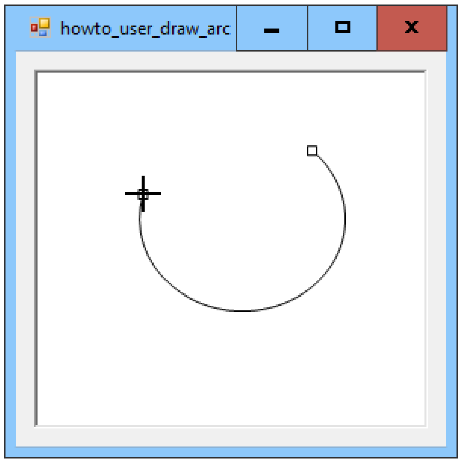
C# Helper

Tips, tricks, and example programs for C# programmers.

Let the user draw, move, and modify an arc in C#, Part 1

Posted on June 6, 2019 by RodStephens



The example [See whether a point is above an arc in C#]() shows how you can find the part of an arc at a particular position. This example uses the techniques described in that one to allow the user to draw, move, and modify and arc.

The only modifications the example allows are adjusting the arc's start and end points. It does not allow you to change the size of the arc's bounding rectangle or to rotate the arc. You can achieve those by using a selection mechanism. When the user selects an arc, display grab handles to allow the user to resize the bounding rectangle or rotate the arc. Experiment with Microsoft Word's arc shape to see how you might do that. You'll need to use a transformation to draw a rotated arc.

The previous example showed how to tell whether the mouse was over parts of an arc. This example moves that functionality into an `Arc` class.

The complete example is fairly long, so I'm breaking into two parts. This part describes the Arc class. My next post will explain how the main program uses that class to allow the user to draw and modify arcs.

All of the following code is contained in the `Arc` class.

Arc Geometry

The `Arc` class needs three pieces of information to draw an arc: a bounding rectangle, a start angle, and a sweep angle. The following code shows how the class stores that information.

```
// Geometry.
private RectangleF _Bounds;
private float _StartAngle, _SweepAngle;
public RectangleF Bounds
{
    get { return _Bounds; }
    set
    {
        _Bounds = value;
        SetArcEndPoints();
    }
}
```

```
public float StartAngle
{
    get { return _StartAngle; }
    set
    {
        _StartAngle = value;
        SetArcEndPoints();
    }
}
public float SweepAngle
{
    get { return _SweepAngle; }
    set
    {
        _SweepAngle = value;
        SetArcEndPoints();
    }
}
public float EndAngle
{
    get { return _StartAngle + _SweepAngle; }
}
public PointF Center
{
    get
    {
        return new PointF(
            Bounds.X + Bounds.Width / 2,
            Bounds.Y + Bounds.Height / 2);
    }
}
```

The private `_Bounds`, `_StartAngle`, and `_SweepAngle` fields store the basic geometric information. Property procedures let the main program get and set those values. The `set` procedures also call the `SetArcEndPoints` method described later to update the modified arc's stored end points.

The read-only `Center` property returns the center of the arc's bounding rectangle because that value is handy later.

Constructor

The class provides the following constructor.

```
public Arc(RectangleF bounds, float start_angle, float sweep_angle)
{
    _Bounds = bounds;
    _StartAngle = start_angle;
    _SweepAngle = sweep_angle;

    // Make the sweep angle positive.
    if (_SweepAngle < 0)
    {
        _StartAngle += _SweepAngle;
        _SweepAngle = -_SweepAngle;
    }

    // Make StartAngle between 0 and 360.
    while (_StartAngle < 0) _StartAngle += 360;
    while (_StartAngle > 360) _StartAngle -= 360;

    // Find the end points.
    SetArcEndPoints();
}
```

The constructor saves the arc's bounding rectangle, start angle, and sweep angle. If the sweep angle is negative, the constructor moves the start angle to the arc's first angle and reverses the sweep angle's sign. (Essentially that swaps the arc's start angle with its end angle and reverses the sweep angle's sign.)

Next the code adjusts the starting angle so it lies between 0 and 360 degrees.

Later it will be important that the start angle lies between 0 and 360 degrees and that the sweep angle is positive.

Drawing

You could just use the `Arc` class to store the arc's geometry and then make the main program use that information to draw the arc. In fact, you could still do that. However, I decided to give the `Arc` class methods to draw itself. The following code shows the class's `Draw` method.

```
    public void Draw(Graphics gr, Brush brush, Pen pen)
    {
        // If the bounding rectangle has
        // zero height or width, do nothing.
        if ((Bounds.Width == 0) || (Bounds.Height == 0))
            return;

        // Fill and draw as appropriate.
        if (brush != null)
            gr.FillPie(brush,
                Bounds.X, Bounds.Y, Bounds.Width, Bounds.Height,
                StartAngle, SweepAngle);
        if (pen != null)
            gr.DrawArc(pen,
                Bounds.X, Bounds.Y, Bounds.Width, Bounds.Height,
                StartAngle, SweepAngle);
    }
```

The `Draw` method takes as parameters a `Graphics` object on which to draw and a `Brush` and `Pen` to use when drawing the arc. The code first checks the arc's bounding rectangle and returns if the rectangle has zero width or height.

