Home/
Table of Content/

THE AGG PROJECT Anti-Grain Geometry

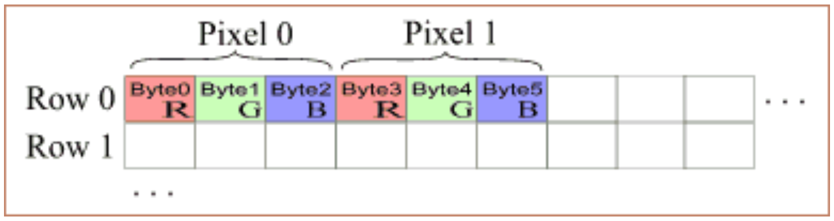
News **Docs** **Download** **Mailing List** **CVS**

Basic Renderers

Getting Started with Simple Console Applications

Rendering Buffer

Here we start with creating a frame buffer in memory and writing it to a file of the simplest possible raster format. It's PPM (Portable Pixel Map). Although, it isn't natively supported by **Microsoft** Windows, there are many viewers and converters that can work with it, for example, IrfanView (www.irfanview.com). All **AGG** console examples use the P6 256 format, that is RGB, one byte per channel. We assume that we work with an RGB-buffer in memory organized as follows:



The First and the Simplest Example

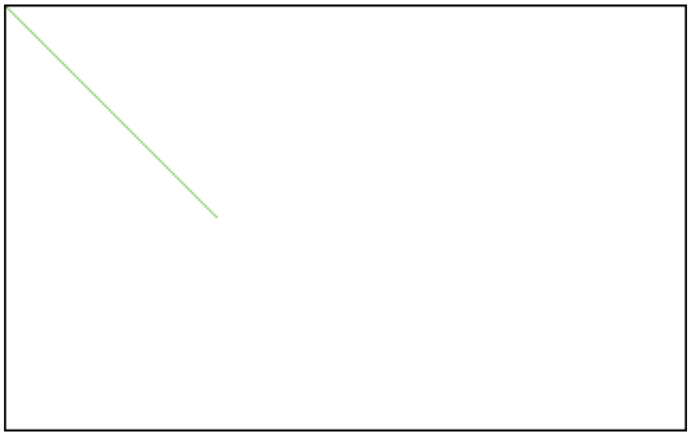
There is the first example, it's in `agg2/tutorial/t01_rendering_buffer.cpp`

```
#include <stdio.h>
#include <string.h>
#include "agg_rendering_buffer.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;
}

// Draw a black frame around the rendering buffer, assuming it has
// RGB-structure, one byte per color component
//--------------------------------------------------
void draw_black_frame(agg::rendering_buffer& rbuf)
{
    unsigned i;
    for(i = 0; i < rbuf.height(); ++i)
    {
        unsigned char* p = rbuf.row_ptr(i);
        *p++ = 0; *p++ = 0; *p++ = 0;
        p += (rbuf.width() - 2) * 3;
        *p++ = 0; *p++ = 0; *p++ = 0;
    }
    memset(rbuf.row_ptr(0), 0, rbuf.width() * 3);
    memset(rbuf.row_ptr(rbuf.height() - 1), 0, rbuf.width() * 3);
}
```

```
int main()
{
    // In the first example we do the following:
    //--------------------
    // Allocate the buffer.
    // Clear the buffer, for now "manually"
    // Create the rendering buffer object
    // Do something simple, draw a diagonal line
    // Write the buffer to agg_test.ppm
    // Free memory

    unsigned char* buffer = new unsigned char[frame_width * frame_height * 3];

    memset(buffer, 255, frame_width * frame_height * 3);

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

    unsigned i;
    for(i = 0; i < rbuf.height()/2; ++i)
    {
        // Get the pointer to the beginning of the i-th row (Y-coordinate)
        // and shift it to the i-th position, that is, X-coordinate.
        //---------------
        unsigned char* ptr = rbuf.row_ptr(i) + i * 3;

        // PutPixel, very sophisticated, huh? :)
        //-------------
        *ptr++ = 127; // R
        *ptr++ = 200; // G
        *ptr++ = 98;  // B
    }

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

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

In this example you don't even have to link it with any **AGG** files, you need only to indicate the **AGG** `include` directory in the command line of your compiler.

When you compile and run it you should see the following result.



Almost everything here is coded “manually”. The only class we use is rendering_buffer. This class doesn't know anything about the pixel format in memory, it just keeps an array of pointers to each row. It is your responsibility to allocate and deallocate actual memory for the buffer. You can use any available mechanism for that, a system API function, simple memory allocation, or even an array defined statically. In the above example we allocate `width * height * 3` bytes of memory, since we use 3 bytes per pixel. The rows are not aligned in memory, but they can be for better performance or if it is required by used API.

Class rendering_buffer

Include file: agg_rendering_buffer.h

The rendering buffer class keeps pointers to each rows, and basically it's all it does. It doesn't look like a great achievement, but try to keep reading. The interface and functionality of it is very simple. It's a `typedef` of class template row_ptr_cache:

```
typedef row_ptr_cache<int8u> rendering_buffer;
```

The interface and the functionality of class row_ptr_cache is:

```
template<class T> class row_ptr_cache
{
public:
    row_ptr_cache();

    row_ptr_cache(T* buf, unsigned width, unsigned height, int stride);

    void attach(T* buf, unsigned width, unsigned height, int stride);

          T* buf();
    const T* buf()    const;
    unsigned width()  const;
    unsigned height() const;
    int      stride() const;
    unsigned stride_abs() const;

