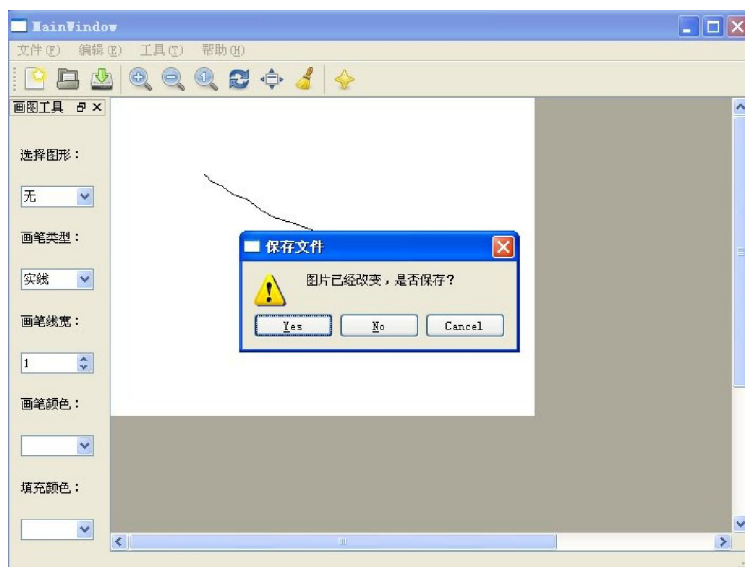
doFileSave();
}

void MainWindow::on_action_A_triggered() //另存为菜单
{
    doFileSaveAs();
}

void MainWindow::on_action_X_triggered() //退出菜单
{
    if(maybeSave())
        qApp->quit();
}

```

这时运行程序，应用这几个菜单进行各种相关操作。
是否保存文件提示如下：



文件另存为效果如下：

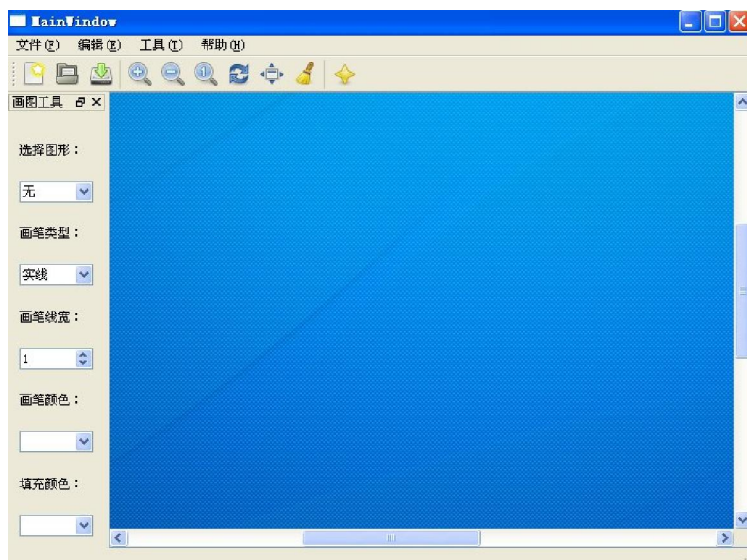




打开文件：



打开文件后效果如下：



为了更完善,我们在点击窗口右上角的关闭按钮时,也应该出现和点击退出菜单一样的效果。

所以下面我们重写窗口的关闭事件函数。

在 mainwindow.h 中的 protected 中声明函数：`void closeEvent(QCloseEvent *)`;

然后在 mainwindow.cpp 中定义该函数：

```
void MainWindow::closeEvent(QCloseEvent *event)    //关闭事件
{
    if(maybeSave())
        qApp->quit();
    else event->ignore();
}
```



这时你可以运行程序，测试一下效果。

4.实现打印菜单的功能。

在 paintarea.h 中 public 中添加函数声明：`void doPrint();`

在 paintarea.cpp 中添加头文件：

```
#include <QPrintDialog>
#include <QPrinter>
```

然后定义该函数：

```
void PaintArea::doPrint()
{
    QPrinter printer(QPrinter::HighResolution);

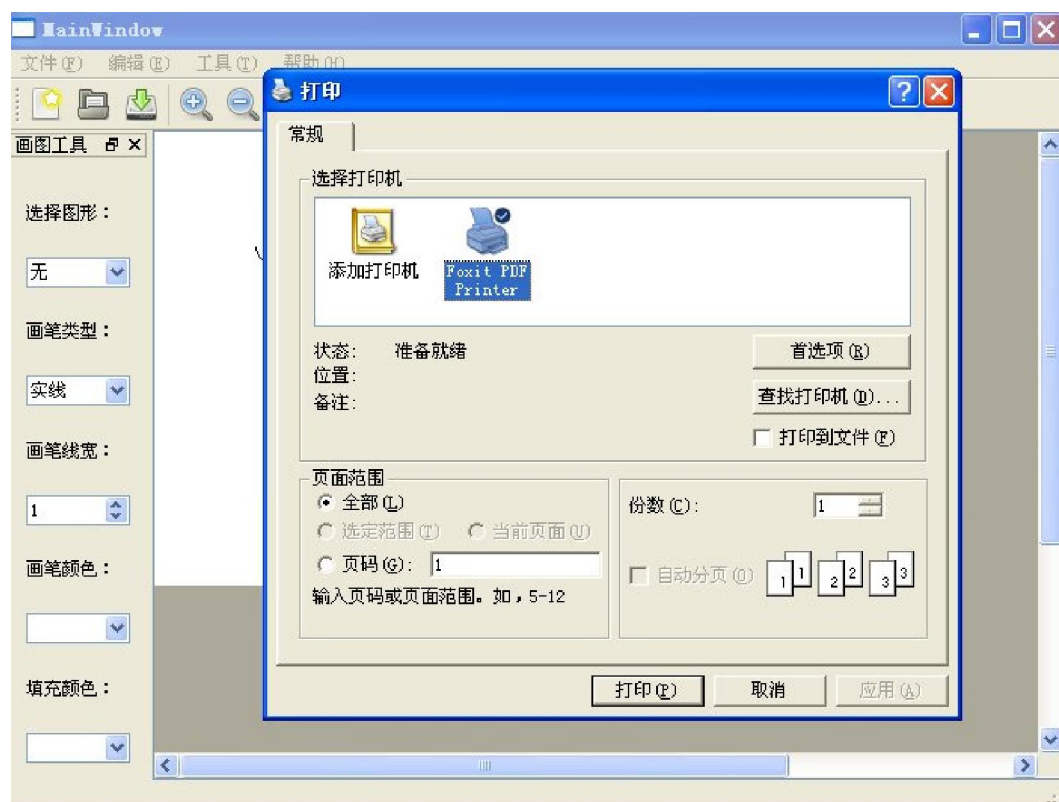
    QPrintDialog *printDialog = new QPrintDialog(&printer, this);
    if (printDialog->exec() == QDialog::Accepted)
    {
        QPainter painter(&printer);
        QRect rect = painter.viewport();
        QSize size = image.size();
        size.scale(rect.size(), Qt::KeepAspectRatio);
        painter.setViewport(rect.x(), rect.y(), size.width(), size.height());
        painter.setWindow(image.rect());
        painter.drawImage(0, 0, image);
    }
}
```

然后从 mainwindow.ui 中的动作编辑器中进入“打印”菜单的 triggered 事件槽函数：

