Лабораторная работа 12

Создание графического редактора

Цель работы: создать графический редактор, позволяющий:

- Создавать, редактировать, загружать, сохранять изображения;
- Рисовать с помощью мыши (при нажатии левой кнопки мыши и её перемещении отображается кривая движения указателя мыши. При нажатии правой кнопки мыши появляется стирательная резинка);
- Задавать цвет, толщину и стиль линии;
- Пользоваться историей изменений в обе стороны – undo и redo.

Компоненты: *MenuStrip, ToolStrip, Panel, ColorDialog, OpenFileDialog, SaveFileDialog, PictureBox, ImageList, TrackBar, ComboBox*.

Теоритические сведения

Компонент MenuStrip. Для быстрого вызова команд можно использовать так называемые быстрые клавиши. Для этого надо установить свойство ShowShortCutKeys, выбрав значение True. Также установить свойство ShortCutKeys, выбрав значение из списка (или набрать). При этом нужно следить, чтобы быстрые клавиши не повторялись во избежание коллизий. Можно использовать любые готовые иконки либо создать их самостоятельно. Для этого в свойствах необходимо найти Image и дважды нажать на значение свойства, появится окно «Выбор ресурса». В окне выберете контекст ресурса (Локальный или Файл ресурсов проекта). Локальный – если вы хотите установить собственную иконку, Файл ресурсов проекта – если вас устраивают стандартные иконки (windows theme).

Компонент ToolStrip. Представляет собой специальный контейнер для создания панелей инструментов. Может управлять любыми вставленными в него дочерними элементами: группировать, выравнивать по размерам, располагать элементы в несколько рядов.

Специально для ToolStripPanel разработан компонент ToolStripButton (кнопка панели инструментов, отсутствует в палитре компонентов). Для добавления в панель компонента ToolStripButton надо: щелкнуть правой кнопкой мыши на ToolStripPanel и выбрать Button|Label|SplitButton|DropDownButton|Separator|ComboBox|TextBox|Progres
sBar. На кнопки можно поместить изображения. Для этого надо установить свойство Image.

Компонент PictureBox. Служит для размещения на форме одного из трех поддерживаемых типов изображений: растрового изображения, значка и метафайла.

Растровое изображение – это произвольные графические изображения в файлах со стандартным расширением .bmp. Значки (иконки) – небольшие растровые изображения, снабженные специальными средствами, регулирующими их прозрачность. Расширение файлов значков, обычно, .ico. (Метафайл – это изображение, построенное на графическом устройстве с помощью специальных команд, которые сохраняются в файле с расширением .wmf)

Компонент TrackBar – ползунок. Для плавного изменения числовой величины (во многом схож с компонентом ScrollBar).

Некоторые свойства:

- Maximum – определяет максимальное значение диапазона изменения;
- Minimum – определяет минимальное значение диапазона изменения;
- Value: integer – определяет текущее положение ползунка;
- Orientation – ориентация компонента (горизонтальная, вертикальная);

