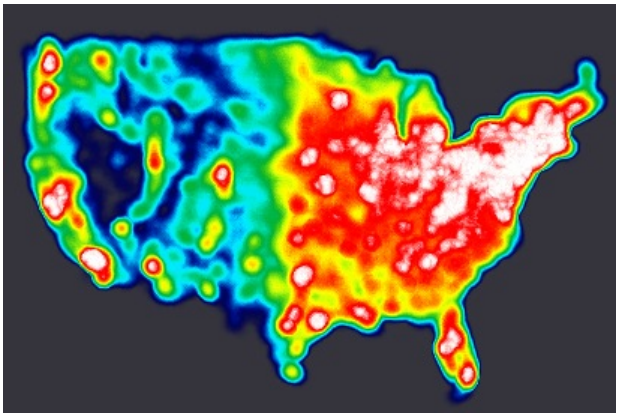
Creating Heat Maps with .NET 2.0 (C#)

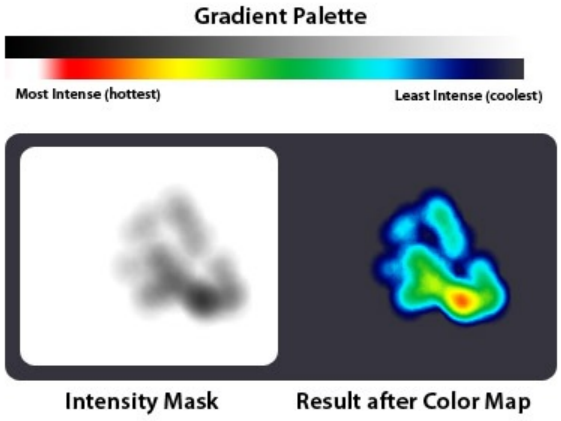


It seems now a days that heat maps are everywhere, from visitor mouse tracking on your website, to geographic density mapping. Heat maps are earning this well deserved deployment primarily do to the fact that they are a powerful visualization tool for three dimensional data (two for plotting, on for density) and nearly everyone has seen them depicted in movies. In this post I would like to show you how easy it is to create your very own heat ma using the power of the GDI+ in the .NET framework.

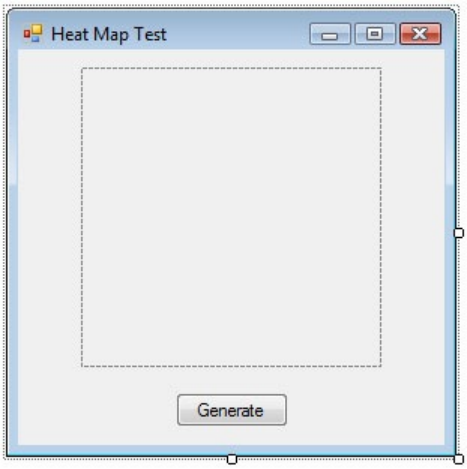
I'd first like to point out that for this first post we'll only be discussing plotting points in screen space, that being the X and Y coordinates of your screen. To do something like geographic plotting you would need an additional step to translate the longitude and latitude coordinates to screen coordinates. O now lets move onto the fun stuff.

Creating the Intensity Mask

Creating a heat map in its simplest form is a two step process. The first step is to create an intensity mask. An intensity mask is basically a grayscale (25 shades of gray) image that we use to mark the intensity of every pixel in our final image. Each shade in the white to black spectrum will map directly to a actual color in our gradient palette. Take a look at the following image to see what I mean.



As you can see from the image above the darker the pixel in the intensity mask the hotter the color is that gets mapped to it. Ok, so now that you know what an intensity mask is, let me show you how to create one. The first thing you need to do is create a new Windows Application project in Visual Studio. On the default form, create a new picture box and a button.