          T* row_ptr(int, int y, unsigned)
          T* row_ptr(int y);
    const T* row_ptr(int y) const;
    row_data row    (int y) const;
    T const* const* rows() const;

    template<class RenBuf> void copy_from(const RenBuf& src);
    void clear(T value)
};
```

The source code: row_ptr_cache

The class doesn't have any assertion or verification code, so that, your responsibility is to properly attach an actual memory buffer to the object before using it. It can be done in the constructor as well as with the `attach()` function. The arguments of them are:

- **buf** — A pointer to the memory buffer.
- **width** — Width of the image in **pixels**. The rendering buffer doesn't know anything about pixel format and about the size of one pixel in memory. This value is simply stored in its data member `m_width` and returned by `width()` function.
- **height** — The height of the buffer in **pixels** (the number of rows).
- **stride** — The “stride” (big step) of the rows measured in objects of type `T`. Class rendering_buffer is “`typedefed`” as `row_ptr_cache<int8u>`, so, this value is in bytes. **Stride** determines the physical width of one row in memory. If this value is negative the direction of the **Y** axis is inverted, that is, `Y==0` will point to the last row of the buffer, `Y==height-1` — to the first one. The absolute value of **stride** is important too, because it allows you to work with buffers whose rows are aligned in memory (like in Windows BMP, for example). Besides, this parameter allows you to work with any rectangular area inside the buffer as with the whole buffer.

Function `attach()` changes the buffer or its parameters. It takes care of reallocating the memory for the row-pointers buffer, so that, you can call it anytime. It will reallocate the memory if (and only if) new `height` is is larger than any previously attached.

The cost of creation is just initializing of the variables (set to 0), the cost of `attach()` is allocating of `sizeof(ptr) * height` bytes of memory and initializing of the pointers to the rows.

The most frequently used function is `row_ptr(y)` that simply returns a pointer to the beginning of the `y`th row, considering the direction of the **Y**-axis.

IMPORTANT!
The rendering buffer does not perform any clipping or bound checking, it's the responsibility of higher level classes.

The accessors, `buf()`, `width()`, `height()`, `stride()`, `stride_abs()` should be obvious.

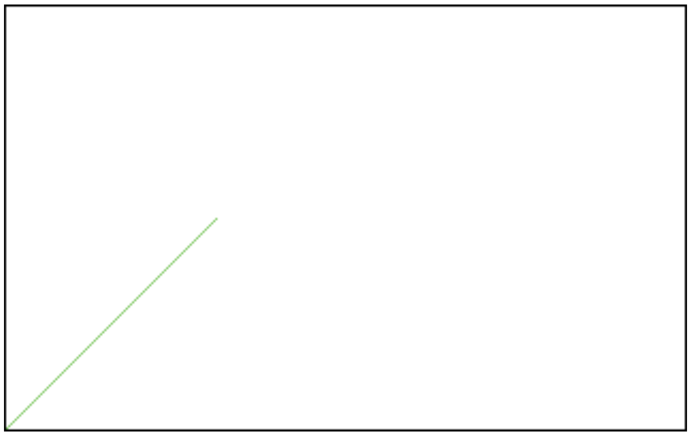
Function `copy_from()` copies the content of another buffer to “this” one. The function is safe, if the **width** or **height** is different it will copy the maximal possible area. Basically, it's used to copy rendering buffers of equal size.

Two Modifications of the Example

First, negate the **stride** when creating the rendering buffer object:

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

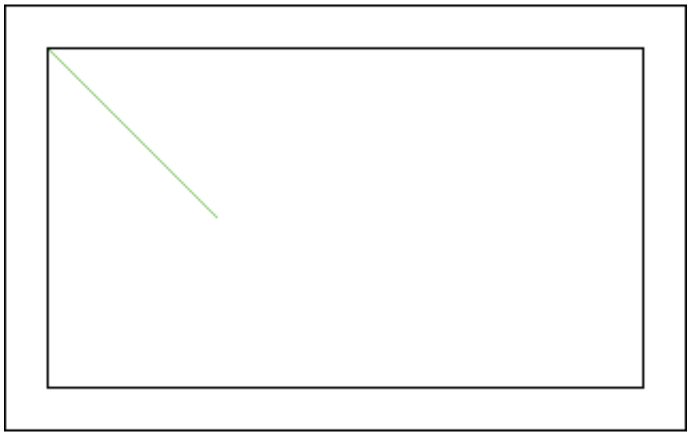
The result is:



Second, let's try to attach to some part of the allocated buffer. This modification actually attaches to the same buffer twice, first, to the whole frame, then to its part, with 20 pixel margin.

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

    memset(buffer, 255, frame_width * frame_height * 3);

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

    // Draw the outer black frame
    //------------------------
    draw_black_frame(rbuf);

    // Attach to the part of the buffer,
    // with 20 pixel margins at each side.
    rbuf.attach(buffer +
                  frame_width * 3 * 20 +      // initial Y-offset
                  3 * 20,                     // initial X-offset
                frame_width - 40,
                frame_height - 40,
                frame_width * 3               // Note that the stride
                                              // remains the same
                );

    // Draw a diagonal line
    //------------------------
    unsigned i;
    for(i = 0; i < rbuf.height()/2; ++i)
    {
        // Get the pointer to the beginning of the i-th row (Y-coordinate)
        // and shift it to the i-th position, that is, X-coordinate.
        //---------------
        unsigned char* ptr = rbuf.row_ptr(i) + i * 3;

        // PutPixel, very sophisticated, huh? :)
        //-------------
        *ptr++ = 127; // R
        *ptr++ = 200; // G
        *ptr++ = 98;  // B
    }

    // Draw the inner black frame
    //------------------------
    draw_black_frame(rbuf);

    // Write to a file
    //------------------------
    write_ppm(buffer, frame_width, frame_height, "agg_test.ppm");

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

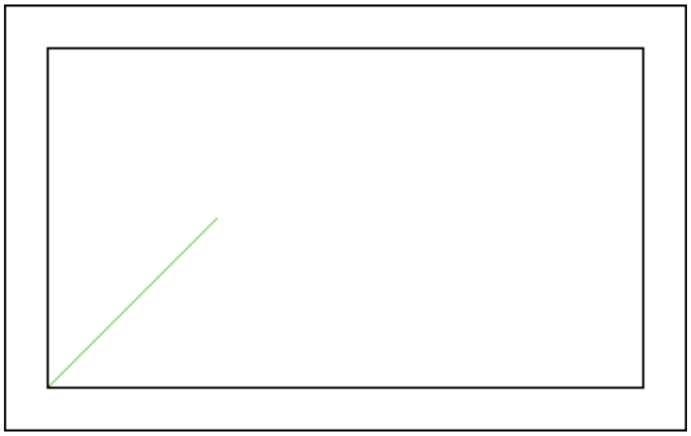
The result:



And the last modification is:

```
    // Attach to the part of the buffer,
    // with 20 pixel margins at each side and negative 'stride'
    rbuf.attach(buffer +
                  frame_width * 3 * 20 +      // initial Y-offset
                  3 * 20,                     // initial X-offset
                frame_width - 40,
                frame_height - 40,
                -frame_width * 3              // Negate the stride
                );
```

The result:



In the last example we just negated the stride value, keeping the pointer to the beginning of the buffer as it was in the previous example.

NOTE
Function `write_ppm()` writes the pixel map to a file. Hereafter it will be omited in this text, but duplicated when necessary in source code in the `agg2/tutorial` directory.

Pixel Format Renderers

First, we create another, more civil example, which is in `agg2/tutorial/t02_pixel_formats`:

```
#include <stdio.h>
#include <string.h>
#include "agg_pixfmt_rgb24.h"

enum
{
    frame_width = 320,
    frame_height = 200
};

// [...write_ppm is skipped...]

// Draw a black frame around the rendering buffer
//--------------------------------------------------
template<class Ren>
void draw_black_frame(Ren& ren)
{
    unsigned i;
    agg::rgba8 c(0,0,0);
    for(i = 0; i < ren.height(); ++i)
    {
        ren.copy_pixel(0,               i, c);
        ren.copy_pixel(ren.width() - 1, i, c);
    }
    for(i = 0; i < ren.width(); ++i)
    {
        ren.copy_pixel(i, 0,                c);
        ren.copy_pixel(i, ren.height() - 1, c);
    }
}


int main()
{
    //--------------------
    // Allocate the buffer.
    // Clear the buffer, for now "manually"
    // Create the rendering buffer object
    // Create the Pixel Format renderer
    // Do something simple, draw a diagonal line
    // Write the buffer to agg_test.ppm
    // Free memory

    unsigned char* buffer = new unsigned char[frame_width * frame_height * 3];

    memset(buffer, 255, frame_width * frame_height * 3);

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

    agg::pixfmt_rgb24 pixf(rbuf);

    unsigned i;
    for(i = 0; i < pixf.height()/2; ++i)
    {
        pixf.copy_pixel(i, i, agg::rgba8(127, 200, 98));
    }

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

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

This example doesn't look very different from the previous one, but the difference in its essence is great. Look at the declaration:

```
agg::pixfmt_rgb24 pixf(rbuf);
```

Here we create a low-level pixel rendering object and attach it to the rendering buffer.

It defined as:

```
typedef pixel_formats_rgb24<order_rgb24> pixfmt_rgb24;
```

Class template pixel_formats_rgb24 has full knowledge about this particular pixel format in memory. The only template parameter can be order_rgb24 or order_bgr24 that determines the order of color channels.

Unlike rendering_buffer, these classes operate with integer pixel coordinates because they know how to calculate the offset for particular **X**. One can say that it would be easier to keep the width of the pixel inside the rendering_buffer, but in practice it would be a restriction. Don't forget that the width of one pixel in memory can be less than one byte, for example, when rendering high resolution B&W images for printers. Thus, there is just a separation of the functionality, rendering_buffer just accelerates access to rows, pixel format renderers have knowledge of how to interpret the rows.

Currently, in **AGG** there are the following files that implement different pixel formats:

- agg_pixfmt_gray8.h, one byte per pixel grayscale buffer. This pixel format allows you to work with one color component of the rgb24 or rgba32 pixel format. It has the template parameters `Step` and `Offset` and for the convenience defines the following types:
  - `typedef pixfmt_gray8_base<1, 0> pixfmt_gray8;`
  - `typedef pixfmt_gray8_base<3, 0> pixfmt_gray8_rgb24r;`
  - `typedef pixfmt_gray8_base<3, 1> pixfmt_gray8_rgb24g;`
  - `typedef pixfmt_gray8_base<3, 2> pixfmt_gray8_rgb24b;`
  - `typedef pixfmt_gray8_base<3, 2> pixfmt_gray8_bgr24r;`
  - `typedef pixfmt_gray8_base<3, 1> pixfmt_gray8_bgr24g;`
  - `typedef pixfmt_gray8_base<3, 0> pixfmt_gray8_bgr24b;`
  - `typedef pixfmt_gray8_base<4, 0> pixfmt_gray8_rgba32r;`
  - `typedef pixfmt_gray8_base<4, 1> pixfmt_gray8_rgba32g;`
  - `typedef pixfmt_gray8_base<4, 2> pixfmt_gray8_rgba32b;`
  - `typedef pixfmt_gray8_base<4, 3> pixfmt_gray8_rgba32a;`
  - `typedef pixfmt_gray8_base<4, 1> pixfmt_gray8_argb32r;`
  - `typedef pixfmt_gray8_base<4, 2> pixfmt_gray8_argb32g;`
  - `typedef pixfmt_gray8_base<4, 3> pixfmt_gray8_argb32b;`
  - `typedef pixfmt_gray8_base<4, 0> pixfmt_gray8_argb32a;`
  - `typedef pixfmt_gray8_base<4, 2> pixfmt_gray8_bgra32r;`
  - `typedef pixfmt_gray8_base<4, 1> pixfmt_gray8_bgra32g;`
  - `typedef pixfmt_gray8_base<4, 0> pixfmt_gray8_bgra32b;`
  - `typedef pixfmt_gray8_base<4, 3> pixfmt_gray8_bgra32a;`
  - `typedef pixfmt_gray8_base<4, 3> pixfmt_gray8_abgr32r;`
  - `typedef pixfmt_gray8_base<4, 2> pixfmt_gray8_abgr32g;`
  - `typedef pixfmt_gray8_base<4, 1> pixfmt_gray8_abgr32b;`
  - `typedef pixfmt_gray8_base<4, 0> pixfmt_gray8_abgr32a;`
- agg_pixfmt_rgb24.h, three bytes per pixel, with RGB or BGR component orders. Defines the following pixel format types:
  - `typedef pixel_formats_rgb24<order_rgb24> pixfmt_rgb24;`
  - `typedef pixel_formats_rgb24<order_bgr24> pixfmt_bgr24;`
- agg_pixfmt_rgb555.h, 15 bits per pixel, 5 bits per channel. The elder bit is unused.
- agg_pixfmt_rgb565.h, 16 bits per pixel, 5 bits for Red, 6 bits for Green, and 5 bits for Blue.
- agg_pixfmt_rgba32.h, four bytes per pixel, RGB plus Alpha with different component orders:
  - `typedef pixel_formats_rgba32<order_rgba32> pixfmt_rgba32;`
  - `typedef pixel_formats_rgba32<order_argb32> pixfmt_argb32;`
  - `typedef pixel_formats_rgba32<order_abgr32> pixfmt_abgr32;`
  - `typedef pixel_formats_rgba32<order_bgra32> pixfmt_bgra32;`