```
void MainWindow::on_action_P_triggered()    //打印菜单
{
    area->doPrint();
}
```

这样就实现了打印功能。对于上面函数里的代码我们不做过多解释，如果有兴趣可以查看与打印相关的类的帮助。

我们运行程序，查看其效果：



如果你装有打印机，或装有 PDF 制作软件，就可以打印文件了。因为我这里装有 Foxit PDF Creator 软件，所以有一个 Foxit PDF Printer，利用它可以将文件打印成 PDF 文档。

这时我们可以对工程文件夹再次进行备份，例如我这里命名为“paint20100207”。

三．完成文件编辑菜单的功能

编辑菜单主要完成图片的一些形状变化操作，所以我们需要在画布类里添加一些函数和变量来实现这些功能。

1.在 paintarea.h 中添加代码。

在 public 中添加函数声明：

```
void zoomIn();    //放大
void zoomOut();   //缩小
void zoom_1();    //还原
void doRotate();  //旋转
void doShear();   //拉伸
void doClear();   //清空
```

在 private 中添加变量声明：

```
qreal scale;      //缩放量
int angle;        //角度
qreal shear;      //拉伸量
```



2.在 paintarea.cpp 中进行更改。

在构造函数里进行变量初始：

```
scale = 1;  
angle = 0;  
shear = 0;
```

然后进行那几个函数的定义：

```
void PaintArea::zoomIn()  
{  
    scale*=1.2;  
    update();  
}
```

```
void PaintArea::zoomOut()  
{  
    scale/=1.2;  
    update();  
}
```

```
void PaintArea::zoom_1()  
{  
    scale = 1;  
    update();  
}
```

```
void PaintArea::doRotate()  
{  
    angle +=90;  
    update();  
}
```

```
void PaintArea::doShear()  
{  
    shear = 0.2;  
    update();  
}
```

```
void PaintArea::doClear()  
{  
    image.fill(backColor);    //用现在的画布背景色进行填充  
    update();  
}
```




下面我们更改重绘事件函数，实现相关的效果。

```
void PaintArea::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.scale(scale,scale);
    if(angle)
    {
        QPointF center(image.width()/2.0,image.height()/2.0);
        painter.translate(center);
        painter.rotate(angle);
        painter.translate(-center);
    }
    if(shear)
    {
        painter.shear(shear,shear);
    }
    painter.drawImage(0,0,image);
}
```

添加的几行代码并不难理解，不再解释。

然后我们从mainwindow.ui 中进入相关菜单的 triggered 事件槽函数，更改如下：

```
void MainWindow::on_action_4_triggered()    //放大菜单
{
    area->zoomIn();
}

void MainWindow::on_action_5_triggered()    //缩小菜单
{
    area->zoomOut();
}

void MainWindow::on_action_6_triggered()    //还原菜单
{
    area->zoom_1();
}

void MainWindow::on_action_7_triggered()    //旋转菜单
{
    area->doRotate();
}

void MainWindow::on_action_8_triggered()    //拉伸菜单
{

```

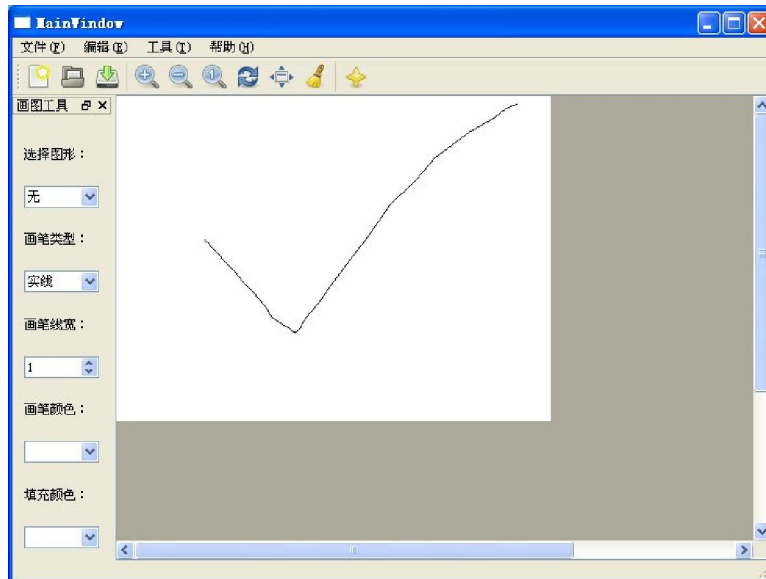