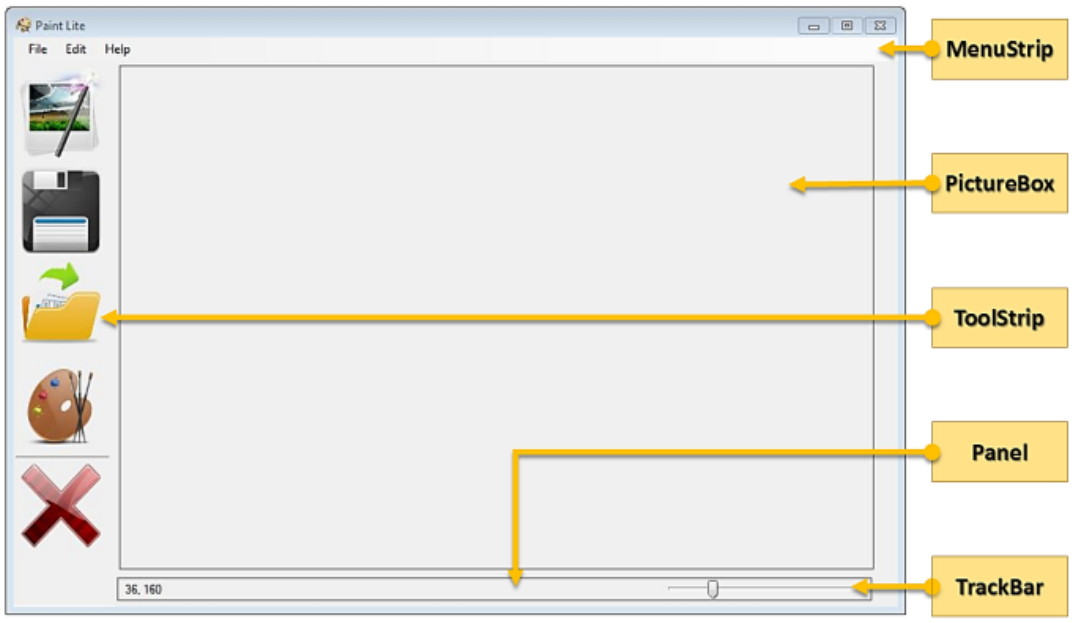
События мыши. Для выполнения какого-либо действия с помощью щелчка левой кнопки мыши для большинства случаев достаточно запрограммировать обработчик событий OnClick, для реакции на двойной щелчок используется событие OnDbClick. Для более совершенного управления мышью лучше использовать обработчики следующих событий:

- OnMouseDown. Вызывается при нажатии любой кнопки мыши.
- OnMouseMove. Вызывается при перемещении мыши.
- OnMouseUp. Вызывается при отпускании какой-нибудь кнопки мыши.

Процедуры обработки этих событий получают следующие параметры:

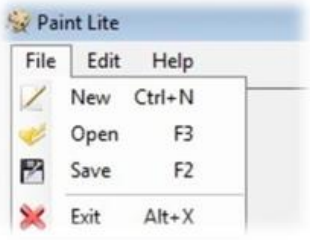
- Sender. Представляет объект, который получил это событие (на каком объекте щелкнули мышью).
- Button. Имеет одно из трех значений: MouseButtons.Right, MouseButtons.Left, MouseButtons.Middle. Используется для определения того, какую кнопку мыши нажал пользователь.
- X, Y. Координаты указателя мыши в пикселях относительно клиентной области окна с координатами (0,0) в верхнем левом углу.

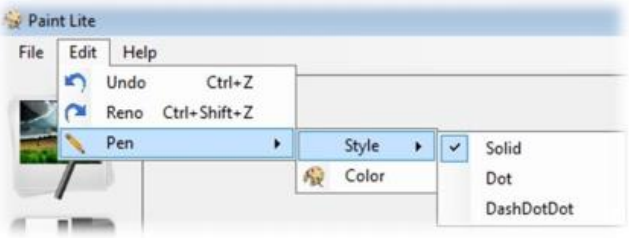
Упражнение 1
Создание формы



Создание визуальной части

1. Создаем главное меню следующего вида:





Необходимо назначить все горячие клавиши и выставить по умолчанию свойство *Checked = true* у *Pen->Style->Solid*, как показано на скриншоте.

2. Создаём форму *ToolStrip* и помещаем туда продублированные команды New, Open, Save, Color и Exit, либо другие пункты на собственное усмотрение.

3. Добавляем управление толщиной пера и вывод текущих координат. Для этого помещаем в удобное место (например под *PictureBox*) *Panel*, *Label* и *TrackBar*, как показано на скриншоте. *Label* будем использовать один общий для двух координат, чтобы не было их смещения, как могло бы быть при использовании двух независимых *Label* отдельно для разделения координат *X* и *Y*.

Создание невизуальной части

Условимся, что PictureBox будет называться picDrawingSurface.

Для начала необходимо запрограммировать работу всех пунктов меню у File и Help. В пункте меню Help можно указать версию программы, имя разработчика и возможности программы. Это творческое задание.

