Working with Gradients

A Simple Step-by-Step Tutorial

This article will explain to you how to set up gradients and render them. We will use a simple command-line example that produces the result in the `agg_test.ppm` file. You can use, for example IrfanView (www.irfanview.com) to see the results.

You will need to tell the compiler the **AGG** include directory and add three source files to the project or to the command line: `agg_rasterizer_scanline_aa.cpp`, `agg_trans_affine.cpp`, and `agg_sqrt_tables.cpp`. You can find the source file here: (gradients.cpp).

```
#include <stdio.h>
#include <string.h>
#include "agg_pixfmt_rgb.h"
#include "agg_renderer_base.h"
#include "agg_renderer_scanline.h"
#include "agg_scanline_u.h"
#include "agg_rasterizer_scanline_aa.h"
#include "agg_ellipse.h"
#include "agg_span_gradient.h"
#include "agg_span_interpolator_linear.h"

enum
{
    frame_width = 320,
    frame_height = 200
};

// Writing the buffer to a .PPM file, assuming it has
// RGB-structure, one byte per color component
//--------------------------------------------------
bool write_ppm(const unsigned char* buf,
               unsigned width,
               unsigned height,
               const char* file_name)
{
    FILE* fd = fopen(file_name, "wb");
    if(fd)
    {
        fprintf(fd, "P6 %d %d 255 ", width, height);
        fwrite(buf, 1, width * height * 3, fd);
        fclose(fd);
        return true;
    }
    return false;
}
```

```
// A simple function to form the gradient color array
// consisting of 3 colors, "begin", "middle", "end"
//---------------------------------------------------
template<class Array>
void fill_color_array(Array& array,
                      agg::rgba8 begin,
                      agg::rgba8 middle,
                      agg::rgba8 end)
{
    unsigned i;
    unsigned half_size = array.size() / 2;
    for(i = 0; i < half_size; ++i)
    {
        array[i] = begin.gradient(middle, i / double(half_size));
    }
    for(; i < array.size(); ++i)
    {
        array[i] = middle.gradient(end, (i - half_size) / double(half_size));
    }
}



int main()
{
    unsigned char* buffer = new unsigned char[frame_width * frame_height * 3];

    agg::rendering_buffer rbuf(buffer,
                               frame_width,
                               frame_height,
                               -frame_width * 3);

    // Pixel format and basic renderers.
    //-----------------
    typedef agg::pixfmt_rgb24 pixfmt_type;
    typedef agg::renderer_base<pixfmt_type> renderer_base_type;


    // The gradient color array
    typedef agg::pod_auto_array<agg::rgba8, 256> color_array_type;


    // Gradient shape function (linear, radial, custom, etc)
    //-----------------
    typedef agg::gradient_x gradient_func_type;


    // Span interpolator. This object is used in all span generators
    // that operate with transformations during iterating of the spans,
    // for example, image transformers use the interpolator too.
    //-----------------
    typedef agg::span_interpolator_linear<> interpolator_type;


    // Span allocator is an object that allocates memory for
    // the array of colors that will be used to render the
    // color spans. One object can be shared between different
```