The pixel format classes define their native color space and the color type, for example:

```
typedef rgba8 color_type;
```

For `pixfmt_gray8_nnn` it's gray8. This mechanism allows you to write your own pixel and color formats, for example, HSV, CMYK, and so on. The rest of the library will work with new pixel formats in exactly same way as with currently implemented ones.

NOTE
It's very important not to confuse the color type that the pixel format renderer works with and the **native** color space that the buffer represents. For example, you can pretend that you work with the CMYK color space using the RGB buffer (you just write a simple conversion function that creates an rgba8 object from some CMYK structure). But it will be only an imitation and you will have color losses, because there are some colors in CMYK that cannot be displayed with RGB and vice versa. To use the capabilities of some color space completely one has to write a pixel format renderer for that particular color space and to work in that color space without any intermediate conversions.

Creation

IMPORTANT!
The pixel format classes do not perform any clipping operations, which means that working directly with these classes is generally unsafe. Clipping is the functionality of higher level classes. The reason of this design is simple — it must be as easy as possible to write your own pixel format classes. There can be many of them while the clipping code remains exactly the same.

```
pixel_formats_rgb24(rendering_buffer& rb);
```

The constructor of the pixel format renderers expects a reference to the created and fully initialized rendering_buffer. The cost of creation is minimal, basically it's initializing of one pointer.

Member Functions

The pixel format renderers must expose the following functionality (interface).

```
unsigned width()  const { return m_rbuf->width();  }
unsigned height() const { return m_rbuf->height(); }
```

Returns width and height of the buffer in pixels.

```
color_type pixel(int x, int y);
```

Returns the color value of the pixel with coordinates `(x,y)`.

```
void copy_pixel(int x, int y, const color_type& c);
```

Copies a pixel of color `c` to the buffer as it is. In the RGB pixel formats it doesn't consider the alpha channel existing in the rgba8 type, in RGBA — it simply copies the alpha value to the buffer together with R, G, and B.

```
void blend_pixel(int x, int y, const color_type& c, int8u cover);
```

Blends a pixel of color `c` with the one in the buffer. Now it's time to explain the concept of blending. Blending is a key feature for **Anti-Aliasing**. In the RGBA color space we use the rgba8 structure to represent colors. The structure already has data member `int8u a;` that is the alpha channel. But in this function we also see argument `cover` that deterines the coverage value for the pixel, i.e, the part of the pixel that is “covered” by the polygon. In fact, you can interpret it as a second "Alpha" ("Beta"?). There are two reasons to do so. First of all, the color type doesn't have to contain the alpha value. The second, even if the color type has the Alpha field, its type doesn't have to be compatible with the ones used in **Anti-Aliasing** algorithms. Suppose you work with a “Hi-End” RGBA color space represented as four floats in range [0…1]. Its alpha value is also of type float — one byte is too bad for general blending in this case, but quite enough for **Anti-Aliasing**. So that, the `cover` value is just a unified secondary alpha used specifically for **Anti-Aliasing** purposes. Globally, it's defined as cover_type, but in the described rasterizers there explicit int8u type is used. It's done intentionally, because if there's a necessity to increase the capacity of cover_type, all the existing pixel format rasterizres become incompatible with the cover_type. They will be **actually** incompatible, in fact, 8-bit coverage values plus 8-bit alpha is the maximum that fits 32-bit intermediate results when blending colors. In case of 16-bit values we will have to use 64-bit integers, which is very expensive on 32-bit platforms.

```
void copy_hline(int x, int y, unsigned len, const color_type& c);
```

```
void copy_vline(int x, int y, unsigned len, const color_type& c);
```

Draw a horizontal or a vertical line of certain color.

```
void blend_hline(int x, int y, unsigned len, const color_type& c, int8u cover);
```

```
void blend_vline(int x, int y, unsigned len, const color_type& c, int8u cover);
```

Blend a horizontal or a vertical line of certain color. The reason to separate “copy” and “blend” versions is the performance. Of course, there can be one extra `if/else` statement (and there is, in the “blend” versions), but still, it becomes critical when using `hline/vline` to draw many of small markers, like in different scatter plot applications.

```
void blend_solid_hspan(int x, int y, unsigned len,
                       const color_type& c, const int8u* covers);
```

```
void blend_solid_vspan(int x, int y, unsigned len,
                       const color_type& c, const int8u* covers);
```

Blend a horizontal or a vertical solid-color span. Span is almost the same as `hline/vline`, but there's an array of the coverage values. These functions are used when rendering solid Anti-Aliased polygons.

```
void blend_color_hspan(int x, int y, unsigned len,
                       const color_type* colors, const int8u* covers);
```

