How to implement gradient fills using shadings and shading patterns in PDF

Written by Apitron Documentation Team

Introduction

PDF is not only about text data storage, but it also specifies a rich set of drawing operations making it possible to achieve very complex visual effects. In this article we'll talk about its gradient filling and stroking capabilities. Using the API provided by Apitron PDF Kit for .NET component you'll be able to create advanced color gradient fills based on standard linear and exponential interpolation functions as well as pure PostScript function-based color transition effects. In PDF, the object used to create a complex fill is a called a Shading, and you may read the low level details involved in article *8.7.4 Shading Patterns* in PDF specification.

The Shading object requires a function object that defines the color transformation needed to create the final color at the each point where the shading is defined. Functions are described in section *7.10 Functions* of the PDF specification. Using various function types it becomes possible to implement linear, bilinear and exponential interpolation as well as other color effects. You can use different functions to calculate the value for color components independently or at once. Currently available function types are: *Sampled*, *Exponential*, *StitchingFunction* and *PostScript.* Shading types available are: axial, radial and function-based.

When the area to be painted is a relatively simple shape whose geometry is the same as that of the gradient fill itself or you have a simple clipping path for it, the *ClippedContent::PaintShading()* function may be used instead of the usual painting approach. It accepts a *Shading* object as an operand and applies the corresponding gradient fill directly to current user space applying the additional clipping path given as a parameter if needed.

Another way to use the shading is to define a *ShadingPattern*. Similar to the *TilingPattern* object it can be used as a color for fill and stroke operations if you set the current stroking or non-stroking colorspace as *PredefinedColorpaces::Pattern.* This way you'll be able to draw whatever you want using the shading as color provider for stroked or filled point.

Both methods have own pros and cons and it's up to you to choose the one that works best it your situation. Read further to see various functions types and shadings types in action.

Axial Shading based on Exponential interpolation function

Here we define the axial shading object that does what it name says - defines a color blend along a line between two points, optionally extended beyond the boundary points by continuing the boundary colors. Exponential interpolation function is being used to calculate color transition. The formula used to produce each color sample is $y_j = C0_j + x^N \times (C1_j - C0_j)$, and input and output values are limited by the *domain* and *range* intervals ([0,1] in this case).

```
// creates and registers axial shading object based on exponential function
private static Shading CreateAndRegisterAxialShadingBasedOnExponentialFunction(FixedDocument doc,
    Color beginColor, Color endColor)
{
    // exponential function producing the gradient
    Function expFn = new ExponentialFunction(Guid.NewGuid().ToString(), beginColor.Components,
       endColor.Components, 3, new double[] { 0, 1 });

    // axial shading demonstrating exponential interpolation between two colors
    AxialShading axialShadingExp = new AxialShading( Guid.NewGuid().ToString(),
        PredefinedColorSpaces.RGB, new Boundary(0, 0, 180, 180), RgbColors.Green.Components,
        new double[] { 0, 90, 180, 90 }, new string[] { expFn.ID });

    doc.ResourceManager.RegisterResource(expFn);
    doc.ResourceManager.RegisterResource(axialShadingExp);

    return axialShadingExp;
}
```

This shading is defined using rectangular boundary [0, 0,180,180] and two points which define the interpolation axis (0,90) and (180,90).

Axial Shading based on Sampled Function

The shading object created here is based on sampled function that uses *SamplerDelegate* to produce the color value for each calculated sample value. Samples count is the number of color passed as the parameter.

```
// creates and registers axial shading object based on sampled function able to interpolate
// between multiple colors private static AxialShading
CreateAndRegisterAxialShadingBasedOnLinearInterpolationFunction(
    FixedDocument doc, params object[] colors)
{
    int samplesCount = colors.Length;
    double[] domain = new double[] {0, 1};
    double step = domain[1] / (samplesCount-1);

    // create the delegate returning color value for corresponding sample
    SamplerDelegate fn = (double[] input) => {
                int k = 0;
                double tmpStep = step;
                while (input[0] >= tmpStep)
                {
                    tmpStep += step;
                    ++k;
                }
                return (colors[k] as Color).Components;
            };

    // linear sampled function producing the gradient
    Function linearFn = new SampledFunction(Guid.NewGuid().ToString(), fn, domain ,
        new double[] { 0, 1, 0, 1, 0, 1 }, new[] { samplesCount }, BitsPerSample.OneByte);

    // axial shading demonstrating linear interpolation between two colors
    AxialShading axialShadingLinear = new AxialShading(Guid.NewGuid().ToString(),
        PredefinedColorSpaces.RGB, new Boundary(0, 0, 180, 180), RgbColors.Green.Components,
        new double[] { 0, 90, 180, 90 }, new string[] { linearFn.ID });

    doc.ResourceManager.RegisterResource(linearFn);
    doc.ResourceManager.RegisterResource(axialShadingLinear);

    return axialShadingLinear;
}
```

This shading is defined using rectangular boundary [0, 0,180,180] and two points which define the interpolation axis (0,90) and (180,90).