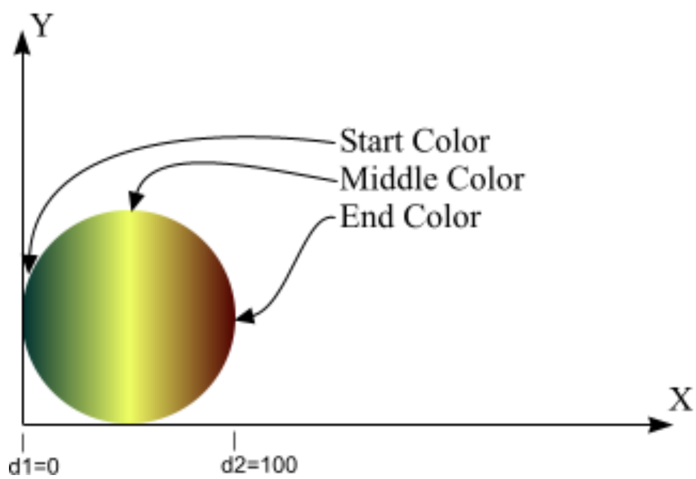
```
    // span generators.
    //------------------
    typedef agg::span_allocator<agg::rgba8> span_allocator_type;


    // Finally, the gradient span generator working with the agg::rgba8
    // color type.
    // The 4-th argument is the color function that should have
    // the [] operator returning the color in range of [0...255].
    // In our case it will be a simple look-up table of 256 colors.
    //------------------
    typedef agg::span_gradient<agg::rgba8,
                               interpolator_type,
                               gradient_func_type,
                               color_array_type,
                               span_allocator_type> span_gradient_type;


    // The gradient scanline renderer type
    //------------------
    typedef agg::renderer_scanline_aa<renderer_base_type,
                                      span_gradient_type> renderer_gradient_type;


    // Common declarations (pixel format and basic renderer).
    //----------------
    pixfmt_type pixf(rbuf);
    renderer_base_type rbase(pixf);


    // The gradient objects declarations
    //----------------
    gradient_func_type  gradient_func;                   // The gradient function
    agg::trans_affine   gradient_mtx;                    // Affine transformer
    interpolator_type   span_interpolator(gradient_mtx); // Span interpolator
    span_allocator_type span_allocator;                  // Span Allocator
    color_array_type    color_array;                     // Gradient colors


    // Declare the gradient span itself.
    // The last two arguments are so called "d1" and "d2"
    // defining two distances in pixels, where the gradient starts
    // and where it ends. The actual meaning of "d1" and "d2" depands
    // on the gradient function.
    //----------------
    span_gradient_type span_gradient(span_allocator,
                                     span_interpolator,
                                     gradient_func,
                                     color_array,
                                     0, 100);

    // The gradient renderer
    //----------------
    renderer_gradient_type ren_gradient(rbase, span_gradient);


    // The rasterizing/scanline stuff
    //----------------
    agg::rasterizer_scanline_aa<> ras;
    agg::scanline_u8 sl;
```

```
    // Finally we can draw a circle.
    //-----------------
    rbase.clear(agg::rgba8(255, 255, 255));

    fill_color_array(color_array,
                     agg::rgba8(0,50,50),
                     agg::rgba8(240, 255, 100),
                     agg::rgba8(80, 0, 0));

    agg::ellipse ell(50, 50, 50, 50, 100);
    ras.add_path(ell);

    agg::render_scanlines(ras, sl, ren_gradient);

    write_ppm(buffer, frame_width, frame_height, "agg_test.ppm");

    delete [] buffer;
    return 0;
}
```

It looks rather complex, especially the necessity to declare a lot of types and objects. But the "complexity" gives you freedom, for example, you can define your own gradient functions or even arbitrary distortions.

The example renders a circle with linear gradient from (0,0) to (100,0). In **AGG** you can define an arbitrary color function, in our case it's a simple look-up table generated from three colors, `start`, `middle`, and `end`.

Here is the result (the axes and text were added in **Xara X**):



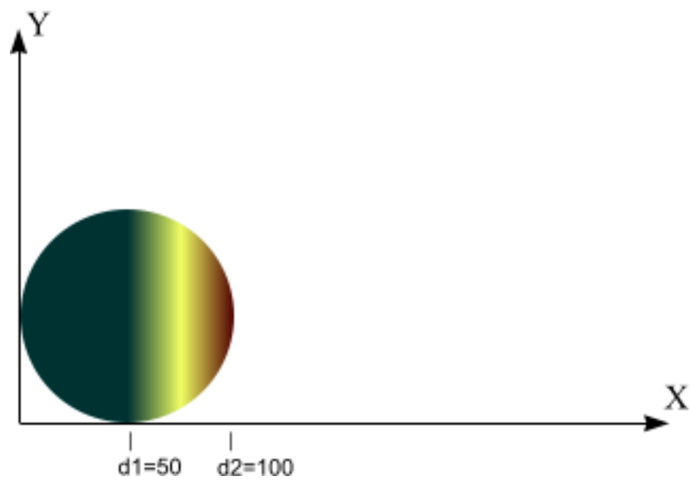
It also can seem like an overkill for this simple task, but later you will see that it's not so.

The next step is one little modification. Modify the following:

```
    // Declare the gradient span itself.
    // The last two arguments are so called "d1" and "d2"
    // defining two distances in pixels, where the gradient starts
    // and where it ends. The actual meaning of "d1" and "d2" depends
    // on the gradient function.
    //-----------------
```

```
    span_gradient_type span_gradient(span_allocator,
                                     span_interpolator,
                                     gradient_func,
                                     color_array,
                                     50, 100);
```

The result:



It should explain those freaky `d1` and `d2` arguments. In fact, they determine the geometrical `start` and `end` of the gradient and their meaning depends on the gradient function.