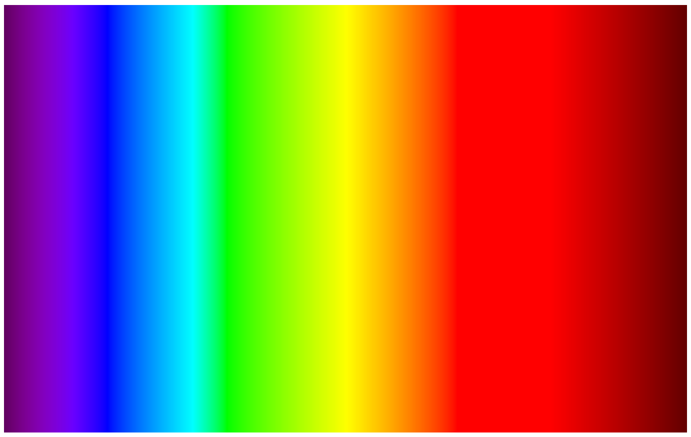
```
void blend_color_vspan(int x, int y, unsigned len,
                       const color_type* colors, const int8u* covers);
```

Blend a horizontal or a vertical color span. The functions are used with different span generators, such as gradients, images, patterns, Gouraud interpolation, etc. They accept an array of colors whose type must be compatible with the used pixel format. For example, all existing RGB pixel formats are compatible with the rgba8 type. Argument **covers** is an array of the coverage values as in the `blend_solid_hspan`. It's optional and can be 0.

Another example is drawing of the solar spectrum. Class rgba, that keeps four components as doubles, has static method `from_wavelength` and the respective constructor, class rgba8 can be constructed from rgba (it's a common policy in **AGG** that any color type can be constructed from the rgba type). We will use it.

```
#include <stdio.h>
#include <string.h>
#include "agg_pixfmt_rgb24.h"

enum
{
    frame_width = 320,
    frame_height = 200
};

// [...write_ppm is skipped...]

int main()
{
    //--------------------
    // Allocate the buffer.
    // Clear the buffer, for now "manually"
    // Create the rendering buffer object
    // Create the Pixel Format renderer
    // Create one line (span) of type rgba8.
    // Fill the buffer using blend_color_span
    // Write the buffer to agg_test.ppm
    // Free memory

    unsigned char* buffer = new unsigned char[frame_width * frame_height * 3];

    memset(buffer, 255, frame_width * frame_height * 3);

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

    agg::pixfmt_rgb24 pixf(rbuf);

    agg::rgba8 span[frame_width];

    unsigned i;
    for(i = 0; i < frame_width; ++i)
    {
        agg::rgba c(380.0 + 400.0 * i / frame_width, 0.8);
        span[i] = agg::rgba8(c);
    }

    for(i = 0; i < frame_height; ++i)
    {
        pixf.blend_color_hspan(0, i, frame_width, span, 0);
    }
    write_ppm(buffer, frame_width, frame_height, "agg_test.ppm");

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

Here is the result:



Alpha-Mask Adaptor

Alpha-mask is a separate buffer that is usually used to perform clipping to an arbitrary shape at the low level. There is a special adapter class that passes all calls to pixel format renderes through the alpha-mask filter. Usually alpha-mask is a gray scale buffer (one byte per pixel) of the same size as the main rendering buffer. Each pixel in the alpha-mask deternines an additional pixel coverage value that is mixed with the main one. Functions like `copy_hline()`, that do not have a coverage value argument translate the calls to the respective functions with this argument. For example, `copy_hline()` takes the horizontal span from the alpha mask buffer and then calls `blend_solid_hspan()`.

Include files:

```
#include "agg_pixfmt_amask_adaptor.h"
#include "agg_alpha_mask_u8.h"
```

Below is an example of how to declare the pixel format renderer with the alpha-mask adaptor.

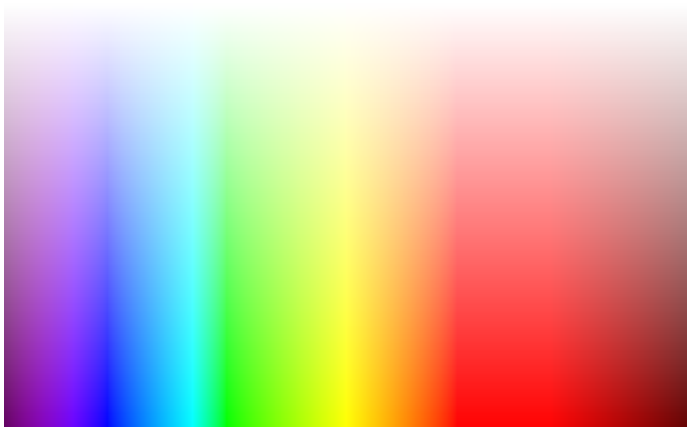
```
#include "agg_pixfmt_rgb24.h"
#include "agg_pixfmt_amask_adaptor.h"
#include "agg_alpha_mask_u8.h"

//. . .

    // Allocate the alpha-mask buffer, create the rendering buffer object
    // and create the alpha-mask object.
    //--------------------------------
    agg::int8u* amask_buf = new agg::int8u[frame_width * frame_height];
    agg::rendering_buffer amask_rbuf(amask_buf,
                                     frame_width,
                                     frame_height,
                                     frame_width);
    agg::amask_no_clip_gray8 amask(amask_rbuf);
    // Create the alpha-mask adaptor attached to the alpha-mask object
    // and the pixel format renderer. Here pixf is a previously
    // created pixel format renderer of type agg::pixfmt_rgb24.
    agg::pixfmt_amask_adaptor<agg::pixfmt_rgb24,
                              agg::amask_no_clip_gray8> pixf_amask(pixf, amask);
    //. . .
```

Note that here we use amask_no_clip_gray8 that doesn't perform clipping. It's because we use the main and the alpha-mask buffers of exactly same size, so, if there are no memory violations in the main rendering buffer, there will be no memory violations in the alpha mask buffer either. Clipping is performed at the higher level. If your alpha-mask buffer is less than the main one you will have to use alpha_mask_gray8 instead.

Below is a complete example.