Before we put any code against this this form we first need to define a struct which will hold our most basic building block for a heat map; the heat point A heat point is very similar to a regular point in the .NET GDI accept for the fact that it will contain an additional property for storing the density. It can be argued that this additional property is not necessary since heat points are additive by nature, meaning that when stacked on top of each other the area they occupy will already appear denser. However, the reason I believe this property is necessary is for performance reasons. It's quite a bit less overhead to generate a "denser" heat point than it is to generate a bunch of less dense heat points, plus this means you won't have to store as many points since you'll be feeding the heat mapping class data that is pre-aggregated.

```
public struct HeatPoint
{
    public int X;
    public int Y;
    public byte Intensity;
    public HeatPoint(int iX, int iY, byte bIntensity)
    {
        X = iX;
        Y = iY;
        Intensity = bIntensity;
    }
}
```

Now that we have a struct to hold heat point data. Lets create a generic list to hold a set of heat points which will use to plot on our surface as our intensity mask. Add this private variable to the form class.

```
private List<HeatPoint> HeatPoints = new List<HeatPoint>();
```

Next we'll create a method to generate the actual intensity mask. This method will accept a memory bitmap object and a generic list of heat points and return a new memory bitmap. Here is what it looks like.

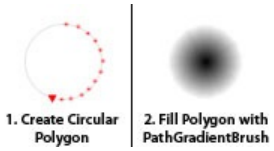
```
private Bitmap CreateIntensityMask(Bitmap bSurface, List<HeatPoint> aHeatPoints)
{
    // Create new graphics surface from memory bitmap
    Graphics DrawSurface = Graphics.FromImage(bSurface);
    // Set background color to white so that pixels can be correctly colorized
    DrawSurface.Clear(Color.White);

    // Traverse heat point data and draw masks for each heat point
    foreach (HeatPoint DataPoint in aHeatPoints)
    {
        // Render current heat point on draw surface
        DrawHeatPoint(DrawSurface, DataPoint, 25);
    }

    return bSurface;
}
```

I'd like you to focus your attentions on the method DrawHeatPoint. This method is used to draw an actual radial gradient "spot" on the drawing surface. It's perhaps the most important method in this entire project as it handles drawing spots of varying size and density. Please read through the comments to get an idea for how this crucial method works.

```
private void DrawHeatPoint(Graphics Canvas, HeatPoint HeatPoint, int Radius)
{
    // Create points generic list of points to hold circumference points
    List<Point> CircumferencePointsList = new List<Point>();

    // Create an empty point to predefine the point struct used in the circumference loop
    Point CircumferencePoint;

    // Create an empty array that will be populated with points from the generic list
    Point[] CircumferencePointsArray;

    // Calculate ratio to scale byte intensity range from 0-255 to 0-1
    float fRatio = 1F / Byte.MaxValue;
    // Precalulate half of byte max value
    byte bHalf = Byte.MaxValue / 2;
    // Flip intensity on it's center value from low-high to high-low
    int iIntensity = (byte)(HeatPoint.Intensity - ((HeatPoint.Intensity - bHalf) * 2));
    // Store scaled and flipped intensity value for use with gradient center location
    float fIntensity = iIntensity * fRatio;

    // Loop through all angles of a circle
    // Define loop variable as a double to prevent casting in each iteration
    // Iterate through loop on 10 degree deltas, this can change to improve performance
    for (double i = 0; i <= 360; i += 10)
    {
```

```
            // Replace last iteration point with new empty point struct
            CircumferencePoint = new Point();

            // Plot new point on the circumference of a circle of the defined radius
            // Using the point coordinates, radius, and angle
            // Calculate the position of this iterations point on the circle
            CircumferencePoint.X = Convert.ToInt32(HeatPoint.X + Radius * Math.Cos(ConvertDegreesToRadians(i)));
            CircumferencePoint.Y = Convert.ToInt32(HeatPoint.Y + Radius * Math.Sin(ConvertDegreesToRadians(i)));

            // Add newly plotted circumference point to generic point list
            CircumferencePointsList.Add(CircumferencePoint);
        }

        // Populate empty points system array from generic points array list
        // Do this to satisfy the datatype of the PathGradientBrush and FillPolygon methods
        CircumferencePointsArray = CircumferencePointsList.ToArray();

        // Create new PathGradientBrush to create a radial gradient using the circumference points
        PathGradientBrush GradientShaper = new PathGradientBrush(CircumferencePointsArray);
        // Create new color blend to tell the PathGradientBrush what colors to use and where to put them
        ColorBlend GradientSpecifications = new ColorBlend(3);

        // Define positions of gradient colors, use intesity to adjust the middle color to
        // show more mask or less mask
        GradientSpecifications.Positions = new float[3] { 0, fIntensity, 1 };
        // Define gradient colors and their alpha values, adjust alpha of gradient colors to match intensity
        GradientSpecifications.Colors = new Color[3]
        {
            Color.FromArgb(0, Color.White),
            Color.FromArgb(HeatPoint.Intensity, Color.Black),
            Color.FromArgb(HeatPoint.Intensity, Color.Black)
        };

        // Pass off color blend to PathGradientBrush to instruct it how to generate the gradient
        GradientShaper.InterpolationColors = GradientSpecifications;
        // Draw polygon (circle) using our point array and gradient brush
        Canvas.FillPolygon(GradientShaper, CircumferencePointsArray);
    }
```