Создание нового файла. Ограничиться только одним перемещением PictureBox на форму для рисования в нем не получится. Ведь там рисовать будет нельзя, даже если очень захочется. Ведь свойство *Image* у *PictureBox* не инициализируется от одного добавления последнего на форму и при попытке что-либо чиркнуть в этом поле в откомпилированной программе – появится ошибка *System.ArgumentNullException* или подобная ей. Поэтому пункт меню New и будет сводиться к следующим строчкам кода:

```
Bitmap pic = new Bitmap(750, 500);
picDrawingSurface.Image = pic;
```

Размеры *new Bitmap* определяем из размеров *PictureBox*, помещенного в форму. Теперь в таком поле появится возможность рисовать.

Примечание 1. Для того, чтобы пользователь по ошибке не начал рисовать в неинициализированной области *PictureBox* (ведь при запуске программы у нас еще ничего не готово) создадим своеобразную «защиту»: при попытке рисовать на неинициализированном *PictureBox* будет всплывать сообщение с предупреждением «Сначала создайте новый файл!» и дальнейшее предотвращение попытки рисования. Для этого нужно зайти в события PictureBox и выбрать событие *MouseDown*. Дважды щелкаем по нему и в открывшемся редакторе кода прописываем следующие строчки:

```
if (picDrawingSurface.Image == null)
{
      MessageBox.Show("Сначала создайте новый файл!");
      return;
}
```

Если поле для рисования *Image* не инициализировано (равно *null*), то выводим предупреждающее пользователя сообщение и выходим из функции. Таким образом будет предотвращена попытка рисования в неинициализированной области и программа не вылетит с ошибкой.

Для возможности сохранить изображение будем использовать работы *SaveFileDialog*. Здесь создаем объект *SaveDlg* и прописываем ему параметры:

```
SaveFileDialog SaveDlg = new SaveFileDialog();
SaveDlg.Filter = "JPEG Image|*.jpg|Bitmap Image|*.bmp|GIF Image|*.gif|PNG
Image|*.png";
SaveDlg.Title = "Save an Image File";
SaveDlg.FilterIndex = 4; //По умолчанию будет выбрано последнее расширение
*.png
SaveDlg.ShowDialog();
```

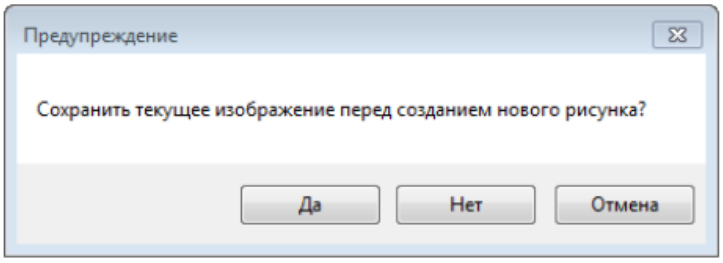
Теперь нужно прописать код для сохранения картинки в различных расширениях. Для этого используем *switch*.

```
if (SaveDlg.FileName != "") //Если введено не пустое имя
{
      System.IO.FileStream fs = (System.IO.FileStream)SaveDlg.OpenFile();
            switch (SaveDlg.FilterIndex)
            {
                  case 1:
                        this.picDrawingSurface.Image.Save(fs,
                  ImageFormat.Jpeg);
                        break;
            case 2:
                        this.picDrawingSurface.Image.Save(fs,
                  ImageFormat.Bmp);
                        break;
            case 3:
                        this.picDrawingSurface.Image.Save(fs,
                  ImageFormat.Gif);
                        break;
            case 4:
                        this.picDrawingSurface.Image.Save(fs,
                  ImageFormat.Png);
                        break;
            }
      fs.Close();
}
```