```
        area->doShear();
    }

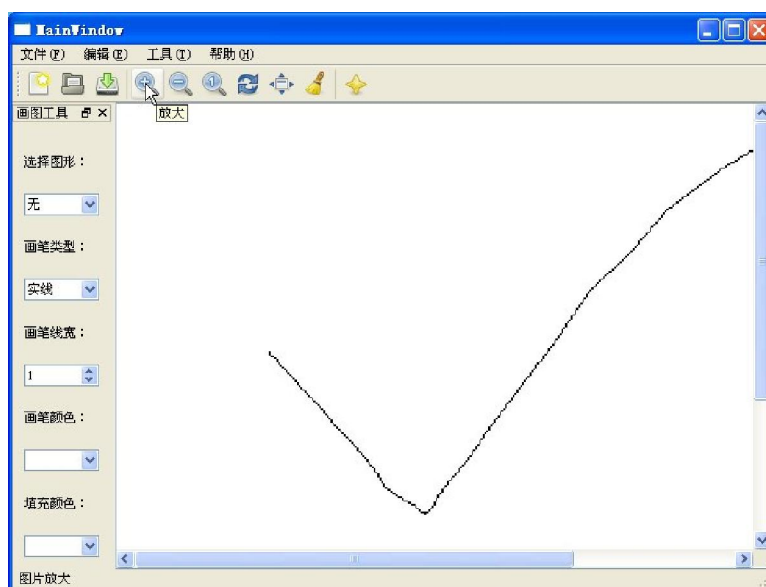
void MainWindow::on_action_10_triggered()    //清空菜单
{
    area->doClear();
}
```

现在运行程序，查看这些菜单的功能效果。

先随意画一个图形：

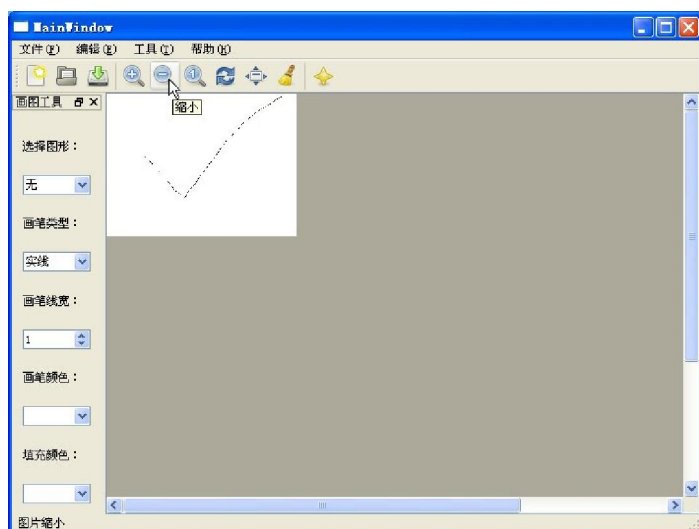


下面是点击“放大”菜单的效果：

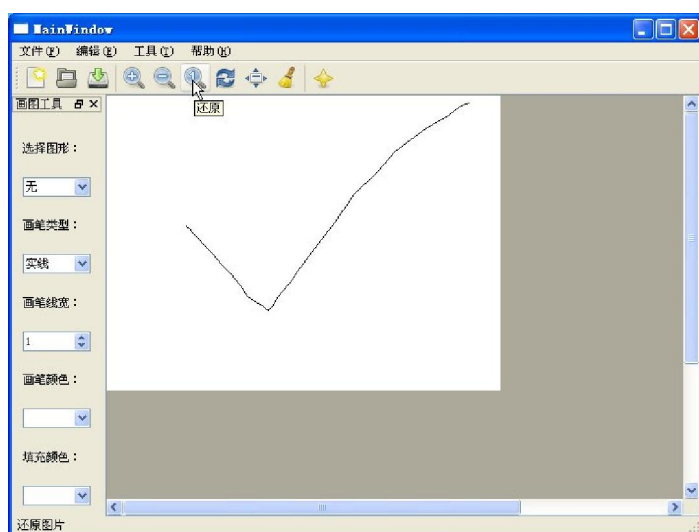




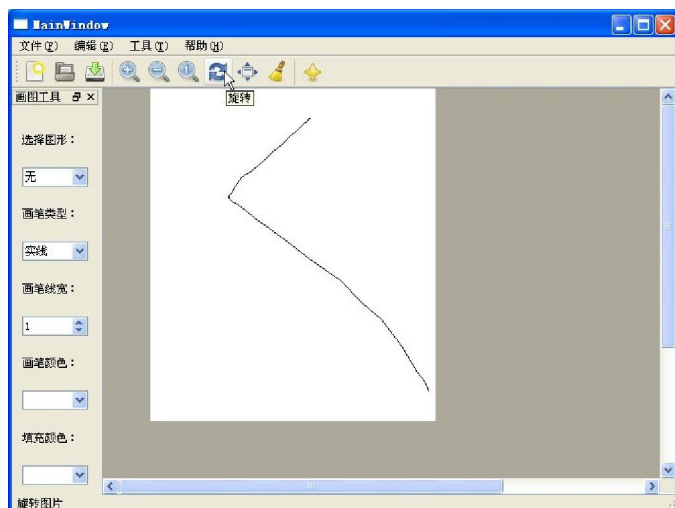
下面是点击“缩小”菜单的效果：



下面是点击“还原”菜单的效果：

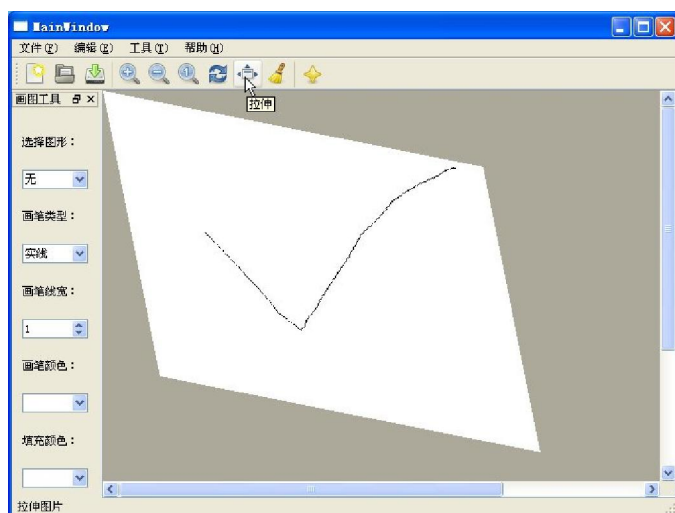


下面是点击“旋转”菜单的效果：

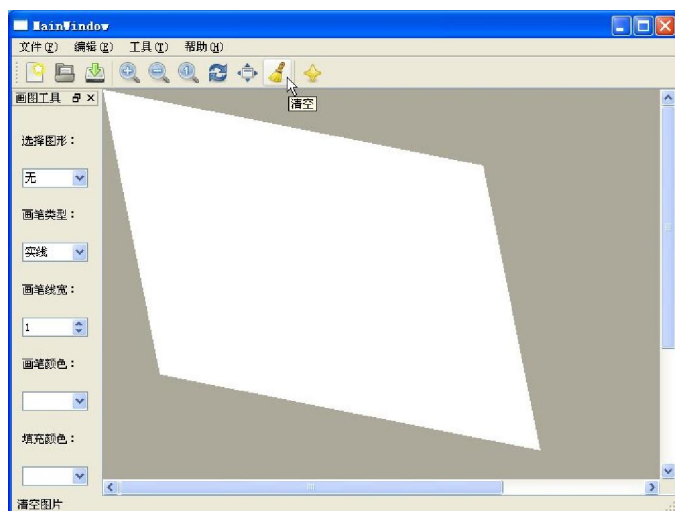




下面是点击“拉伸”菜单的效果：

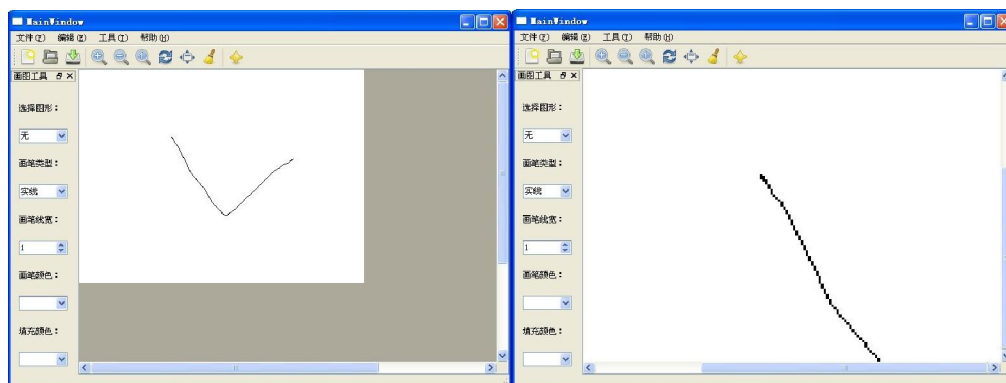


下面是点击“清空”菜单的效果：



问题一：画布无限放大后，滚动条不随着放大。

我们随意画个图形，然后放大几次，最后只能看到它的一部分了。效果如下：



也许你会想到，当时我们在“打开”菜单的函数里加了一行代码来让滚动条放大：



```
scrollArea->widget()->resize(area->getImageSize());  
    //获得图片的大小，然后更改 scrollArea 的大小
```

我们把它放到“放大”菜单的函数里：

```
void MainWindow::on_action_4_triggered()    //放大菜单  
{  
    area->zoomIn();  
    scrollArea->widget()->resize(area->getImageSize());  
    //获得图片的大小，然后更改 scrollArea 的大小  
}
```

这时你再编译运行程序，你会发现，效果并没有改变。这是为什么呢？这个问题我们会在下一部分讲述。

四．完成工具菜单的功能

工具菜单只有一个“绘图工具栏”菜单，它的作用就是当绘图工具栏（就是那个主窗口上的 Dock Widget）被关闭后，点击该菜单，让其可以重新显示出来。

我们从 mainwindow.ui 中进入该菜单的 triggered 事件槽函数，更改如下：