One thing I would like to talk about in this method is the FOR loop. The reason for it's existence is to allow the GDI to produce a radial gradient, because unfortunately there is no easy way to produce a variable radial gradient using the GDI in .NET 2.0. The way my method works is by creating a new polygon object and adding points to it that fall along the circumference of heat point. The circumference is calculated using the radius passed to the DrawHeatPoint method. The rest of the code in the method is used to adjust the ratio of white and black in the gradient brush that we use to fill the polygon generated in the FOR loop.



A much simpler method of drawing a heat point would be to simply have a semi-transparent png image the mimics the black radial gradient that you see above. However, the problem with doing it like that is that you lose the ability to dynamically adjust the size and density of the heat point without a lose in quality and flexibility.

Also, don't forget this little guy because the DrawHeatPoint method is dependent on him.

```
private double ConvertDegreesToRadians(double degrees)
{
    double radians = (Math.PI / 180) * degrees;
    return (radians);
}
```

Ok, so the last thing we need to do to create our intensity mask is to create a button click event handler and sprinkle enough code in it to force it to generate an intensity mask. Here is what the handler should look like.

```
private void button1_Click(object sender, EventArgs e)
{
    // Create new memory bitmap the same size as the picture box
    Bitmap bMap = new Bitmap(pictureBox1.Width, pictureBox1.Height);

    // Initialize random number generator
    Random rRand = new Random();

    // Loop variables
    int iX;
    int iY;
    byte iIntense;

    // Lets loop 500 times and create a random point each iteration
    for (int i = 0; i < 500; i++)
```

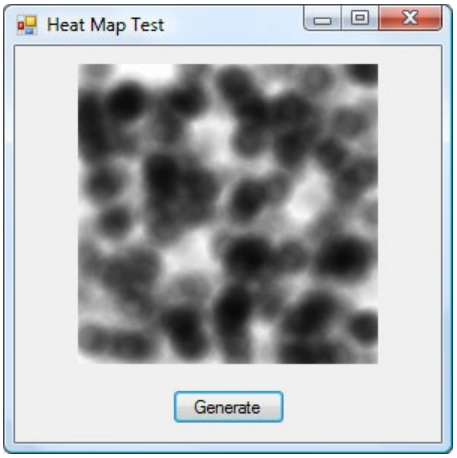
```
    {
        // Pick random locations and intensity
        iX = rRand.Next(0, 200);
        iY = rRand.Next(0, 200);
        iIntense = (byte)rRand.Next(0, 120);

        // Add heat point to heat points list
        HeatPoints.Add(new HeatPoint(iX, iY, iIntense));
    }

    // Call CreateIntensityMask, give it the memory bitmap, and use it's output to set the picture box image
    pictureBox1.Image = CreateIntensityMask(bMap, HeatPoints);
}
```

As you can see by the comments we are generating 500 random heat points as test data so that you will actually have something to look at when the heat map is generated. This is what your code file should look like for your form.

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

public partial class Form1 : Form
{
    private List<HeatPoint> HeatPoints = new List<HeatPoint>();

    public Form1()
    {
        InitializeComponent();
    }

    private void button1_Click(object sender, EventArgs e)
    {
        // Create new memory bitmap the same size as the picture box
        Bitmap bMap = new Bitmap(pictureBox1.Width, pictureBox1.Height);

        // Initialize random number generator
        Random rRand = new Random();

        // Loop variables
        int iX;
        int iY;
        byte iIntense;

        // Lets loop 500 times and create a random point each iteration
        for (int i = 0; i < 500; i++)
        {
            // Pick random locations and intensity
            iX = rRand.Next(0, 200);
            iY = rRand.Next(0, 200);
            iIntense = (byte)rRand.Next(0, 120);
            // Add heat point to heat points list
            HeatPoints.Add(new HeatPoint(iX, iY, iIntense));
        }