Function-based shading with PostScript function

In function-based shadings, the color at every point in the domain is defined by a specified mathematical function. The function need not be smooth or continuous. This type is the most general of the available shading types and is useful for shadings that cannot be adequately described with any of the other types. We used a separate PostScript function for each of the color components to produce the final color.

```
// creates and registers function-based shading object
private static Shading CreateAndRegisterFunctionBasedShading(FixedDocument doc)
{
    // create post script functions for each color component
    // it's possible to use only for one function and calculate all components at once
    // the function used for R is 1- ((x-90)/90)^2 + ((y-90)/90)^2)
    PostScriptFunction psFunctionR = new PostScriptFunction(Guid.NewGuid().ToString(),
        new double[] {0, 180, 0, 180},
        new double[] {0, 1}, "{90 sub 90 div dup mul exch 90 sub 90 div dup mul add 1 exch sub}");
    // the function used for G is 1- ((x-90)/60)^2 + ((y-90)/60)^2)
    PostScriptFunction psFunctionG = new PostScriptFunction(Guid.NewGuid().ToString(),
        new double[] {0, 180, 0, 180},
        new double[] {0, 1}, "{90 sub 60 div dup mul exch 90 sub 60 div dup mul add 1 exch sub}");
    // the function used for G is 1- ((x-90)/30)^2 + ((y-90)/30)^2)
    PostScriptFunction psFunctionB = new PostScriptFunction(Guid.NewGuid().ToString(),
        new double[] {0, 180, 0, 180},
        new double[] {0, 1}, "{90 sub 30 div dup mul exch 90 sub 30 div dup mul add 1 exch sub}");

    // create shading based on three functions
    FunctionShading functionShading = new FunctionShading(Guid.NewGuid().ToString(),
        PredefinedColorSpaces.RGB,
        new Boundary(180, 180), RgbColors.Green.Components,
        new string[] {psFunctionR.ID, psFunctionG.ID, psFunctionB.ID});

    // register functions and shading object
    doc.ResourceManager.RegisterResource(psFunctionR);
    doc.ResourceManager.RegisterResource(psFunctionG);
    doc.ResourceManager.RegisterResource(psFunctionB);
    doc.ResourceManager.RegisterResource(functionShading);

    return functionShading;
}
```

The domain for each function is the same as shading’s boundary, so the shading’s color is defined within each point of it.

Radial shading based on PostScript function

Radial shadings define a color blend that varies between two circles. Shadings of this type are commonly used to depict three-dimensional spheres and cones. Here we used a simple PostScript function that produces the output color using the simple formula: (1-x) for the red component, and zeros for others.

```
// creates and registers radial shading object
private static Shading CreateAndRegisterRadialShading(FixedDocument doc)
{
    // create simple PS function that returns the following RGB color value: [(1-x), 0, 0]
    PostScriptFunction psFunction = new PostScriptFunction(Guid.NewGuid().ToString(),
        new double[]{0,1},new double[]{0,1,0,1,0,1}, "{1 exch sub 0 0}");

    RadialShading radialShading = new RadialShading( Guid.NewGuid().ToString(),
        PredefinedColorSpaces.RGB,new Boundary(180,180), RgbColors.White.Components,
        new double[]{120,110,0,90,90,90},new string[]{psFunction.ID});

    // register function and shading
    doc.ResourceManager.RegisterResource(psFunction);
    doc.ResourceManager.RegisterResource(radialShading);

    return radialShading;
}
```

This shading is defined using rectangular boundary [0, 0,180,180] and two circles defined by their centers and radii.