```
void MainWindow::on_action_11_triggered()    //绘图工具栏菜单  
{  
    ui->dockWidget->show();  
}
```

五．完成帮助菜单的功能

帮助菜单就是写一些关于软件使用的帮助和一些版本版权声明。我们这里就不再介绍。

六．完成绘图工具栏的功能

在前一部分设计界面时，绘图工具栏还有两个颜色组合框没有设置。所以我们先对其进行设置。

我们先在 mainwindow.h 中添加头文件：`#include <QComboBox>`

然后在 public 中添加函数声明：

```
void creatColorComboBox(QComboBox *comboBox);    //构建颜色组合框
```

再到 mainwindow.cpp 中进行如下更改。

添加头文件：`#include <QPainter>`

进行该函数的定义：

```
void MainWindow::creatColorComboBox(QComboBox *comboBox)  
{  
    QPixmap pix(16,16);
```



```
QPainter painter(&pix);
```

```
painter.fillRect(0,0,16,16,Qt::black); //先绘制一个 16*16 的小图片，然后给其涂色
comboBox->addItem(QIcon(pix),tr("黑色"),Qt::black); //再用该图片作为组合框条目的
```

图标

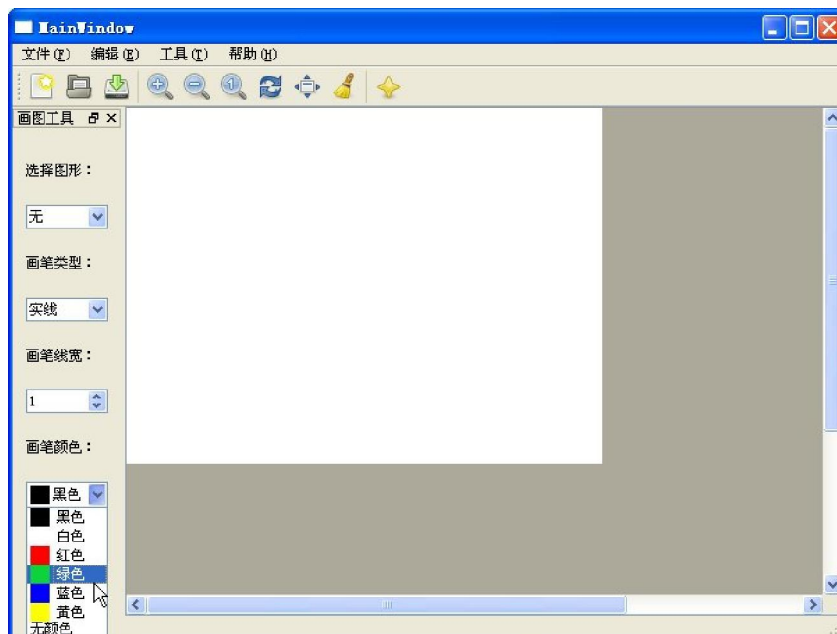
```
painter.fillRect(0,0,16,16,Qt::white);
comboBox->addItem(QIcon(pix),tr("白色"),Qt::white);
painter.fillRect(0,0,16,16,Qt::red);
comboBox->addItem(QIcon(pix),tr("红色"),Qt::red);
painter.fillRect(0,0,16,16,Qt::green);
comboBox->addItem(QIcon(pix),tr("绿色"),Qt::green);
painter.fillRect(0,0,16,16,Qt::blue);
comboBox->addItem(QIcon(pix),tr("蓝色"),Qt::blue);
painter.fillRect(0,0,16,16,Qt::yellow);
comboBox->addItem(QIcon(pix),tr("黄色"),Qt::yellow);

comboBox->addItem(tr("无颜色"),Qt::transparent); //即透明
}
```

在构造函数里对那两个颜色组合框进行初始化：

```
creatColorComboBox(ui->penColorComboBox); //画笔颜色组合框
creatColorComboBox(ui->brushColorComboBox); //填充颜色组合框
```

此时运行程序，两个颜色组合框就有相应的条目了。



下面我们将开始实现绘图工具栏的功能。

1.在 paintarea.h 文件中进行更改。



在 public 中进行函数声明：

```
void setPenStyle(Qt::PenStyle style); //设置画笔风格
void setPenWidth(int width); //设置画笔宽度
void setPenColor(QColor color); //设置画笔颜色
void setBrushColor(QColor color); //设置填充颜色

enum ShapeType //枚举变量，几个图形的选择
{
    None,
    Line,
    Rectangle,
    Ellipse
};
void setShape(ShapeType shape); //设置要绘制的图形
```

在 private 中进行变量的声明：

```
QColor penColor; //画笔颜色
QColor brushColor; //填充颜色
int penWidth; //画笔宽度
Qt::PenStyle penStyle; //画笔风格

ShapeType curShape; //当前图形
```

2.在 paintarea.cpp 中进行更改。

在构造函数中进行变量的初始化：

```
penColor = Qt::black;
brushColor = Qt::black;
penWidth = 1;
penStyle = Qt::SolidLine;
curShape = None;
```

对上面的函数进行定义：

```
void PaintArea::setPenStyle(Qt::PenStyle style)
{
    penStyle = style;
}
void PaintArea::setPenWidth(int width)
{
    penWidth = width;
}
void PaintArea::setPenColor(QColor color)
```



```
{  
    penColor = color;  
}  
  
void PaintArea::setBrushColor(QColor color)  
{  
    brushColor = color;  
}  
  
void PaintArea::setShape(ShapeType shape)  
{  
    curShape = shape;  
}
```

然后更改 paint()函数：

```
void PaintArea::paint(QImage &theImage)  
{  
    QPainter pp(&theImage);    //在 theImage 上绘图  
    QPen pen = QPen();  
    pen.setColor(penColor);  
    pen.setStyle(penStyle);  
    pen.setWidth(penWidth);  
    QBrush brush = QBrush(brushColor);  
    pp.setPen(pen);  
    pp.setBrush(brush);  
  
    int x,y,w,h;  
    x = lastPoint.x();  
    y = lastPoint.y();  
    w = endPoint.x() - x;  
    h = endPoint.y() - y;  
  
    switch(curShape)  
    {  
    case None:                //不绘制特殊图形  
    {  
        pp.drawLine(lastPoint,endPoint);    //由起始坐标和终止坐标绘制直线  
        lastPoint = endPoint;    //让终止坐标变为起始坐标  
        break;  
    }  
    case Line:                //绘制直线  
    {  
        pp.drawLine(lastPoint,endPoint);  
        break;  
    }  
    }
```