        // Call CreateIntensityMask, give it the memory bitmap, and use it's output to set the picture box image
        pictureBox1.Image = CreateIntensityMask(bMap, HeatPoints);
    }

    private Bitmap CreateIntensityMask(Bitmap bSurface, List<HeatPoint> aHeatPoints)
    {
        // Create new graphics surface from memory bitmap
        Graphics DrawSurface = Graphics.FromImage(bSurface);

        // Set background color to white so that pixels can be correctly colorized
        DrawSurface.Clear(Color.White);

        // Traverse heat point data and draw masks for each heat point
        foreach (HeatPoint DataPoint in aHeatPoints)
        {
            // Render current heat point on draw surface
            DrawHeatPoint(DrawSurface, DataPoint, 15);
        }

        return bSurface;
    }

    private void DrawHeatPoint(Graphics Canvas, HeatPoint HeatPoint, int Radius)
    {
        // Create points generic list of points to hold circumference points
        List<Point> CircumferencePointsList = new List<Point>();

        // Create an empty point to predefine the point struct used in the circumference loop
        Point CircumferencePoint;

        // Create an empty array that will be populated with points from the generic list
        Point[] CircumferencePointsArray;
```

```
        // Calculate ratio to scale byte intensity range from 0-255 to 0-1
        float fRatio = 1F / Byte.MaxValue;
        // Precalulate half of byte max value
        byte bHalf = Byte.MaxValue / 2;
        // Flip intensity on it's center value from low-high to high-low
        int iIntensity = (byte)(HeatPoint.Intensity - ((HeatPoint.Intensity - bHalf) * 2));
        // Store scaled and flipped intensity value for use with gradient center location
        float fIntensity = iIntensity * fRatio;

        // Loop through all angles of a circle
        // Define loop variable as a double to prevent casting in each iteration
        // Iterate through loop on 10 degree deltas, this can change to improve performance
        for (double i = 0; i <= 360; i += 10)
        {
            // Replace last iteration point with new empty point struct
            CircumferencePoint = new Point();
            // Plot new point on the circumference of a circle of the defined radius
            // Using the point coordinates, radius, and angle
            // Calculate the position of this iterations point on the circle
            CircumferencePoint.X = Convert.ToInt32(HeatPoint.X + Radius * Math.Cos(ConvertDegreesToRadians(i)));
            CircumferencePoint.Y = Convert.ToInt32(HeatPoint.Y + Radius * Math.Sin(ConvertDegreesToRadians(i)));
            // Add newly plotted circumference point to generic point list
            CircumferencePointsList.Add(CircumferencePoint);
        }

        // Populate empty points system array from generic points array list
        // Do this to satisfy the datatype of the PathGradientBrush and FillPolygon methods
        CircumferencePointsArray = CircumferencePointsList.ToArray();

        // Create new PathGradientBrush to create a radial gradient using the circumference points
        PathGradientBrush GradientShaper = new PathGradientBrush(CircumferencePointsArray);
        // Create new color blend to tell the PathGradientBrush what colors to use and where to put them
        ColorBlend GradientSpecifications = new ColorBlend(3);

        // Define positions of gradient colors, use intesity to adjust the middle color to
        // show more mask or less mask
        GradientSpecifications.Positions = new float[3] { 0, fIntensity, 1 };
        // Define gradient colors and their alpha values, adjust alpha of gradient colors to match intensity
        GradientSpecifications.Colors = new Color[3]
        {
            Color.FromArgb(0, Color.White),
            Color.FromArgb(HeatPoint.Intensity, Color.Black),
            Color.FromArgb(HeatPoint.Intensity, Color.Black)
        };