Next, if the input `Brush` is not `null`, the method fills the pie slice defined by the arc. The if the input `Pen` is not `null`, the method uses the `Graphics` object's `DrawArc` method to draw the arc.

The following code shows the class's other drawing method, `DrawEndPoints`.

```
// Draw boxes at the arc's end points.
public void DrawEndPoints(Graphics gr,
    Brush brush, Pen pen, float radius)
{
    for (int i = 0; i < 2; i++)
    {
        // Fill and draw as appropriate.
        RectangleF rect = new RectangleF(
            EndPoints[i].X - radius,
            EndPoints[i].Y - radius,
            2 * radius, 2 * radius);
        if (brush != null) gr.FillRectangle(brush, rect);
        if (pen != null)
            gr.DrawRectangle(pen,
                rect.X, rect.Y, rect.Width, rect.Height);
    }
}
```

This method draws grab handles at the arc's end points. Those end points are stored in the `EndPoints` array, which is described later.

For each end point, the method makes a rectangle centered at the end point and with diameter twice the `radius` parameter. If the input `Brush` is not null, the method fills the rectangle. Then if the input `Pen` is not null, the method draws the rectangle.

Arc Parts

The `Arc` class uses the techniques described in the previous post to determine what part of an arc lies at a particular point. It uses the following enumeration to identify the part.

```
// Pieces of the arc that the mouse might be over.
public enum Part
{
    None,
    StartPoint,
    EndPoint,
    Body,
}
```

This enumeration is defined inside the `Arc` class, so code in the main program would refer to it as in the following.

```
Arc.Part part = Arc.Part.Body;
```

The `ArcPartAtPoint` method uses the techniques described in the previous post to return the part of the arc at a given point. See that post for details.

One thing that is somewhat new is the way the class stores its end points. The following code shows how the class stores and sets the end points.

```
// Calculated when the geometry parameters change.
public PointF[] EndPoints = null;

// Find the points on an ellipse
// at the indicated angles from is center.
private void SetArcEndPoints()
{
    // Find the ellipse's center.
    PointF center = new PointF(
        Bounds.X + Bounds.Width / 2f,
        Bounds.Y + Bounds.Height / 2f);

    // If the bounding rectangle has zero
    // height or width, use the center point
    // for both end points.
    if ((Bounds.Width == 0) ||
        (Bounds.Height == 0))
    {
        EndPoints = new PointF[] { center, center };
        return;
    }

    // Find the start and end angles in radians.
    double start_radians = StartAngle * Math.PI / 180;
    double end_radians = EndAngle * Math.PI / 180;

    // Find segments from the center in the
    // desired directions and long enough to
    // cut the ellipse.
    float dist = Bounds.Width + Bounds.Height;
    PointF pt1 = new PointF(
        (float)(center.X + dist * Math.Cos(start_radians)),
        (float)(center.Y + dist * Math.Sin(start_radians)));
    PointF pt2 = new PointF(
        (float)(center.X + dist * Math.Cos(end_radians)),
        (float)(center.Y + dist * Math.Sin(end_radians)));

    // Find the points of intersection.
    PointF[] intersections1 =
        FindEllipseSegmentIntersections(
            Bounds, center, pt1, true);
    PointF[] intersections2 =
        FindEllipseSegmentIntersections(
            Bounds, center, pt2, true);
    EndPoints = new PointF[]
        {
            intersections1[0],
            intersections2[0]
        };
}
```

The `EndPoints` array holds the end points. The `SetArcEndPoints` method sets them.

The program could call `SetArcEndPoints` each time it needs the end points. For example, the `ArcPartAtPoint` method could call the method. However, I assume that the program might call `ArcPartAtPoint` many times. When you move the mouse around in the example program, it calls `ArcPartAtPoint` every time the mouse moves. If that method called `SetArcEndPoints`. it would slow things down.

To avoid repeated calls to `SetArcEndPoints`, the program calls that method when one of the arc's properties changes and then the value is stored for later use.