Shading Pattern

The shading pattern is like any other pattern except it's based on shading, see *8.7 Patterns* section of the PDF specification for the details. Thinking abstractly, the pattern is a like a color, depending on the location you fill or stoke in. In order to use it, you have to set the current stroking or non-stroking colorspace to a special value returned by the *PredefinedColorspaces::Pattern* property. We can use any shading as a base for the shading pattern. See the code below:

```
// create shading object
Shading functionShading = CreateAndRegisterFunctionBasedShading(doc);

// register shading pattern based on existing function-based shading
ShadingPattern shadingPattern = new ShadingPattern(Guid.NewGuid().ToString(),functionShading.ID);
doc.ResourceManager.RegisterResource(shadingPattern);
```

We used the shading created by one of the functions described earlier, created *ShadingPattern* object based on it and registered it as a document's resource. After that we can start using this pattern. Let's create a *FormXObject* and use the pattern as a text fill.

```
// creates a reusable piece of content (FormXObject)
private static FixedContent CreateAndRegisterXObject(FixedDocument doc, ShadingPattern fillColor)
{
    ClippedContent content = new ClippedContent(Path.CreateRoundRect(0, 0, 180, 180, 10, 10, 10, 10));

    content.SaveGraphicsState();

    // draw gray rect as a background for our text
    content.SetDeviceNonStrokingColor(RgbColors.LightGray.Components);
    content.FillPath(Path.CreateRect(0, 0, 180, 180));

    // draw text using shading pattern as a fill
    content.Translate(3, 85);
    TextObject txt = new TextObject(StandardFonts.Helvetica, 25);
    // set the fill colorspace to Pattern in order to use the pattern as a color
    txt.SetNonStrokingColorSpace(PredefinedColorSpaces.Pattern);
    // set our pattern as a fill color
    txt.SetNonStrokingColor(fillColor.ID);
    txt.AppendText("Shading pattern");
    content.AppendText(txt);
    content.RestoreGraphicsState();

    FixedContent formXObject = new FixedContent(Guid.NewGuid().ToString(),
        new Boundary(180, 180), content);
    doc.ResourceManager.RegisterResource(formXObject);

    return formXObject;
}
```

And draw it:

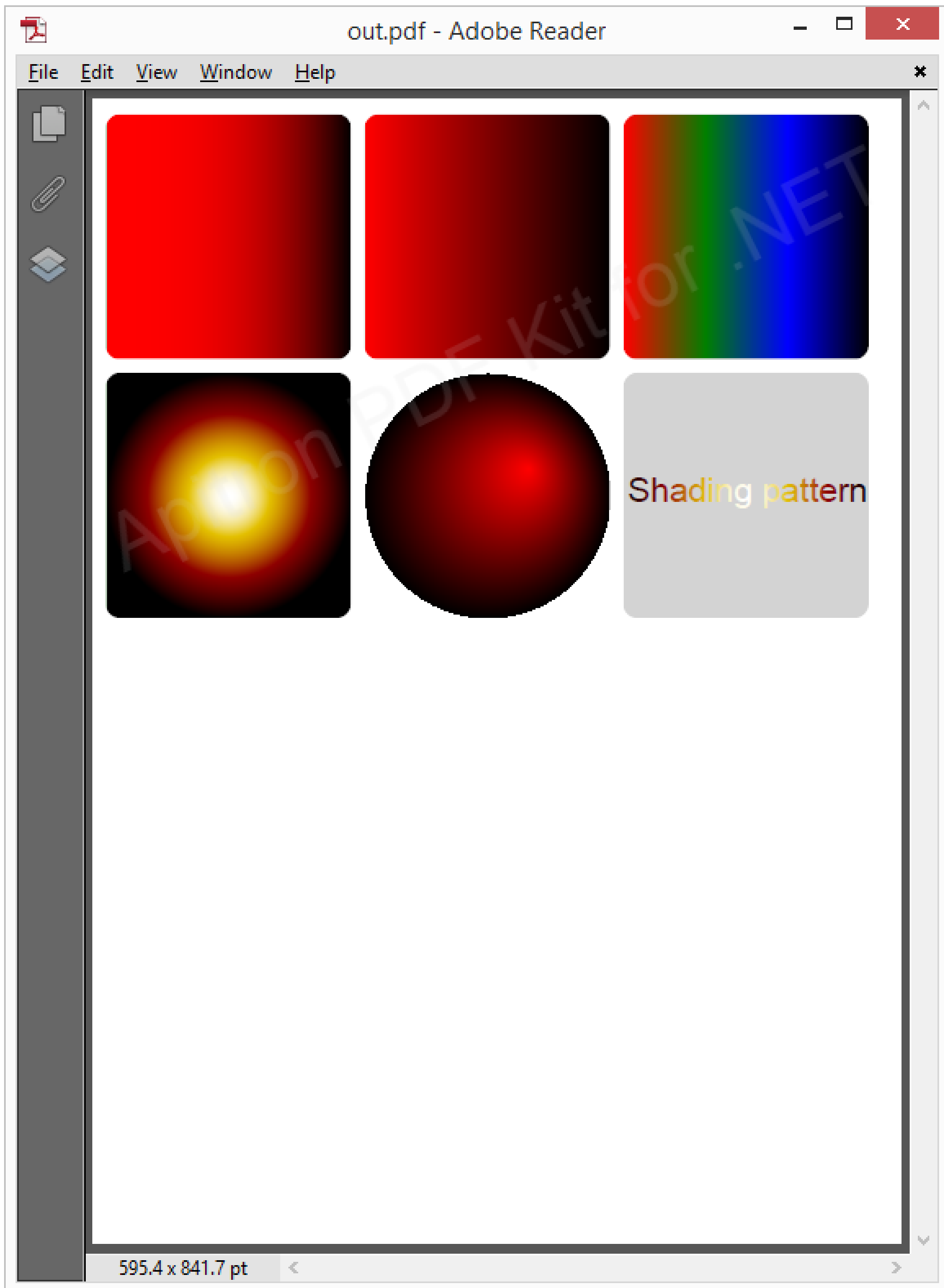
```
// draw the sample FormXObject demonstrating text fill using shading pattern
firstPage.Content.AppendXObject(CreateAndRegisterXObject(doc,shadingPattern).ID, 390, 460);
```

