Пользовательский интерфейс, практика

Юрий Литвинов
y.litvinov@spbu.ru

1. Рисование на формах

Рисовать всякие графические примитивы на форме не нужно (даже если очень хочется «сделать интерфейс особенным»), но может быть интересно — и уж точно полезно как учебное упражнение. Поэтому на этой практике предлагается порисовать.

Собственно, вывод графических примитивов в программах на WinForms осуществляется через API Windows, точнее его часть, которая называется GDI+ (Graphics Device Interface). Сама библиотека его использует для отрисовки контролов и предоставляет доступ через класс Graphics для пользовательского кода. Несмотря на то, что это часть WinAPI, оно работает и под Linux (и по идее под MacOS), поскольку реализовано отдельно (в libgdiplus) ещё во времена проекта Mono, и прямо из коробки работает в современном .NET.

У каждого контрола (точнее, у класса Control, родительского для всех контролов) есть событие Paint, которое инициируется, когда операционная система считает нужным перерисовать какую-то часть формы (например, при перемещении окна, изменении размеров, перемещении другого окна поверх нашего и т.п.). Обычно код рисования пишется в обработчике Paint, где из аргументов события PaintEventArgs можно получить ссылку на канву (тот самый объект Graphics) и вызывать её методы для отрисовки точек, линий, прямоугольников и т.п. Объект Graphics можно создать и самим, через метод Control.CreateGraphics — это позволяет не ждать, пока нас захотят перерисовать, а рисовать когда угодно, например, по событию таймера.

Вот небольшой пример того, как это может выглядеть. Допустим, у нас есть форма, на которой лежит какой-то наследник Control (например, Panel — она сама внешнего вида не имеет, поэтому удобна для рисования) с именем «control». Тогда можно подписать на его событие Paint такой обработчик:

```
private void ControlPaint(object sender, System.Windows.Forms.PaintEventArgs e)
{
    var font = new Font("Arial", 10);

    e.Graphics.DrawString("This is a diagonal line drawn on the control",
        font, System.Drawing.Brushes.Blue, new Point(30, 30));

    e.Graphics.DrawLine(System.Drawing.Pens.Red, control.Left, control.Top,
        control.Right, control.Bottom);
}
```

Тут мы создаём шрифт, выбирая его по имени из наличествующих в системе (так что конкретно у вас с Arial может не сложиться), рисуем синюю надпись в координатах (левого верхнего угла надписи) 30, 30 (в пикселах от левого верхнего угла рабочей области формы, то есть формы без заголовка). Дальше рисуем диагональную линию через всю форму, красным.

У Graphics есть ещё несколько разных методов, которые автодополнение покажет, и в теории этого вполне достаточно, чтобы рисовать на форме что угодно. Однако стоит обратить внимание на ещё интересные классы:

- Brush — класс, представляющий стиль заливки. Сам он абстрактный класс, имеет несколько наследников, среди которых самый ходовой — SolidBrush (сплошная заливка цветом). Ещё бывает LinearGradientBrush (заливка градиентом цветов), TextureBrush (заливка битмапом, которым можно по-разному замостить фигуру — растянуть или «замостить»). Цвет кисти в любом случае хранится как структура Color, в формате ARGB (Alpha-Red-Green-Blue). RGB кодирует цвет, альфа — это прозрачность цвета, от 0 (полностью прозрачный) до 255 (полностью непрозрачный).

- Pen — класс, представляющий стиль линий. Хранит в себе цвет линии, внезапно, в виде объекта Brush, хотя имеет и конструктор, принимающий Color.

- Bitmap — изображения и иконки. Bitmap умеет работать с нежатыми .bmp-файлами, чего для всяких мелких изображений вполне достаточно.

- Ещё у контрола есть свойство DoubleBuffered — использовать двойную буферизацию при выводе графики. Двойная буферизация — это когда изображение готовится в отдельном буфере в памяти, и как закончило рисоваться, целиком замещает текущее изображение на контроле. По идее, это уменьшает мерцание, поскольку медленное рисование графических примитивов не видно пользователю, который смотрит на предыдущий кадр, а скопировать уже готовое изображение быстро.

- Манипуляции с системой координат: методы ScaleTransform, RotateTransform, TranslateTransform у Graphics. Если, например, хочется нарисовать наклонный прямоугольник, вызовом RotateTransform можно повернуть систему координат на заданный угол и вывести прямоугольник так, будто он не наклонён — он отрисуется повёрнутым. Поворачивать систему координат можно только относительно центра, поэтому перед поворотом можно вызвать TranslateTransform, чтобы сдвинуть центр. Вообще, есть хорошее правило, что если у вас в коде отрисовки встречаются тригонометрические функции, которыми вы считаете, где должны находиться концы линий или что-то такое — скорее всего, вы просто не используете трансформации.

Вот небольшой пример работы с преобразованием координат при рисовании графических примитивов:

```
var rect = new Rectangle(0, 0, 50, 50);
var pen = new Pen(Color.FromArgb(128, 200, 0, 200), 2);
e.Graphics.ResetTransform();
e.Graphics.ScaleTransform(1.75f, 0.5f);
e.Graphics.RotateTransform(28, MatrixOrder.Append);
```

```
e.Graphics.TranslateTransform(150, 150, MatrixOrder.Append);
e.Graphics.DrawRectangle(pen, rect);
```

2. Задача

Собственно, что надо сделать за остаток практики.

- Вспомнить игру, которую мы командно писали на прошлой практике и не дописали, собраться в те команды, что были на прошлой практике.
- Реализовать вывод карты и персонажа на форму, в любом удобном виде — лучше всего стены рисовать квадратами одного цвета, свободное пространство — другого, персонажа — как хотите (кругом, например). Размер карты и размер формы стоит зафиксировать, хотя можно извратиться и сделать автомасштабирование карты (да ещё и реагирующее на изменение размера формы).
- Сделать на форме кнопки, управляющие перемещением персонажа — снизу или с боков, кнопки «вверх-вниз-влево-вправо». Используйте лейауты, а не прямое указание координат для кнопок — размер формы фиксирован, поэтому ничего никуда не уедет, но поддерживать код, если контролы позиционируются лейаутом, гораздо приятнее.
- Если успеете, поддержать и управление с клавиатуры.