```
#include <stdio.h>
#include <string.h>
#include "agg_pixfmt_rgb24.h"
#include "agg_pixfmt_amask_adaptor.h"
#include "agg_alpha_mask_u8.h"

enum
{
    frame_width = 320,
    frame_height = 200
};

// [...write_ppm is skipped...]

int main()
{
    // Allocate the main rendering buffer and clear it, for now "manually",
    // and create the rendering_buffer object and the pixel format renderer
    //--------------------------------
    agg::int8u* buffer = new agg::int8u[frame_width * frame_height * 3];
    memset(buffer, 255, frame_width * frame_height * 3);
    agg::rendering_buffer rbuf(buffer,
                               frame_width,
                               frame_height,
                               frame_width * 3);
    agg::pixfmt_rgb24 pixf(rbuf);


    // Allocate the alpha-mask buffer, create the rendering buffer object
    // and create the alpha-mask object.
    //--------------------------------
    agg::int8u* amask_buf = new agg::int8u[frame_width * frame_height];
    agg::rendering_buffer amask_rbuf(amask_buf,
                                     frame_width,
                                     frame_height,
                                     frame_width);
    agg::amask_no_clip_gray8 amask(amask_rbuf);

    // Create the alpha-mask adaptor attached to the alpha-mask object
    // and the pixel format renderer
    agg::pixfmt_amask_adaptor<agg::pixfmt_rgb24,
                              agg::amask_no_clip_gray8> pixf_amask(pixf, amask);


    // Draw something in the alpha-mask buffer.
    // In this case we fill the buffer with a simple verical gradient
    unsigned i;
    for(i = 0; i < frame_height; ++i)
    {
        unsigned val = 255 * i / frame_height;
        memset(amask_rbuf.row_ptr(i), val, frame_width);
    }


    // Draw the spectrum, write a .ppm and free memory
    //----------------------
    agg::rgba8 span[frame_width];

    for(i = 0; i < frame_width; ++i)
    {
        agg::rgba c(380.0 + 400.0 * i / frame_width, 0.8);
        span[i] = agg::rgba8(c);
    }

    for(i = 0; i < frame_height; ++i)
    {
        pixf_amask.blend_color_hspan(0, i, frame_width, span, 0);
    }

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

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

And the result:



Note that we cleared the main buffer with the white color. Now change

```
    memset(buffer, 255, frame_width * frame_height * 3);
```

to

```
    memset(buffer, 0, frame_width * frame_height * 3);
```

The result:

In other words, the alpha-mask works as a separete alpha-channel mixed with rendered primitives. The fact that it contains 8-bit values allows you to clip all drawing to arbitrary shapes with perfect **Anti-Aliasing**.

Basic Renderers

There are two basic renderers with almost the same functionality: renderer_base and renderer_mclip. The first one is used most often and it performs low level clipping. Clipping in general is a complex task. In **AGG** there can be at least two levels of clipping, the low (pixel) level, and the high (vectorial) level. These classes perform the pixel level clipping to protect the buffer from out-of-bound memory access. Class renderer_mclip performs clipping to multi-rectangle clipping areas, but its performance depends on the number of clipping rectangles.

renderer_base and renderer_mclip are class templates parametrized with a pixel format renderer:

```
template<class PixelFormat> class renderer_base
{
public:
    typedef PixelFormat pixfmt_type;
    typedef typename pixfmt_type::color_type color_type;

    . . .
};
```

See sources renderer_base renderer_mclip

Creation

```
renderer_base(pixfmt_type& ren);
renderer_mclip(pixfmt_type& ren);
```

Both classes accept a reference to the pixel format renderer. renderer_mclip uses renderer_base<PixelFormat> inside itself to perform implemented clipping to a single rectangle region. Note that you can use pixfmt_amask_adaptor as `PixelFormat` template parameter.

The cost of creation is minimal, it's just initializing of the class member variables. But renderer_mclip allocates memory when you add new clipping rectangles and deallocates it when destroying. It uses the pod_deque class that allocates blocks of memory of equal size and never reallocates it. When you reset clipping, the memory is not deallocated, it's reused. The renderer_mclip deallocates memory only when destroying. This technique is widely used in **AGG** and prevents from deep memory fragmentation.

Member Functions

```
const pixfmt_type& ren() const;
pixfmt_type& ren();
```

Returns a reference to the pixel format renderer.

```
unsigned width()  const;
unsigned height() const;
```

Returns width and height of the rendering buffer.

```
void reset_clipping(bool visibility);
```

The function resets the clipping. If the `visibility` is `true` the clipping box is set to `(0, 0, width()-1, height()-1)`, if `false`, it sets an invisible box, i.e., `(1,1,0,0)`. In renderer_mclip this function also removes all the clipping areas that were previously added.

IMPORTANT!
If you attach another memory buffer to the rendering_buffer, connected with this particular basic renderer, you **must** call `reset_clipping`, otherwise, the clipping box will be not valid. It's because there's no any “feedback” from the rendering buffer to the renderers, in other words, renderer_base and renderer_mclip don't know anything if the rendering buffer is changed. Having some mechanism of events or delegates would be an overkill in this case.

```
bool clip_box(int x1, int y1, int x2, int y2);
```

Set new clipping box. Only renderer_base has this function. The clipping box **includes** the boundaries, so that, the maximal one is `(0, 0, width()-1, height()-1)`. The clipping box is clipped to the maximal value before setting, so that, it's safe to set a box bigger than `(0, 0, width()-1, height()-1)`.

```
void add_clip_box(int x1, int y1, int x2, int y2);
```