Sample program (shadings)

The code below draws all shadings described above on one page, including shading pattern.

```
static void Main(string[] args)
{
    // create document object
    FixedDocument doc = new FixedDocument();

    // create shading objects
    Shading axialShadingExp = CreateAndRegisterAxialShadingBasedOnExponentialFunction(doc,
        RgbColors.Red, RgbColors.Black);

    Shading axialShadingLinear = CreateAndRegisterAxialShadingBasedOnLinearInterpolationFunction(doc,
        new object[] { RgbColors.Red, RgbColors.Black });

    Shading axialShadingLinearMultipleColors =
        CreateAndRegisterAxialShadingBasedOnLinearInterpolationFunction( doc,
        new object[] { RgbColors.Red, RgbColors.Green, RgbColors.Blue, RgbColors.Black });

    Shading radialShading = CreateAndRegisterRadialShading(doc);

    Shading functionShading = CreateAndRegisterFunctionBasedShading(doc);

    // register shading pattern based on existing function-based shading
    ShadingPattern shadingPattern = new ShadingPattern(Guid.NewGuid().ToString(),functionShading.ID);
    doc.ResourceManager.RegisterResource(shadingPattern);

    // create document page
    Page firstPage = new Page();

    // draw shadings
    DrawShading(firstPage, axialShadingExp.ID, 10, 650);
    DrawShading(firstPage, axialShadingLinear.ID, 200, 650);
    DrawShading(firstPage, axialShadingLinearMultipleColors.ID, 390, 650);
    DrawShading(firstPage, functionShading.ID, 10, 460);
    DrawShading(firstPage, radialShading.ID, 200, 460);

    // draw the sample xObject demonstrating text fill using shading pattern
    firstPage.Content.AppendXObject(CreateAndRegisterXObject(doc,shadingPattern).ID, 390, 460);

    // add page to document
    doc.Pages.Add(firstPage);

    // save document
    using (Stream stream = File.Create("out.pdf"))
    {
        doc.Save(stream);
    }
    Process.Start("out.pdf");
}

// draws shading object on page at the specified coordinates
private static void DrawShading(Page page, string resourceName, double xOffset, double yOffset)
{
    page.Content.SaveGraphicsState();
    page.Content.Translate(xOffset, yOffset);
    page.Content.PaintShading(resourceName, Path.CreateRoundRect(0, 0, 180, 180, 10, 10, 10, 10));
    page.Content.RestoreGraphicsState();
}
```

The image below demonstrates the created PDF document:



Pic. 1 Shadings and shading patterns

The complete code sample can be found in our github repo.

Conclusion

Using Apitron PDF Kit for .NET you can create advanced PDF export tools, for example modules for CAD software, graphical editors, or document export routines. Its API is flexible enough, and allows you to implement complex visual effects using graphical operations described in the PDF specification.