        // Pass off color blend to PathGradientBrush to instruct it how to generate the gradient
        GradientShaper.InterpolationColors = GradientSpecifications;
        // Draw polygon (circle) using our point array and gradient brush
        Canvas.FillPolygon(GradientShaper, CircumferencePointsArray);
    }

    private double ConvertDegreesToRadians(double degrees)
    {
        double radians = (Math.PI / 180) * degrees;
        return (radians);
    }
}

public struct HeatPoint
{
    public int X;
    public int Y;
    public byte Intensity;
    public HeatPoint(int iX, int iY, byte bIntensity)
    {
        X = iX;
        Y = iY;
        Intensity = bIntensity;
    }
}
```

Ok, now fire this thing up and click the button on your form. You should get an output that looks similar to this.



You've done it, you've created an intensity mask which is the biggest hurdle in generating a heat map. In this sample all we did was generate a bunch o nonsense test data. But at this point you should be able to feed it your own meaningful data and have it visualize the numbers. You didn't come here to end up with a bunch of black blobs though. You came for a majestically full color visualization masterpiece, and believe it or not, that's the easy part!

Colorizing the Intensity Mask

Ok, so this step is cake, because all we have to do now is create a color map that tells the GDI rendering method how it should draw the colors. What we're actually doing here is creating a table that specifies a new color for each shade of gray (all 256) of them. The easiest way to do this is to create a palette image that's 256 pixels wide by 1 pixel high. That means that we can directly map each pixel in the palette image to a shade of gray. I'll let you have my palette image since it can be sort of a pain to create one that looks nice.

So this brings me to the last two methods we'll need to generate our final image. One to do the colorization, and one to create a color remap table.

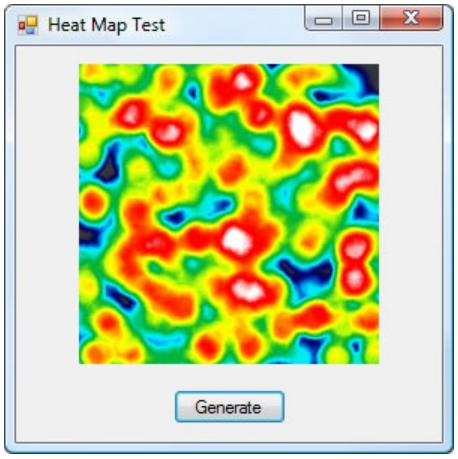
```
public static Bitmap Colorize(Bitmap Mask, byte Alpha)
{
    // Create new bitmap to act as a work surface for the colorization process
    Bitmap Output = new Bitmap(Mask.Width, Mask.Height, PixelFormat.Format32bppArgb);

    // Create a graphics object from our memory bitmap so we can draw on it and clear it's drawing surface
    Graphics Surface = Graphics.FromImage(Output);
    Surface.Clear(Color.Transparent);

    // Build an array of color mappings to remap our greyscale mask to full color
    // Accept an alpha byte to specify the transparancy of the output image
    ColorMap[] Colors = CreatePaletteIndex(Alpha);

    // Create new image attributes class to handle the color remappings
    // Inject our color map array to instruct the image attributes class how to do the colorization
    ImageAttributes Remapper = new ImageAttributes();
    Remapper.SetRemapTable(Colors);

    // Draw our mask onto our memory bitmap work surface using the new color mapping scheme
    Surface.DrawImage(Mask, new Rectangle(0, 0, Mask.Width, Mask.Height), 0, 0, Mask.Width, Mask.Height, GraphicsUnit.Pixel, Remapper);

    // Send back newly colorized memory bitmap
    return Output;
}

private static ColorMap[] CreatePaletteIndex(byte Alpha)
{
    ColorMap[] OutputMap = new ColorMap[256];

    // Change this path to wherever you saved the palette image.
    Bitmap Palette = (Bitmap)Bitmap.FromFile(@"C:\Users\Dylan\Documents\Visual Studio 2005\Projects\HeatMapTest\palette.bmp");

    // Loop through each pixel and create a new color mapping
    for (int X = 0; X <= 255; X++)
    {
        OutputMap[X] = new ColorMap();
        OutputMap[X].OldColor = Color.FromArgb(X, X, X);
        OutputMap[X].NewColor = Color.FromArgb(Alpha, Palette.GetPixel(X, 0));
    }

    return OutputMap;
}
```

Now with just a slight modification to the button click event handler we'll include the ability to colorize the image and we should be golden. Here is what the new button click event handler will look like.