Modifying the Arc

Here's where the code that lets the user modify the arc begins. The following code shows the `Move` method.

```
// Move the Arc.
public void Move(int dx, int dy)
{
    Bounds = new RectangleF(
        Bounds.X + dx,
        Bounds.Y + dy,
        Bounds.Width, Bounds.Height);
}
```

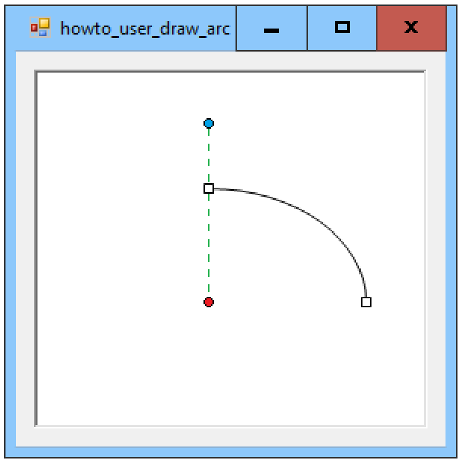
This method takes parameters that indicate how far the arc should be moved in the X and Y directions. It simply sets the arc's `Bounds` property to a new rectangle with the same width and height as the old one but shifted by the appropriate amount.

Note that the `Bounds` property set procedure calls `SetArcEndPoints`, so the end points are automatically updated.

The following code shows the `MoveStartPoint` method, which allows the program to adjust the arc's starting end point.

```
// Move the start point.
public void MoveStartPoint(PointF point)
{
    // Fond the angle the point makes with the center.
    PointF center = Center;
    float dx = point.X - center.X;
    float dy = point.Y - center.Y;

    // If the point is at the center, do nothing.
    if ((dx == 0) && (dy == 0)) return;

    float start_angle =
        (float)(Math.Atan2(dy, dx) * 180 / Math.PI);
    if (start_angle < 0) start_angle += 360;

    // Calculate the end angle.
    float end_angle = StartAngle + SweepAngle;

    // Make sure
```

```
        // start_angle <= end_angle <= start_angle + 360
        while (end_angle < start_angle)
            end_angle += 360;
        while (end_angle > start_angle + 360)
            end_angle -= 360;

        // Calculate the sweep angle needed to
        // get to the ending point.
        float sweep_angle = end_angle - start_angle;

        StartAngle = start_angle;
        SweepAngle = sweep_angle;
    }
```



This method updates the start angle to point from the center of the arc's bounding rectangle toward the indicated point. In the picture on the right, the center point is red and the mouse's position is green. The new start point is where the dashed line intersects the ellipse that defines the arc.

The `MoveStartPoint` method first gets the X and Y differences between the given point and the arc's center. If the two points are the same, then the dashed line in the picture is not defined so the method returns without doing anything.

Next, the method calculates the angle that the dashed line makes with respect to the horizontal. That will be the new start angle. The `Math.Atan2` method returns an angle in radians between -π and +π, and we convert that into and angle in degrees between -180 and +180. The program then adds 360 degrees if necessary to ensure that the new start angle lies between 0 and 360 degrees.

Next, the code adds or subtracts 360 degrees from the end angle until that angle lies between start_angle and the start_angle + 360.

Finally, the method calculates the sweep angle between the new start and end angles, and then updates the arc's `StartAngle` and `SweepAngle` properties.

The last piece of the `Arc` class is the following `MoveEndPoint` method, which allows the program to adjust the arc's ending point.

```
    // Move the end point.
    public void MoveEndPoint(PointF point)
    {
        // Fond the angle the point makes with the center.
        PointF center = Center;
        float dx = point.X - center.X;
        float dy = point.Y - center.Y;

        // If the point is at the center, do nothing.
        if ((dx == 0) && (dy == 0)) return;
        // Calculate the end angle.
        float end_angle =
            (float)(Math.Atan2(dy, dx) * 180 / Math.PI);

        // Make sure
        // start_angle <= end_angle <= start_angle + 360
        while (end_angle < StartAngle)
            end_angle += 360;
        while (end_angle > StartAngle + 360)
            end_angle -= 360;

        // Calculate the sweep angle.
        SweepAngle = end_angle - StartAngle;
    }
```

This method calculates the angle with respect to the arc's center much as the `MoveStartPoint` method did. It then adjusts the new end angle so it lies between the start angle and the start angle plus 360. The method finishes by setting the arc's `SweepAngle` property to the difference between the ending and starting angles.

Summary

The `Arc` class provides the following main public methods for use by the rest of the program.

- Arc – The constructor
- Draw – Draws the arc
- DrawEndPoints – Draws grab handles at the arc's end points
- Move – Moves the whole arc
- MoveStartPoint – Moves the arc's starting point
- MoveEndPoint – Moves the arc's ending point

My next post will explain how the example program uses the `Arc` class to allow the user to draw, move, and modify arcs. Meanwhile, download the example and try it out. You'll find that the `MoveStartPoint` and `MoveEndPoint` methods allow you to adjust the arc's starting and ending points easily and intuitively.

Download

Donate