Add new clipping box. Only renderer_mclip has this function. You can add any number of rectangular regions, but they must not overlap. In case of overlapping areas, some elements can be drawn twice or more. The clipping boxes are being clipped by the bounding box `(0, 0, width()-1, height()-1)` before adding, which is done for the sake of efficiency. It also means that calling `reset_clipping(false)` for the renderer_mclip doesn't make any sense because all adding regions will be clipped by an invisible area and will not be actually added. The visible areas also includes the boundaries of the boxes, that is, `add_clip_box(100,100,100,100)` will add a clipping region of 1 pixel size.

```
void clip_box_naked(int x1, int y1, int x2, int y2);
```

Only renderer_base has this function. Set new clipping box without clipping to the rendering buffer size. This function is unsafe and used in the renderer_mclip for fast switching between partial regions when rendering. The purpose of this function is just to avoid extra overhead.

```
bool inbox(int x, int y) const;
```

Check if point `(x, y)` is inside of the clipping box. Only renderer_base has this function.

```
void first_clip_box();
bool next_clip_box();
```

These two functions are used to enumerate all the clipping regions of the renderer. In the renderer_base class they are empty, `next_clip_box()` always return false.

```
const rect& clip_box() const;
int         xmin()     const;
int         ymin()     const;
int         xmax()     const;
int         ymax()     const;
```

Returns the clipping box as a rectangle and as separate integer values. In renderer_mclip the functions always return `(0, 0, width()-1, height()-1)`.

```
const rect& bounding_clip_box() const;
int         bounding_xmin()     const;
int         bounding_ymin()     const;
int         bounding_xmax()     const;
int         bounding_ymax()     const;
```

Returns the bounding clipping box as a rectangle and as separate integer values. In renderer_base the functions always return the same values as the respective ones from the previous group. In renderer_mclip they return the bounding box calculated on the basis of added regions.

```
void clear(const color_type& c);
```

Clears the buffer with the given color without clipping.

```
void copy_pixel(int x, int y, const color_type& c);
```

Set a pixel with clipping.

```
void blend_pixel(int x, int y, const color_type& c, cover_type cover);
```

Blend a pixel with clipping. The behavior is the same as in the pixel format renderers (pixfmt).

```
color_type pixel(int x, int y) const;
```

Returns the value of the pixel with coordinates `(x, y)`. If the point is clipped, the function returns `color_type::no_color()`. For rgba8 it's `(0,0,0,0)`.

```
void copy_hline(int x1, int y, int x2, const color_type& c);
void copy_vline(int x, int y1, int y2, const color_type& c);

void blend_hline(int x1, int y, int x2,
                 const color_type& c, cover_type cover);

void blend_vline(int x, int y1, int y2,
                 const color_type& c, cover_type cover);
```

Draw (copy) or blend horizontal or vertical line of pixels. The behaviour is the same as in pixel format renders (pixfmt), but here you use coordinates of the begin and end of the lines instead of `(x, y, length)`.

```
void copy_bar(int x1, int y1, int x2, int y2, const color_type& c);
void blend_bar(int x1, int y1, int x2, int y2,
               const color_type& c, cover_type cover);
```

Draw (copy) or blend a solid bar (rectangle).

```
void blend_solid_hspan(int x, int y, int len,
                       const color_type& c, const cover_type* covers);

void blend_solid_vspan(int x, int y, int len,
                       const color_type& c, const cover_type* covers);
```

Blend a horizontal or a vertical solid-color span. These functions are used when rendering solid Anti-Aliased polygons.

```
void blend_color_hspan(int x, int y, int len,
                       const color_type* colors, const cover_type* covers);

void blend_color_vspan(int x, int y, int len,
                       const color_type* colors, const cover_type* covers);
```

Blend a horizontal or a vertical color span. The functions are used with different span generators, such as gradients, images, patterns, Gouraud interpolation, etc. They accept an array of colors whose type must be compatible with the used pixel format.

```
void blend_color_hspan_no_clip(int x, int y, int len,
                               const color_type* colors,
                               const cover_type* covers);

void blend_color_vspan_no_clip(int x, int y, int len,
                               const color_type* colors,
                               const cover_type* covers);
```

The same as above, but without clipping. These functions are used in the scanline renderers. The reason to do so is the performance. Tne scanline consists of a number of spans and it's a little bit more efficient to perform clipping when we have information about the whole scanline rather than to clip every span, especially in renderer_mclip.

```
void copy_from(const rendering_buffer& from,
               const rect* rc=0,
               int x_to=0,
               int y_to=0);
```

Copy a rectangular area of the buffer `from` to `this` one considering clipping. It's assumed that the rendering buffer `from` has the same pixel format as the destination `(this)` one. `rc` is an optional rectangle in the `from` buffer, `x_to` and `y_to` — the coordinates of the `rc->x1, rc->y1` mapped to the destination buffer.

A common example

The code below is a very common example of declaring and using of the rendring buffer and low level renderers.

```
    // Typedefs of the low level renderers to simplify the declarations.
    // Here you can use any other pixel format renderer and
    // agg::renderer_mclip if necessary.
    //--------------------------
    typedef agg::pixfmt_rgb24                     pixfmt_type;
    typedef agg::renderer_base<agg::pixfmt_rgb24> renbase_type;
    enum { bytes_per_pixel = 3 };

    unsigned char* buffer = new unsigned char[frame_width *
                                              frame_height *
                                              bytes_per_pixel];
    agg::rendering_buffer rbuf(buffer,
                               frame_width,
                               frame_height,
                               frame_width * bytes_per_pixel);


    pixfmt_type pixf(rbuf);
    renbase_type rbase(pixf);

    rbase.clear(clear_color);

    //. . .
```

At last we can clear the buffer with certain color instead of "manual" calling of `memset()` :-). Also note that unlike these examples, the `stride` value is not obligatory equal to `frame_width * bytes_per_pixel`. Most often there will be some alignment requiremants, for example, the width of Windows bitmaps must be a multiple of 4 bytes.