```
private void button1_Click(object sender, EventArgs e)
```

```
{
    // Create new memory bitmap the same size as the picture box
    Bitmap bMap = new Bitmap(pictureBox1.Width, pictureBox1.Height);

    // Initialize random number generator
    Random rRand = new Random();

    // Loop variables
    int iX;
    int iY;
    byte iIntense;

    // Lets loop 500 times and create a random point each iteration
    for (int i = 0; i < 500; i++)
    {
        // Pick random locations and intensity
        iX = rRand.Next(0, 200);
        iY = rRand.Next(0, 200);
        iIntense = (byte)rRand.Next(0, 120);

        // Add heat point to heat points list
        HeatPoints.Add(new HeatPoint(iX, iY, iIntense));
    }

    // Call CreateIntensityMask, give it the memory bitmap, and store the result back in the memory bitmap
    bMap = CreateIntensityMask(bMap, HeatPoints);

    // Colorize the memory bitmap and assign it as the picture boxes image
    pictureBox1.Image = Colorize(bMap, 255);
}
```

Go ahead and run the project and click the button on the form and you should get a sexy little heat map.



Congratulations on generating your first heat map with .NET 2.0! Please feel free to post any comments about my code or my writing below. Thanks, and finally here is what the whole code file looks like when completed.

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Imaging;
using System.Windows.Forms;

public partial class Form1 : Form
{
    private List<HeatPoint> HeatPoints = new List<HeatPoint>();

    public Form1()
    {
        InitializeComponent();
    }

    private void button1_Click(object sender, EventArgs e)
    {
        // Create new memory bitmap the same size as the picture box
        Bitmap bMap = new Bitmap(pictureBox1.Width, pictureBox1.Height);

        // Initialize random number generator
        Random rRand = new Random();

        // Loop variables
        int iX;
        int iY;
        byte iIntense;

        // Lets loop 500 times and create a random point each iteration
        for (int i = 0; i < 500; i++)
```

```
        {
            // Pick random locations and intensity
            iX = rRand.Next(0, 200);
            iY = rRand.Next(0, 200);
            iIntense = (byte)rRand.Next(0, 120);

            // Add heat point to heat points list
            HeatPoints.Add(new HeatPoint(iX, iY, iIntense));
        }

        // Call CreateIntensityMask, give it the memory bitmap, and store the result back in the memory bitmap
        bMap = CreateIntensityMask(bMap, HeatPoints);

        // Colorize the memory bitmap and assign it as the picture boxes image
        pictureBox1.Image = Colorize(bMap, 255);
    }

    private Bitmap CreateIntensityMask(Bitmap bSurface, List<HeatPoint> aHeatPoints)
    {
        // Create new graphics surface from memory bitmap
        Graphics DrawSurface = Graphics.FromImage(bSurface);

        // Set background color to white so that pixels can be correctly colorized
        DrawSurface.Clear(Color.White);

        // Traverse heat point data and draw masks for each heat point
        foreach (HeatPoint DataPoint in aHeatPoints)
        {
            // Render current heat point on draw surface
            DrawHeatPoint(DrawSurface, DataPoint, 15);
        }

        return bSurface;
    }

    private void DrawHeatPoint(Graphics Canvas, HeatPoint HeatPoint, int Radius)
    {
        // Create points generic list of points to hold circumference points
        List<Point> CircumferencePointsList = new List<Point>();

        // Create an empty point to predefine the point struct used in the circumference loop
        Point CircumferencePoint;

        // Create an empty array that will be populated with points from the generic list
        Point[] CircumferencePointsArray;

        // Calculate ratio to scale byte intensity range from 0-255 to 0-1
        float fRatio = 1F / Byte.MaxValue;
        // Precalulate half of byte max value
        byte bHalf = Byte.MaxValue / 2;
        // Flip intensity on it's center value from low-high to high-low
        int iIntensity = (byte)(HeatPoint.Intensity - ((HeatPoint.Intensity - bHalf) * 2));
        // Store scaled and flipped intensity value for use with gradient center location
        float fIntensity = iIntensity * fRatio;

        // Loop through all angles of a circle
        // Define loop variable as a double to prevent casting in each iteration
        // Iterate through loop on 10 degree deltas, this can change to improve performance
        for (double i = 0; i <= 360; i += 10)
        {
            // Replace last iteration point with new empty point struct