```

    }
    case Rectangle:           //绘制矩形
    {
        pp.drawRect(x,y,w,h);
        break;
    }
    case Ellipse:             //绘制椭圆
    {
        pp.drawEllipse(x,y,w,h);
        break;
    }
}

update(); //进行更新界面显示，可引起窗口重绘事件，重绘窗口
modified = true;
}

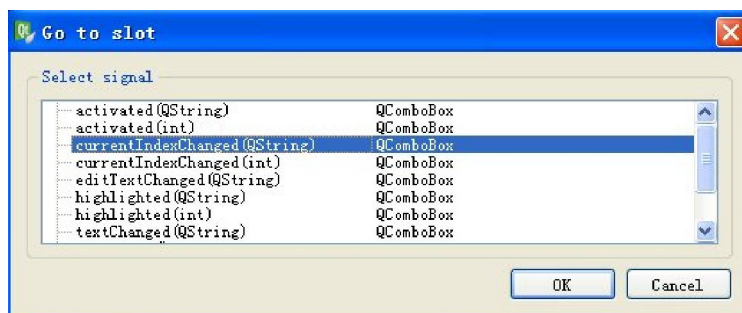
```

3. 写绘图栏中部件的槽函数。

进入 mainwindow.ui 文件，右击“选择图形”下的组合框，选择 Go to slot，如下图：



然后选择当前条目内容更改事件 currentIndexChanged(QString)，如下图：





进入其槽函数，更改如下：

void MainWindow::on_shapeComboBox_currentIndexChanged(QString shape) //选择图形组合框

```
{
    if(shape == tr("无"))
        area->setShape(PaintArea::None);    //利用 PaintArea 类中的枚举变量
    else if(shape == tr("矩形"))
        area->setShape(PaintArea::Rectangle);
    else if(shape == tr("直线"))
        area->setShape(PaintArea::Line);
    else if(shape == tr("椭圆"))
        area->setShape(PaintArea::Ellipse);
}
```

同样的，进入“画笔类型”组合框的 currentIndexChanged(QString)事件槽函数，更改如下：

void MainWindow::on_penStyleComboBox_currentIndexChanged(QString style) //画笔风格组合框

```
{
    if(style == tr("实线"))
    {
        area->setPenStyle(Qt::SolidLine);
    }
    else if(style == tr("点线"))
    {
        area->setPenStyle(Qt::DotLine);
    }
}
```

进入“画笔线宽”的选择框的 valueChanged(int)事件的槽函数，更改如下：

void MainWindow::on_penWidthSpinBox_valueChanged(int width) //画笔线宽组合框

```
{
    area->setPenWidth(width);
}
```

进入两个颜色组合框的 currentIndexChanged(int)事件的槽函数，分别更改如下：

void MainWindow::on_penColorComboBox_currentIndexChanged(int index) //画笔颜色组合框

```
{
    QColor color = ui->penColorComboBox->itemData(index,Qt::UserRole).value<QColor>();

    area->setPenColor(color);
}
```

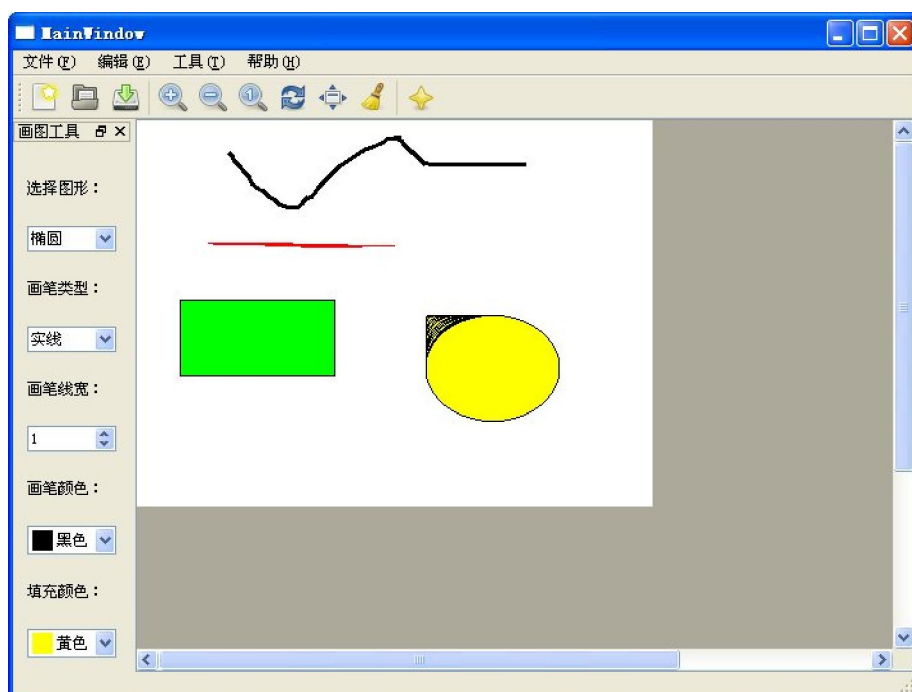


```
}  
void MainWindow::on_brushColorComboBox_currentIndexChanged(int index)    //填充颜色组合框  
{  
    QColor color = ui->brushColorComboBox->itemData(index,Qt::UserRole).value<QColor>();  
  