Now change the gradient function:

```
    // Gradient shape function (linear, radial, custom, etc)
    //-----------------
    typedef agg::gradient_circle gradient_func_type;
```

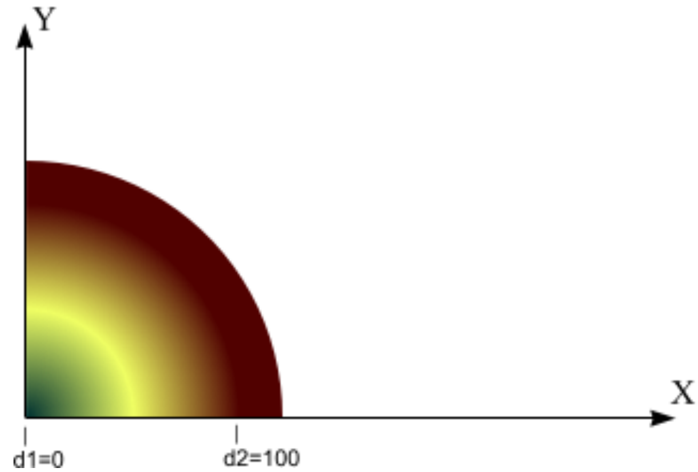
Set `d1` back to 0:

```
    // Declare the gradient span itself.
    // The last two arguments are so called "d1" and "d2"
    // defining two distances in pixels, where the gradient starts
    // and where it ends. The actual meaning of "d1" and "d2" depends
    // on the gradient function.
    //----------------
    span_gradient_type span_gradient(span_allocator,
                                     span_interpolator,
                                     gradient_func,
                                     color_array,
                                     0, 100);
```

And modify the circle:

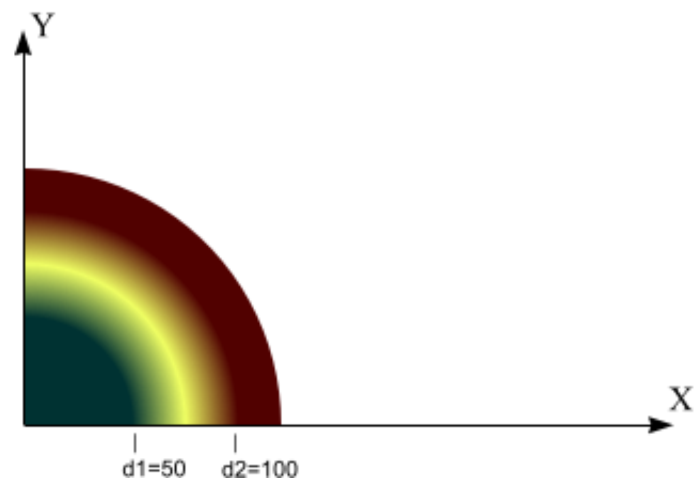
```
    agg::ellipse ell(0, 0, 120, 120, 100);
```

The result:



Modify d1 again:

```
    // Declare the gradient span itself.
    // The last two arguments are so called "d1" and "d2"
    // defining two distances in pixels, where the gradient starts
    // and where it ends. The actual meaning of "d1" and "d2" depends
    // on the gradient function.
    //----------------
    span_gradient_type span_gradient(span_allocator,
                                     span_interpolator,
                                     gradient_func,
                                     color_array,
                                     50, 100);
```



So that, in case of a radial gradient, d1 and d2 define the starting and ending radii.