            CircumferencePoint = new Point();

            // Plot new point on the circumference of a circle of the defined radius
            // Using the point coordinates, radius, and angle
            // Calculate the position of this iterations point on the circle
            CircumferencePoint.X = Convert.ToInt32(HeatPoint.X + Radius * Math.Cos(ConvertDegreesToRadians(i)));
            CircumferencePoint.Y = Convert.ToInt32(HeatPoint.Y + Radius * Math.Sin(ConvertDegreesToRadians(i)));

            // Add newly plotted circumference point to generic point list
            CircumferencePointsList.Add(CircumferencePoint);
        }

        // Populate empty points system array from generic points array list
        // Do this to satisfy the datatype of the PathGradientBrush and FillPolygon methods
        CircumferencePointsArray = CircumferencePointsList.ToArray();

        // Create new PathGradientBrush to create a radial gradient using the circumference points
        PathGradientBrush GradientShaper = new PathGradientBrush(CircumferencePointsArray);

        // Create new color blend to tell the PathGradientBrush what colors to use and where to put them
        ColorBlend GradientSpecifications = new ColorBlend(3);

        // Define positions of gradient colors, use intesity to adjust the middle color to
        // show more mask or less mask
        GradientSpecifications.Positions = new float[3] { 0, fIntensity, 1 };
        // Define gradient colors and their alpha values, adjust alpha of gradient colors to match intensity
        GradientSpecifications.Colors = new Color[3]
        {
            Color.FromArgb(0, Color.White),
            Color.FromArgb(HeatPoint.Intensity, Color.Black),
            Color.FromArgb(HeatPoint.Intensity, Color.Black)
        };
```

```
        // Pass off color blend to PathGradientBrush to instruct it how to generate the gradient
        GradientShaper.InterpolationColors = GradientSpecifications;

        // Draw polygon (circle) using our point array and gradient brush
        Canvas.FillPolygon(GradientShaper, CircumferencePointsArray);
    }

    private double ConvertDegreesToRadians(double degrees)
    {
        double radians = (Math.PI / 180) * degrees;
        return (radians);
    }

    public static Bitmap Colorize(Bitmap Mask, byte Alpha)
    {
        // Create new bitmap to act as a work surface for the colorization process
        Bitmap Output = new Bitmap(Mask.Width, Mask.Height, PixelFormat.Format32bppArgb);

        // Create a graphics object from our memory bitmap so we can draw on it and clear it's drawing surface
        Graphics Surface = Graphics.FromImage(Output);
        Surface.Clear(Color.Transparent);

        // Build an array of color mappings to remap our greyscale mask to full color
        // Accept an alpha byte to specify the transparancy of the output image
        ColorMap[] Colors = CreatePaletteIndex(Alpha);

        // Create new image attributes class to handle the color remappings
        // Inject our color map array to instruct the image attributes class how to do the colorization
        ImageAttributes Remapper = new ImageAttributes();
        Remapper.SetRemapTable(Colors);

        // Draw our mask onto our memory bitmap work surface using the new color mapping scheme
        Surface.DrawImage(Mask, new Rectangle(0, 0, Mask.Width, Mask.Height), 0, 0, Mask.Width, Mask.Height, GraphicsUnit.Pixel, Remapper

        // Send back newly colorized memory bitmap
        return Output;
    }

    private static ColorMap[] CreatePaletteIndex(byte Alpha)
    {
        ColorMap[] OutputMap = new ColorMap[256];

        // Change this path to wherever you saved the palette image.
        Bitmap Palette = (Bitmap)Bitmap.FromFile(@"C:\Users\Dylan\Documents\Visual Studio 2005\Projects\HeatMapTest\palette.bmp");

        // Loop through each pixel and create a new color mapping
        for (int X = 0; X <= 255; X++)
        {
            OutputMap[X] = new ColorMap();
            OutputMap[X].OldColor = Color.FromArgb(X, X, X);
            OutputMap[X].NewColor = Color.FromArgb(Alpha, Palette.GetPixel(X, 0));
        }

        return OutputMap;
    }
}

public struct HeatPoint
{
    public int X;
    public int Y;
    public byte Intensity;
    public HeatPoint(int iX, int iY, byte bIntensity)
    {
        X = iX;
        Y = iY;
        Intensity = bIntensity;
    }
}
```

Currently rated 4.8 by 31 people

Posted on March 3, 2008 16:20 by Dylan
Tags: heat map, visualization, csharp, code sample
Categories: Heat Map
Actions: E-mail | Permalink | Comments (20) | Trackback