    area->setBrushColor(color);  
}
```

这里的 `itemData(index,Qt::UserRole).value<QColor>()` 就是和前面进行生成颜色组合框时的 `void MainWindow::creatColorComboBox(QComboBox *comboBox)` 函数里的代码对应的，如 `comboBox->addItem(QIcon(pix),tr("黑色"),Qt::black);` 这里的 `Qt::UserRole` 就是 `Qt::black`，所以我们这里使用了 `value<QColor>()` 函数来获取颜色类型。你可以在帮助中查看 `QComboBox` 类的 `addItem()` 函数。

现在基本的功能我们已经全部实现了。运行程序，进行测试。

我们在画布上用“无”，“直线”，“矩形”，“椭圆”分别进行了绘制，效果如下：

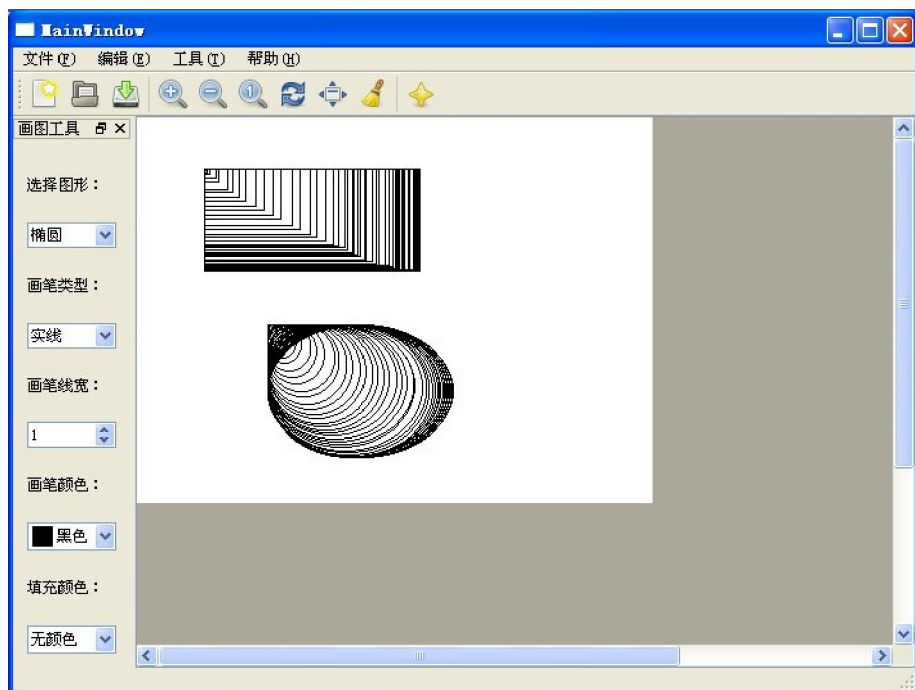


我们发现用“直线”和“椭圆”时都出现了问题，那到底问题出在哪里呢？



问题二：绘制图形时出现混乱。

我们把“填充颜色”设置为“无颜色”，再进行绘制，效果如下：

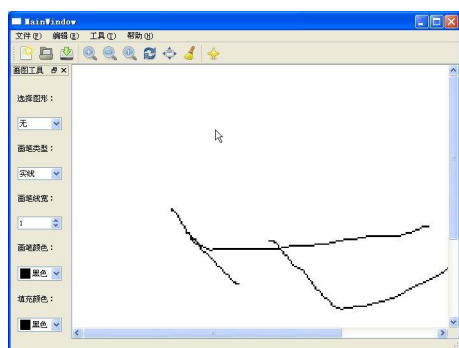


这下你应该看出是什么问题了吧？！比如绘制矩形，这里是绘制了无数的小矩形，因为以前有填充颜色，把其它矩形的边框挡住了，看不到了而已。

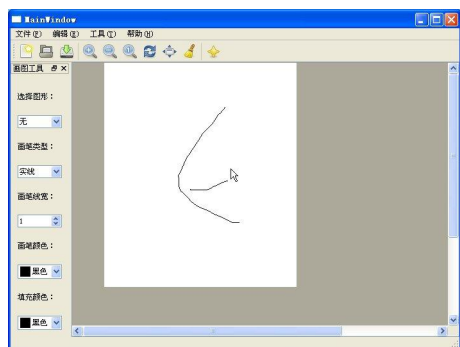
问题三：图形变化后无法进行正常绘图。

我们再看下面的情况。

当随便绘制一个图形后，将其放大，这时如果继续绘制，那么绘制出的图形不会和鼠标指针的轨迹重合。



当我们把画布进行旋转后，纵向绘制直线，可画出来的直线却是横向的。



三个问题的解决：

这一篇虽然完成了所有的功能，但是遗留了三个致命的问题，对于这三个问题的原因和解决办法，我们将在下一部分给出。

我们现在保存文件，并再次对工程文件夹进行备份。我这里把备份文件夹命名为 paint20100208。

第三部分：双缓冲绘图

在上一篇中我们留下了三个问题，下面再叙述一下：

- (1) 画布放大后，滚动条不跟着放大。
- (2) 绘制直线，矩形等图形时有重影。
- (3) 画布进行放大，旋转等操作后无法正常绘图。

对于其中的第一个和第三个问题，它们都与放大等操作有关，而放大等操作又是通过改变坐标系完成的。那么是不是坐标系的原因呢？

我们回想一下当时讲的坐标系的知识，比如放大操作，是将坐标系放大了，那到底是怎么放大的呢？你可以测试一下，如果利用 `painter.scale(2,2)` 函数将横纵坐标都扩大 2 倍，那么原来的 (50, 50) 点是不会变成 (100, 100) 点的，该点的坐标值还是 (50, 50)，虽然我们看着图片是放大了，但是坐标的数值没有变，那变化的是什么呢？是像素的大小，或者说是像素之间的距离。如果以前的像素之间的距离为 1，那么现在就变成了 2。而坐标只是像素的个数，所以它没有变。

这样，你是否已经有所感悟了？！我们进行图片放大，但是却没有改变坐标的值，所以滚动条没有变化。那么第一个问题也就很容易解决了。

解决第 (1) 个问题：

我们在 `paintarea.cpp` 中更改 `getImageSize()` 函数如下：

```
QSize PaintArea::getImageSize()
{
    return image.size()*scale;
}
```

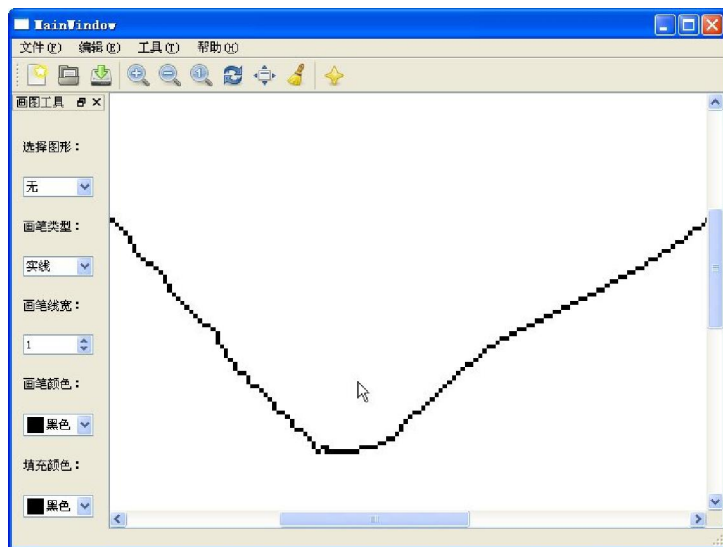



这样就能返回我们需要的画布的大小的值了。

然后在 mainwindow.cpp 中对放大菜单进行更改：

```
void MainWindow::on_action_4_triggered()    //放大菜单
{
    area->zoomIn();
    scrollArea->widget()->resize(area->getImageSize());
    //获得图片的大小，然后更改 scrollArea 的大小
}
```

这时添加这一行代码就有效果了。运行程序后效果如下：



解决完第一个问题，那么第三个问题也就应该有思路了。我们将图片放大后，鼠标指针所在点的坐标值与我们想要绘画点的坐标值已经不同了。比方说鼠标指针放在坐标值为（100，100）的点处，因为鼠标指针的坐标值是由窗口部件提供的，而这时画布的坐标是那个程序中的 image 变量的坐标，在没有改变画布大小时，鼠标指针所在点（100，100）和画布上的点（100，100）是重合的。但是将画布利用 painter.scale(2,2)扩大两倍后，鼠标指针所在的点（100，100）处，在画布上已经是（50，50）了，而我们绘图是获取的鼠标指针的坐标值，所以就出现了绘制的线条与鼠标指针的轨迹不重合的问题。

知道了原因，更改就容易多了。

解决第（3）个问题：

在 paintarea.cpp 中更改 paint()函数：