Примечание 2. Если вдруг пользователь захотел создать еще один новый файл, при условии, что уже было что-то нарисовано в *PictureBox*, можно будет ему предложить сохранить текущее изображение, чтобы оно не потерялось. Ведь создание нового файла ведет к удалению предыдущего. Для этого потребуется следующий кусок кода:

```
if (picDrawingSurface.Image != null)
{
      var result = MessageBox.Show("Сохранить текущее изображение перед
      созданием нового рисунка?", "Предупреждение",
      MessageBoxButtons.YesNoCancel);
      switch (result)
      {
            case DialogResult.No: break;
            case DialogResult.Yes: saveMenu_Click(sender, e); break;
            case DialogResult.Cancel: return;
     }
}
```

Здесь, как и в предыдущем примечании 1, мы используем условный оператор для того, чтобы узнать, инициализирована ли форма *PictureBox* или нет. Если она уже была инициализирована, значит нужно предложить пользователю сохранить изображение. Данный *MessageBox* будет иметь 3 кнопки: *Yes, No* или *Cancel*. В зависимости от выбора пользователя будем решать, как поступить: отменить создание нового файла (*Cancel*), сохранить данное изображение (*Yes*) или не сохранять (*No*). Данный *MessageBox* будет выглядеть следующим образом:



Вставить данный кусок кода нужно непосредственно в метод, ответственный за пункт меню New.

Теперь перейдем непосредственно к Open. По тому же принципу, как и в Save, создаем объект класса *OpenFileDialog* и прописываем ему ряд параметров:

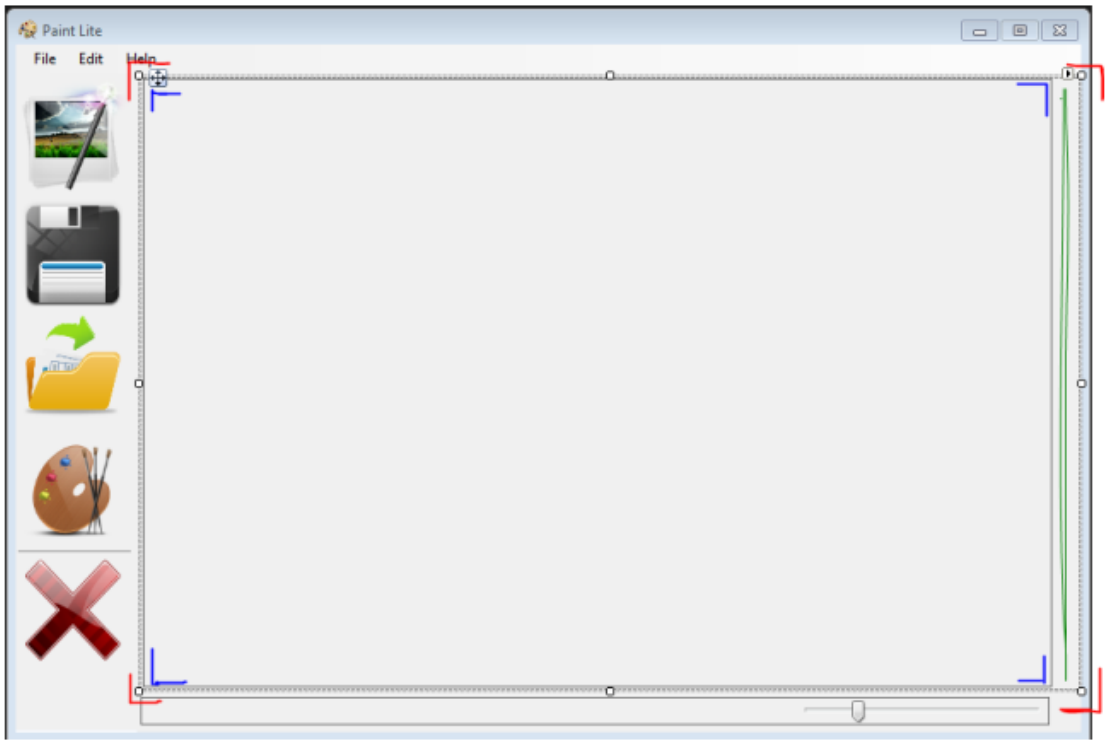
```
OpenFileDialog OP = new OpenFileDialog();
OP.Filter = "JPEG Image|*.jpg|Bitmap Image|*.bmp|GIF Image|*.gif|PNG
Image|*.png";
OP.Title = "Open an Image File";
OP.FilterIndex = 1; //По умолчанию будет выбрано первое расширение *.jpg
```