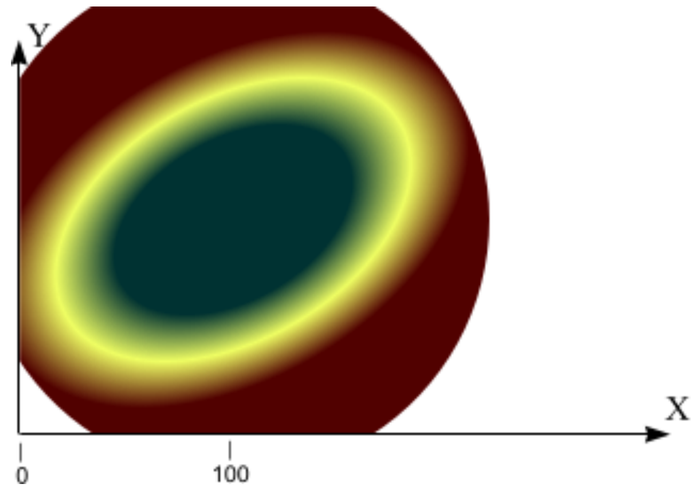
By default the origin point for the gradients is (0,0). How to draw a gradient in some other place? The answer is to use affine transformations. Strictly speaking, the transformations are fully defined by the span interpolator. In our case we use span_interpolator_linear with an affine matrix. The linear interpolator allows you to speed up the calculations vastly, because we calculate the floating point coordinates only in the begin and end of the horizontal spans and then use a fast, integer, Bresenham-like interpolation with **Subpixel Accuracy**.

Add the following code somewhere before calling `agg::render_scanlines(ras, sl, ren_gradient);`

```
    gradient_mtx *= agg::trans_affine_scaling(0.75, 1.2);
    gradient_mtx *= agg::trans_affine_rotation(-agg::pi/3.0);
    gradient_mtx *= agg::trans_affine_translation(100.0, 100.0);
    gradient_mtx.invert();
```

And modify the circle:

```
    agg::ellipse ell(100, 100, 120, 120, 100);
```



The code of initializing of the affine matrix should be obvious except for some strange `gradient_mtx.invert()`. It's necessary because the gradient generator uses **reverse** transformations instead of **direct** ones. In other words it takes the destination point, applies the transformations and obtains the coordinates in the gradient. Note that the affine transformations allow you to turn a circular gradient into elliptical.

Now it should be obvious how to define a linear gradient from some `Point1` to `Point2`. So, get back to the original code and add the following function:

```
// Calculate the affine transformation matrix for the linear gradient
// from (x1, y1) to (x2, y2). gradient_d2 is the "base" to scale the
// gradient. Here d1 must be 0.0, and d2 must equal gradient_d2.
//---------------------------------------------------------------
void calc_linear_gradient_transform(double x1, double y1, double x2, double y2,
                                    agg::trans_affine& mtx,
                                    double gradient_d2 = 100.0)
{
    double dx = x2 - x1;
    double dy = y2 - y1;
    mtx.reset();
    mtx *= agg::trans_affine_scaling(sqrt(dx * dx + dy * dy) / gradient_d2);
    mtx *= agg::trans_affine_rotation(atan2(dy, dx));
    mtx *= agg::trans_affine_translation(x1, y1);
    mtx.invert();
```

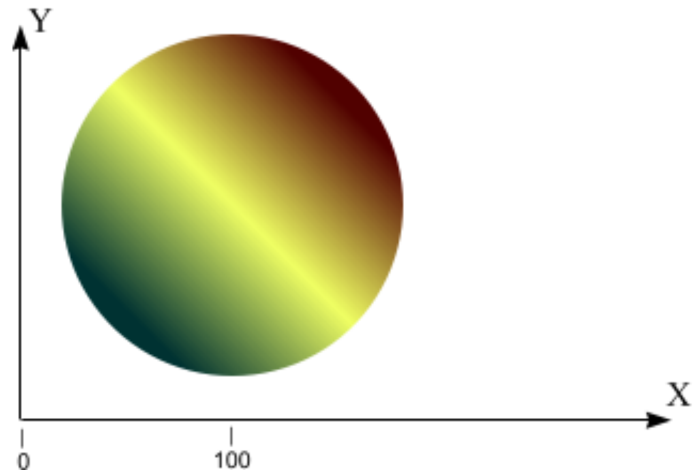
```
}
```

Then modify the circle:

```
    agg::ellipse ell(100, 100, 80, 80, 100);
```

And add the transformations:

```
    calc_linear_gradient_transform(50, 50, 150, 150,  gradient_mtx);
```



Try to play with different parameters, transformations, and gradient functions: gradient_circle, gradient_x, gradient_y, gradient_diamond, gradient_xy, gradient_sqrt_xy, gradient_conic. Also look at the gradient functions and try to write your own. Actually, the set of the gradient functions in **AGG** is rather poor, it just demonstrates the possibilities. For example, repeating or reflecting gradients should be implemented in gradient functions (or you can write adaptors that will use the existing functions).