```
void PaintArea::paint(QImage &theImage)
{
    QPainter pp(&theImage);    //在 theImage 上绘图
    QPen pen = QPen();
    pen.setColor(penColor);
    pen.setStyle(penStyle);
    pen.setWidth(penWidth);
    QBrush brush = QBrush(brushColor);
```



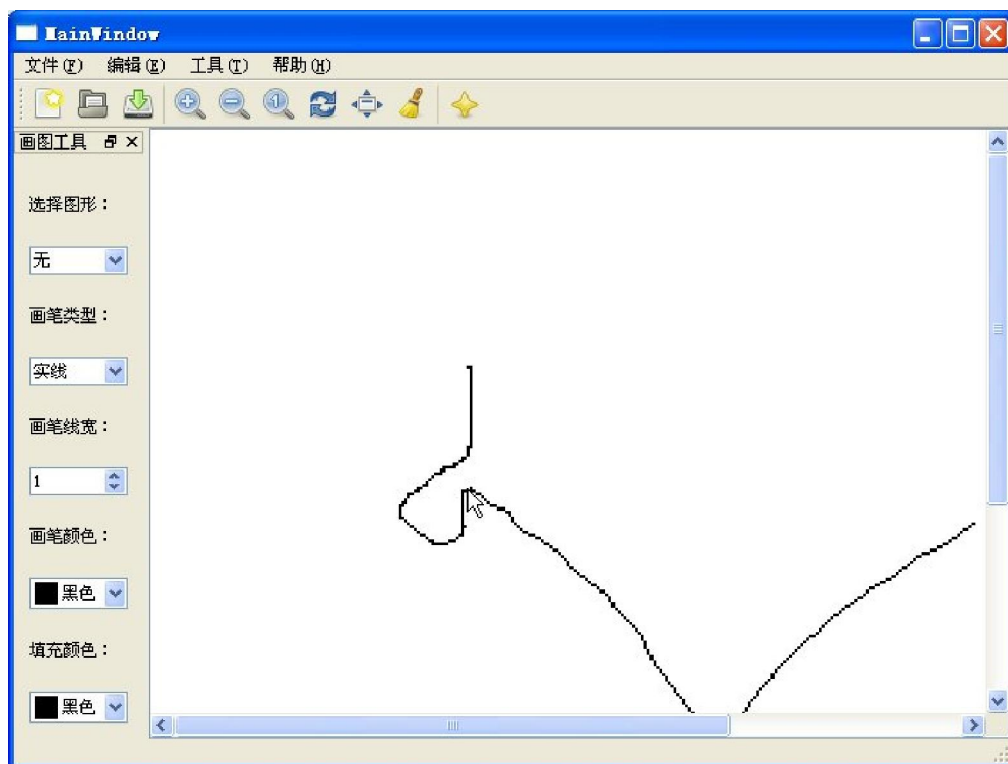
```
pp.setPen(pen);
pp.setBrush(brush);

int x,y,w,h;
x = lastPoint.x()/scale;
y = lastPoint.y()/scale;
w = endPoint.x()/scale - x;
h = endPoint.y()/scale - y;

switch(curShape)
{
case None:
    {
        pp.drawLine(lastPoint/scale,endPoint/scale);    //由起始坐标和终止坐标绘制直
线
        lastPoint = endPoint; //让终止坐标变为起始坐标
        break;
    }
case Line:
    {
        pp.drawLine(lastPoint/scale,endPoint/scale);
        break;
    }
case Rectangle:
    {
        pp.drawRect(x,y,w,h);
        break;
    }
case Ellipse:
    {
        pp.drawEllipse(x,y,w,h);
        break;
    }
}

update();    //进行更新界面显示，可引起窗口重绘事件，重绘窗口
modified = true;
}
```

这里就是将鼠标的坐标缩小了要放大的倍数。效果如下：



放大时绘图的问题已经解决了，那么旋转、拉伸等操作后的绘图问题怎么解决呢？比方说旋转操作，我们将图片旋转了 90 度，我们想保存现在的图片效果，又不想改变坐标系的状态，只有这样我们才能在旋转后的图片上正常绘图。在这里，我们可以添加一个 QImage 临时变量，把图片内容先给它，然后用它完成旋转操作，再把旋转后的图片传回来。

我们在 paintarea.cpp 中的重绘函数里进行如下更改：

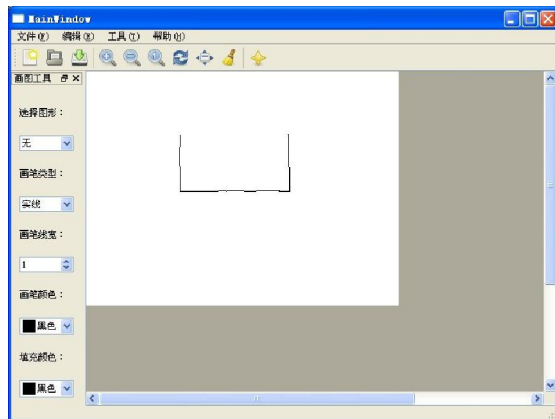
```
void PaintArea::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.scale(scale,scale);
    if(angle)
    {
        QImage copyImage = image; //新建临时的 copyImage，利用它进行旋转操作
        QPainter pp(&copyImage);
        QPointF center(copyImage.width()/2.0,copyImage.height()/2.0);
        pp.translate(center);
        pp.rotate(angle);
        pp.translate(-center);
        pp.drawImage(0,0,image);
        image = copyImage; //只会复制图片上的内容，不会复制坐标系统
        angle = 0; //完成旋转后将角度值重新设为 0
    }
    if(shear)
    {
```



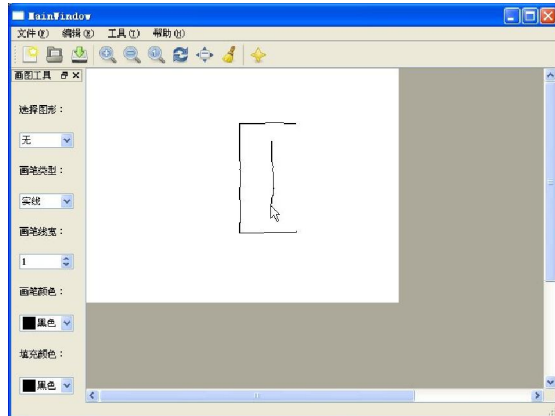
```
    QImage copyImage = image;
    QPainter pp(&copyImage);
    pp.shear(shear,shear);
    pp.drawImage(0,0,image);
    image = copyImage;
    shear = 0;
}
painter.drawImage(0,0,image);
}
```

这样更改后，运行程序，效果如下：

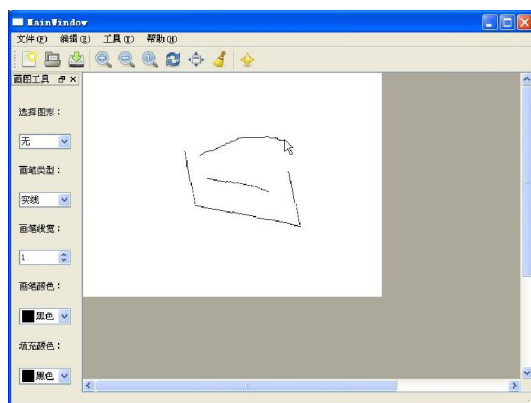
随意绘制一个图形：



旋转 90 度后再绘图：



拉伸后再绘图：



这样第三个问题就解决了。



到这里，我们应该对坐标系统有了一个更高层次的理解。而对于利用临时变量进行绘图也应该有些感觉了。这时我们再考虑第二个问题，我们绘制出的矩形是无数个小矩形的重叠，更确切地说是很多个矩形由小变大。我们回想一下我们所写的程序，每移动一下鼠标，获取一个坐标值，马上就进行绘制，所以这样就出现了移动完鼠标，绘制出很多个矩形的问题。我们能不能也再引入一个变量来解决这个问题呢？

我们可以设想，如果要绘制直线，矩形等特殊图形，当鼠标移动时我们不在真正的画布上进行绘图，而是将它绘制在一个临时画布上，当鼠标左键释放了，我们才在真正的画布上进行图形最终形状的绘制，这样不就解决了第二个问题。

解决第（2）个问题：

我们现在进行程序的更改：

在 paintarea.h 中的 private 中我们声明两个变量：