И, когда пользователь укажет нужный путь к картинке, ее нужно будет загрузить в PictureBox:

```
if (OP.ShowDialog() != DialogResult.Cancel)
     picDrawingSurface.Load(OP.FileName);
     picDrawingSurface.AutoSize = true;
```

Примечание 3. Если пользователь захочет загрузить картинку, размеры которой превышают размеры самого окна *PictureBox*, то возникнет одна проблема. Во-первых, загруженная картинка не сожмется до размеров окна *PictureBox*, а откроется в полном

размере, но видно будет лишь та часть, которая влезет в окно. Вовторых, у пользователя не будет возможности перемещать данную картинку во всех направлениях для ее просмотра и изменения. Для решения этой проблемы можно воспользоваться следующей хитростью: нужно поместить под *PictureBox* *Panel*. С помощью *Panel* у пользователя появится возможность двигать картинку во всех направлениях в том случае, если ее размер будет превышать размеры окна *PictureBox*.



Здесь красными уголками выделен элемент *Panel*, синими – *PictureBox*, а зеленой двойной линией показан отступ, который нужно оставить для вертикальной полосы прокрутки, который появится, как только длинна загруженной картинки станет больше длинны *PictureBox*. Для горизонтальной полосы прокрутки место оставлять не нужно, она появится в нужном месте сама.

4. Самостоятельно запрограммируйте весь *ToolStrip*. Для этого достаточно продублировать все методы из *MenuStrip*, которые вы продублировали на свой выбор.
5. Переходим непосредственно к рисованию. Для начала пропишем ряд глобальных переменных:

```
bool drawing;
GraphicsPath currentPath;
Point oldLocation;
Pen currentPen;
```

И пропишем пару строк в конструкторе формы:

```
public Form1()
{
      InitializeComponent();
      drawing = false; //Переменная, ответственная за рисование
      currentPen = new Pen(Color.Black); //Инициализация пера с
черным цветом
}
```

Так же для рисования нам потребуется добавить еще 2 события помимо *MouseDown* – *MouseUp* и *MouseMove*.

MouseDown отвечает за нажатую кнопку мыши, *MouseUp* – за отпущенную, а *MouseMove* соответственно за перемещение мыши. Алгоритм будет следующий: при нажатии клавиши мы активируем режим рисования (для этого и нужна переменная типа *bool*, названная *drawing*, которая в момент нажатия мыши будет принимать значение *true*, а в момент отпускания *false*). Тогда, при нажатии мыши мы активируем режим рисования, а в момент отпускания деактивируем. Дополним код *MouseDown*:

```
private void picDrawingSurface_MouseDown(object sender, MouseEventArgs e)
{
 if (picDrawingSurface.Image == null)
 {
       MessageBox.Show("Сначала создайте новый файл!");
       return;
 }
 if (e.Button == MouseButtons.Left)
 {
       drawing = true;
       oldLocation = e.Location;
       currentPath = new GraphicsPath();
 }
}
```

и пропишем код для MouseUp:

```
private void picDrawingSurface_MouseUp(object sender, MouseEventArgs e)
{
 drawing = false;
 try
 {
       currentPath.Dispose();
 }
 catch { };
}
```

При каждом нажатии мыши мы будем выделять память для currentPath под GraphicsPath (этот класс представляет последовательность соединенных линий и кривых), ну, а чтобы не было переполнения памяти, в MouseUp будем удалять объект currentPath.

Само рисование будет происходить в MouseMove. Пропишем это событие:

```
private void picDrawingSurface_MouseMove(object sender, MouseEventArgs e)
{
      if (drawing)
      {
            Graphics g = Graphics.FromImage(picDrawingSurface.Image);
            currentPath.AddLine(oldLocation, e.Location);
            g.DrawPath(currentPen, currentPath);
            oldLocation = e.Location;
            g.Dispose();
            picDrawingSurface.Invalidate();
     }
}
```

Таким образом, только в случае, если *drawing* будет равное *true* (клавиша мыши нажата), то будет происходить рисование. С помощью данного кода за нажатой мышкой будет рисоваться линия. Теперь можно протестировать программу.