Primitives and Markers Renderers

The primitives and marker renderers were added to **AGG** to provide you a mecahnism of fast drawing regular, aliased objects, such as lines, rectangles, and ellipses. The marker renderer can draw some shapes commonly used in different scatter plots. If you are not going to use it just skip this section.

Primitives Renderer

The header file: agg_renderer_primitives.h

Declaration

```
template<class BaseRenderer> class renderer_primitives
{
public:
    typedef BaseRenderer base_ren_type;
    typedef typename base_ren_type::color_type color_type;
    //. . .
};
```

Here class RendererBase can be of type renderer_base or renderer_mclip.

Creation

```
renderer_primitives(base_ren_type& ren) :
    m_ren(&ren),
    m_fill_color(),
    m_line_color(),
    m_curr_x(0),
    m_curr_y(0)
{}
```

The cost of creation is minimal, it's only initialization of the pointer to the base renderer object, two colors and initial coordinates that are used in `move_to` and `line_to` functions.

Member functions

```
static int coord(double c);
```

Converts a coordinate of the `double` type to integer one with subpixel accuracy. It just multiplies the value to 256 (by default) and returns the integer part.

```
void fill_color(const color_type& c);
void line_color(const color_type& c);
const color_type& fill_color() const;
const color_type& line_color() const;
```

Set and get the fill and line colors. The colors may have the alpha value that will take effect, i.e., the primitives will be alpha-blended.

```
void rectangle(int x1, int y1, int x2, int y2);
```

Draw a rectangle without filling with the line color. There are regular, pixel coordinates are used. The width of the border is always 1 pixel and cannot be changed.

```
void solid_rectangle(int x1, int y1, int x2, int y2);
```

Draw a filled rectangle without border with the fill color. The coordinates are in pixels.

```
void outlined_rectangle(int x1, int y1, int x2, int y2);
```

Draw a filled rectangle with a border. There are both colors line and fill are used.

```
void ellipse(int x, int y, int rx, int ry);
void solid_ellipse(int x, int y, int rx, int ry);
void outlined_ellipse(int x, int y, int rx, int ry);
```

Draw ellipses, non-filled, filled and filled with a border. The coordinates are integer, in pixels. `rx` and `ry` — radii in pixels.

```
void line(int x1, int y1, int x2, int y2, bool last=false);
```

Draw a bresenham line with **Subpixel Accuracy**. The coordinates are in integer format of 24.8, that is, in the 1/256 pixel units. Flag `last` defines whether or not to draw the last pixel, which is important when drawing consecutive line segments with alpha-blending. There no pixels should be drawn more than once.

```
void move_to(int x, int y);
void line_to(int x, int y, bool last=false);
```

The version of `line()`.

```
const base_ren_type& ren() const { return *m_ren; }
base_ren_type& ren() { return *m_ren; }
```

Returns a reference to the base renderer object.

```
const rendering_buffer& rbuf() const { return m_ren->rbuf(); }
rendering_buffer& rbuf() { return m_ren->rbuf(); }
```

Returns a reference to the rendering buffer attached to the base renderer.

Marker Renderer

The header file: agg_renderer_markers.h

The marker renderer can draw or alpha-blend the following simple shapes:

```
enum marker_e
{
    marker_square,
    marker_diamond,
    marker_circle,
    marker_crossed_circle,
    marker_semiellipse_left,
    marker_semiellipse_right,
    marker_semiellipse_up,
    marker_semiellipse_down,
    marker_triangle_left,
    marker_triangle_right,
    marker_triangle_up,
    marker_triangle_down,
    marker_four_rays,
    marker_cross,
    marker_x,
    marker_dash,
    marker_dot,
    marker_pixel
};
```

Declaration

```
template<class BaseRenderer> class renderer_markers :
public renderer_primitives<BaseRenderer>
{
public:
    typedef renderer_primitives<BaseRenderer> base_type;
    typedef BaseRenderer base_ren_type;
    typedef typename base_ren_type::color_type color_type;
// . . .
};
```

Creation

```
renderer_markers(base_ren_type& rbuf) :
    base_type(rbuf)
{}
```

As you can see, the creation is as simple as the renderer_primitives one.

Member Functions

All the marker functions accept `x`, `y`, and `radius` in pixels. The radius in the `pixel()` marker doesn't have any effect. The fill and line colors are taken from the base class renderer_primitives.

```
void square(int x, int y, int r);
void diamond(int x, int y, int r);
void circle(int x, int y, int r);
void crossed_circle(int x, int y, int r);
void semiellipse_left(int x, int y, int r);
void semiellipse_right(int x, int y, int r);
void semiellipse_up(int x, int y, int r);
void semiellipse_down(int x, int y, int r);
void triangle_left(int x, int y, int r);
void triangle_right(int x, int y, int r);
void triangle_up(int x, int y, int r);
void triangle_down(int x, int y, int r);
void four_rays(int x, int y, int r);
void cross(int x, int y, int r);
void xing(int x, int y, int r);
void dash(int x, int y, int r);
void dot(int x, int y, int r);
void pixel(int x, int y, int);
```

Also, there are some functions for the convenience that basically just use the `switch/case` construction:

```
void marker(int x, int y, int r, marker_e type);
```

Draw a single marker of the given type.

```
template<class T>
void markers(int n, const T* x, const T* y, T r, marker_e type);
```

Draw a number of markers of the given type and radius and coordinates stored in two arrays, `x` and `y`.

```
template<class T>
void markers(int n, const T* x, const T* y, const T* r, marker_e type);
```

Draw a number of markers of the given type with coordinates and radii stored in three arrays.

```
template<class T>
void markers(int n, const T* x, const T* y, const T* r,
             const color_type* fill_colors,
             marker_e type);
```

Draw a number of markers of the given type with coordinates and radii stored in three arrays and of different fill colors.

```
template<class T>
void markers(int n, const T* x, const T* y, const T* r,
             const color_type* fc, const color_type* lc,
             marker_e type);
```

Draw a number of markers of the given type with coordinates and radii stored in three arrays and of different fill and line colors.