```
QImage tempImage;    //临时绘图区
bool isDrawing;       //是否在绘制特殊图形
```

在 paintarea.cpp 中我们在构造函数里对变量进行初始化：`isDrawing = false;`

然后对鼠标事件进行如下更改：

```
void PaintArea::mousePressEvent(QMouseEvent *event)
{
    if(event->button() == Qt::LeftButton) //当鼠标左键按下
    {
        lastPoint = event->pos(); //获得鼠标指针的当前坐标作为起始坐标
        isDrawing = true;
    }
}

void PaintArea::mouseMoveEvent(QMouseEvent *event)
{
    if(event->buttons() & Qt::LeftButton) //如果鼠标左键按着的同时移动鼠标
    {
        endPoint = event->pos(); //获得鼠标指针的当前坐标作为终止坐标
        if(curShape == None) //如果不进行特殊图形绘制，则直接在 image 上绘制
        {
            isDrawing = false;
            paint(image);
        }
        else //如果绘制特殊图形，则在临时绘图区 tempImage 上绘制
        {
            tempImage = image; //每次绘制 tempImage 前用上一次 image 中的图片对其进行填充
            paint(tempImage);
        }
    }
}
```



```
}  
void PaintArea::mouseReleaseEvent(QMouseEvent *event)  
{  
    if(event->button() == Qt::LeftButton)    //如果鼠标左键释放  
    {  
        endPoint = event->pos();  
        isDrawing = false;  
        paint(image);  
    }  
}
```

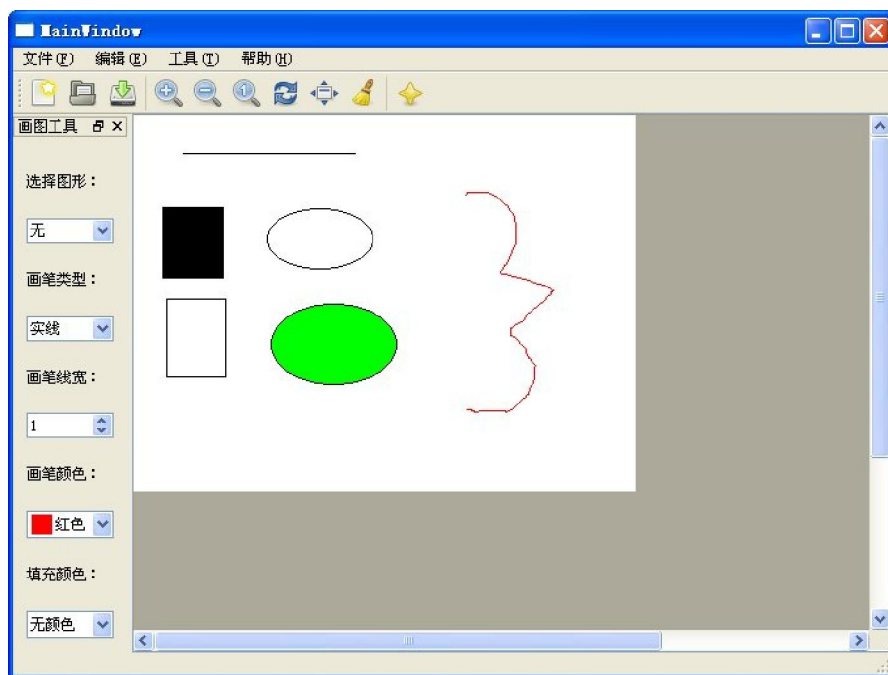
最后对重绘事件进行更改：

```
void PaintArea::paintEvent(QPaintEvent *)  
{  
    QPainter painter(this);  
    painter.scale(scale,scale);  
    if(isDrawing)    //如果正在绘制特殊图形，则显示临时绘图区上的内容  
    {  
        painter.drawImage(0,0,tempImage);  
    }  
    else  
    {  
        if(angle)  
        {  
            QImage copyImage = image;    //新建临时的 copyImage，利用它进行旋转操作  
            QPainter pp(&copyImage);  
            QPointF center(copyImage.width()/2.0,copyImage.height()/2.0);  
            pp.translate(center);  
            pp.rotate(angle);  
            pp.translate(-center);  
            pp.drawImage(0,0,image);  
            image = copyImage;    //只会复制图片上的内容，不会复制坐标系统  
            angle = 0;    //完成旋转后将角度值重新设为 0  
        }  
        if(shear)  
        {  
            QImage copyImage = image;  
            QPainter pp(&copyImage);  
            pp.shear(shear,shear);  
            pp.drawImage(0,0,image);  
            image = copyImage;  
            shear = 0;  
        }  
        painter.drawImage(0,0,image);  
    }  
}
```



```
}  
}
```

这时运行程序，效果如下：



可以看到绘制特殊图形时已经没有问题了。

我们注意一下上面程序中的这一句：

`tempImage = image;` //每次绘制 tempImage 前用上一次 image 中的图片对其进行填充
这就是最关键的一步，就是这句代码，让我们实现了可以看着图形的大小变化进行绘图，也就是所谓的交互式绘图。其中的那个 `isDrawing` 变量，作为是否绘制特殊图形的标志，如果绘制特殊图形，就在临时绘图区进行，如果不是，就直接绘制在真实的画布上。

我们上面所用的方法，有个专业名称，叫做“双缓冲绘图”。

双缓冲绘图简介：

到底为什么叫双缓冲绘图呢？那我们现在就来简单总结一下以前进行过的绘图。

无缓冲绘图：

如果在重绘事件函数里直接进行绘图，如：

```
void Dialog::paintEvent(QPaintEvent *)  
{  
    QPainter painter(this); //this 表明直接在窗口上绘图  
    painter.drawLine(0,0,100,100);  
}
```

这样直接在窗口上绘图就可以叫做无缓冲绘图。



单缓冲绘图：

而如果先在 QImage 或 QPixmap 上进行绘图，然后再将其绘制好的内容整体在窗口上绘制出来，就是单缓冲绘图。我们上面的程序开始时就是单缓冲绘图。

双缓冲绘图：

就像我们程序最后一样，利用两个 QImage 对象，一个保存临时图像，一个保存最终图像，这样实现交互绘图的方法就叫做双缓冲绘图。

到这里，整个绘图软件就完成了，我们从这篇文章中应该学到一些东西，不光是 Qt 应用的知识，更重要的是写程序，写一个较大点软件的流程和技巧。

我们 2D 绘图的内容讲到这里也就结束了。你可以发现一个大点的程序是很难讲述清楚地，而 Qt 的 2D 绘图的内容还有很多，我们不可能全部像这样详细讲述，这些教程也只是起一个基本入门的作用，要想深入学习，还要靠自己的多加练习。

最后，如果你喜欢我的写作风格，并且初学 Qt，可以在我的空间查看 Qt Creator 系列教程，希望能对你的入门有所帮助。