6. Ластик по нажатию правой кнопки мыши. Для самостоятельного решения. Алгоритм будет следующий. Сначала нужно создать новую глобальную переменную Color historyColor. В нее нужно будет сохранять текущий цвет пера при нажатии на правую клавишу мыши в событии *MouseDown*. Далее, в том же самом событии для правой клавиши мыши нужно будет менять цвет пера с текущего на белый в переменной *currentPen*. А в событии *MouseUp* нужно возвращать цвет пера, который был до использования ластика с помощью переменной *historyColor*.

7. Привязываем *TrackBar* для толщины пера и выводим текущие координаты. Для начала пропишем код для вывода текущих координат. Как уже было сказано выше, мы будем использовать лишь один *Label*, который расположили в *Panel* под *PictureBox*. Код будем прописывать в событии *MouseMove*, что достаточно очевидно. Итак, дополним *MouseMove* следующей строчкой:

```
label_XY.Text = e.X.ToString() + ", " + e.Y.ToString();
```

Здесь *label_XY* – это наш *Label*, (у вас по умолчанию он будет называться label1). Но для информативности в пределах данной работы он был переименован. *Label* приминает значение *string*. Параметр MouseEventArgs e передаст координаты *X* и *Y*, если к нему обратиться как *e.X* и *e.Y*, соответственно. По умолчанию они (координаты) имеют целочисленный тип, поэтому к ним мы добавляем через точку метод *ToString()*. Операцией сложения мы добавляем разделение двух координат через запятую. Таким образом при движении мыши будут постоянно обновляться координаты и выводиться в виде *X*, *Y*, где под *X* и *Y* будут подставлены текущие координаты положения мыши в поле *PictureBox*. Эту строчку нужно вставить в самом начале метода *picDrawingSurface_MouseMove* перед условным оператором, т.к. координаты нас будут интересовать и без нажатой кнопки мыши.

Теперь перейдем к подключению *TrackBar*. У него есть 3 интересующих нас свойства: *Maximum*, *Minimum* и *Value*. В данной работе, в *Maximum* задано значение 20, в *Minimum* 1, а в *Value* 5. Значения заданы в конструкторе форм, хотя их можно прописать в ручную в конструкторе класса *Form1*. При движении ползунка *TrackBar* будет меняться значение *Value* от *Minimum* до *Maximum*. Именно *Value* нас и будет интересовать. Дополним код конструктора Form1:

```
public Form1()
{
      InitializeComponent();
      drawing = false; //Переменная, ответственная за рисование
      currentPen = new Pen(Color.Black); //Инициализация пера с
черным цветом
     currentPen.Width = trackBarPen.Value; //Инициализация толщины
пера
}
```

И, щелкнув по *TrackBar* дважды в конструкторе форм, пропишем код:

```
private void trackBar1_Scroll(object sender, EventArgs e)
{
      currentPen.Width = trackBarPen.Value;
}
```

Все готово, можно откомпилировать и проверить новые возможности вашего графического редактора.

История изменений Undo и Redo. Переходим к самой интересной части. Идея такая: создать некоторый динамический список, в котором будет хранить последние 10 изменений в редакторе (рисование, стирание). Можно сделать другое количество изменений, но предупреждаю, каждое такое изменение будет отщипывать у оперативной памяти порядка 2 мб. В пределах данной методички мы не будем рассматривать более оптимизированные способы хранения истории, т.к. важно понять сам принцип работы с нею. Итак, приступим. Для начала нужно определиться какой именно динамический массив (список) мы будем использовать. В данной работе был выбран *List*. А так же нам потребуется переменная-счетчик для истории. Дополним поля класса этими переменными:

```
bool drawing;
int historyCounter; //Счетчик истории

GraphicsPath currentPath;
Point oldLocation;
public Pen currentPen;
      Color historyColor; //Сохранение текущего цвета перед
      использованием ластика
      List<Image> History; //Список для истории
```

Наш список будет хранить переменные типа *Image*, т.к. область, в которой мы рисуем в *PictureBox* так же является объектом этого типа. В конструкторе формы нужно выделить память под данный список:

```
History = new List<Image>(); //Инициализация списка для истории
```

Теперь алгоритм следующий. При создании нового файла нужно занести только что созданное чистое поле в историю. Далее, при каждом отпускании левой клавиши мыши нужно заносить в список истории только что нарисованную область. Также нужно следить за переполнением истории. В этом же событии *MouseUp* нужно сделать проверку на ее размер. Если в ней уже более 9 элементов, то удалять первый. А также нужно очищать историю, как только пользователь захотел создать новый файл.

Дополним метод создания нового файла:

```
private void newToolStripMenuItem_Click(object sender, EventArgs e)
{
      History.Clear();
      historyCounter = 0;
      Bitmap pic = new Bitmap(750, 500);
      picDrawingSurface.Image = pic;
     History.Add(new Bitmap(picDrawingSurface.Image));
}
```

Дополним событие *MouseUp*:

```
private void picDrawingSurface_MouseUp(object sender, MouseEventArgs e)
{
//Очистка ненужной истории
      History.RemoveRange(historyCounter + 1, History.Count -
historyCounter - 1);
      History.Add(new Bitmap(picDrawingSurface.Image));
      if (historyCounter + 1 < 10) historyCounter++;
      if (History.Count - 1 == 10) History.RemoveAt(0);
      drawing = false;
      try
      {
            currentPath.Dispose();
      }
      catch { };
}
```

В событии MouseUp не забываем, что должны быть еще пару строк для ластика, которые нужно было прописать самостоятельно в пункте 6. Здесь они специально вырезаны.

А теперь запрограммируем пункты меню Undo:

```
private void undoToolStripMenuItem_Click(object sender, EventArgs e)
{
      if (History.Count != 0 && historyCounter != 0)
      {
            picDrawingSurface.Image = new Bitmap(History[--historyCounter]);
      }
      else MessageBox.Show("История пуста");
}
```

и Redo:

```
private void renoToolStripMenuItem_Click(object sender, EventArgs e)
{
      if (historyCounter < History.Count - 1)
      {
            picDrawingSurface.Image = new Bitmap(History[++historyCounter]);
      }
      else MessageBox.Show("История пуста");
}
```

8. Стиль пера.

У класса *Pen* есть несколько интересующих нас стилей: *Solid* (сплошная линия), *Dot* (линия, состоящая из точек) и *DashDotDot* (штрих-две точки). Стили будем менять в меню *Edit->Pen->Style*. Пропишем код для первого стиля *Solid*:

```
private void solidToolStripMenuItem_Click(object sender, EventArgs e)
{
      currentPen.DashStyle = DashStyle.Solid;
      solidStyleMenu.Checked = true;
      dotStyleMenu.Checked = false;
      dashDotDotStyleMenu.Checked = false;
}
```

Здесь первая строчка - задания стиля пера. Остальные 3 строчки нужны для установки галочки выбранного стиля в меню. Аналогично прописываем 2 других стиля.

Замечание.При попытке сохранить файл в любом расширении, помимо *.png упользователя будет сохранен черный квадрат. Это связано с тем, что по умолчанию PictureBox имеет прозрачный фон. Прозрачный фон поддерживает (в нашем случае) только формат png. Остальные форматы, не имеющие данной поддержки, заменяют прозрачный фон на черный. Линии, нарисованные черным пером сливаются с фоном и получается черный прямоугольник. Вот небольшой кусок кода, который может помочь решить проблему:

```
Graphics g = Graphics.FromImage(picDrawingSurface.Image);
g.Clear(Color.White);
g.DrawImage(picDrawingSurface.Image, 0, 0, 750, 500);
```

Так же есть и другие варианты исправления этой ситуации. Всё это для самостоятельного изучения и исправления.