Painter and How AGG Works

Article contributed by stippi on Sat, 2005-01-29 06:00 Tags: agg, app_server, drawing, painter

This article is intended to give an overview of the Anti-Grain Geometry 2D engine by Maxim Shemanarev. This C++ graphics engine is currently the back-end of the Haiku `Painter` class used by the app_server as its implementation of `BView` style drawing. By introducing some of the concepts of AGG and how they are used within Painter, I'm hoping to make it easier for others to join the development of Painter and finding ways of improving its efficiency, adding features and/or finding means of adding hardware acceleration.

Template Pipelines

The fundamental concept of AGG is rendering vector paths, which define the enclosed area of a geometrical shape, into arbitrary frame buffers. The first step of this procedure is rasterizing a path to scanlines of coverage values. A "scanline of coverage values" simply means a row of image pixels, for each of which there is a value indicating the geometrical coverage of the path on the implied rectangular area of the pixel (or whatever shape a pixel has in your opinion, just accept the fact that the pixel "covers an area" and that a path might only cover a portion of that area). These coverage scanlines are than used to fill the frame buffer's scanlines with some color. Now, there are many interpretations of "some color", the simplest being "solid color". AGG supports many more types of fills, like image fills, gradient fills, pattern fills and potentially more. These are implemented by different "Span Generators".

To render graphics with AGG, you have to construct something called a "rendering pipeline". The classes in the AGG framework can be plugged together at compile time using C++ templates. Most steps of a pipeline are formed by templating a class to the type of class used in the previous step of the pipeline. The classes implementing the "Vertex Source" interface serve as a good example of this concept.

```
unsigned vertex(double* x, double* y);
void rewind(unsigned pathID);
```

Vertex Source is an iterator interface to obtain all coordinates from an object. The vertex function will put a coordinate in x and y and return a "path command" (for example *agg::path_cmd_line_to*). The iteration is finished when vertex() returns agg::path_cmd_stop. A call to rewind() is supposed to make the next call to vertex() return the first coordinate of the given sub path again. To be usable as a Vertex Source template, a class must simply provide these two functions. For example, the rasterizers in AGG don't support curves within paths, these must first be converted into straight line segments for rendering. There is a curve converter class that handles just that.

```
double cx = 100.0;
double cy = 100.0;
double rx = 25.0;
double ry = 50.0;

agg::path_storage path;

path.move_to(cx - rx, cy);
path.curve4(cx - rx, cy + ry * 0.56,
            cx - rx * 0.56, cy + ry,
            cx, cy + ry);
path.curve4(cx + rx * 0.56, cy + ry,
            cx + rx, cy + ry * 0.56,
            cx + rx, cy);
path.curve4(cx + rx, cy - ry * 0.56,
            cx + rx * 0.56, cy - ry,
            cx, cy - ry);
path.curve4(cx - rx * 0.56, cy - ry,
            cx - rx, cy - ry * 0.56,
```

```
            cx - rx, cy);
path.close_polygon();


agg::conv_curve<agg::path_storage> segmentedPath(path);
```

The above code will construct a Beziér path approximation of an ellipse around the center (`cx`, `cy`) with an x radius of `rx` and a y radius of `ry`. As you can see, the `agg::conv_curve` object is templated to the type of object (`agg::path_storage`), that we use as the initial Vertex Source. The constructor of `agg::conv_curve` than takes an instance of such a class as a parameter. The `conv_curve` object will understand the `path_cmd_curve4` commands returned by `path_storage` and segment the ellipse for rasterization. When we pass the `segmentedPath` object to the rasterization process, the result will be a filled ellipse, but what if we wanted an outline?

```
agg::conv_stroke<agg::conv_curve<agg::path_storage> > outline(segmentedPa
```

The `conv_stroke` is a stroke converter class, it generates a new vector path enclosing the outline around another path. The next step shows how to apply a 2D transformation before rendering the path.

```
agg::trans_affine transformation;
transformation *= agg::trans_affine_rotation(30.0 * 3.1415926 / 180.0);

agg::conv_transform<agg::conv_stroke<agg::conv_curve<agg::path_storage> >
       agg::trans_affine> transformedOutline(outline, transformation);
```

In this example, the `transformedOutline` object is the final one used for rendering. `agg::conv_transform` is another converter which takes a Vertex Source and a transformation as input. (Transformations define a separate interface from Vertex Source, because they are used by other classes in the AGG framework, for example to transform images or gradients.)

Rendering

OK, now that we have explained the template pipelines that are used throughout the AGG framework, we can go a little deeper into rendering something into a frame buffer. A frame buffer in AGG is represented by a Pixel Format object on top of a Rendering Buffer object. The Rendering Buffer caches row pointer offsets (the memory address pointing to the first byte of a frame buffer scanline). The Pixel Format knows how to render internal AGG colors into the frame buffer pixels, for which it uses the scanline coverage information at each pixel. There are quite a few Pixel Format classes defined for the most common colorspaces. There is also a Renderer, Scanline Storage and Rasterizer interface which complete the rendering pipeline. After Vertex Source objects have been added to a Rasterizer, it uses the Scanline object to store the generated coverage values. The Renderer, which is based on top of a Pixel Format, uses the Scanline to render into the frame buffer. This process is handled by the global function `agg::render_scanlines()` defined as such:

```
template<class Rasterizer, class Scanline, class Renderer>
void render_scanlines(Rasterizer& ras, Scanline& sl, Renderer& ren);
```

There are different versions of Scanline classes, for example one which does automatic run length compression, or a "binary version" that effectively removes anti-aliasing information. There are also special `Rasterizer` classes for outlines, so that vector paths can be rendered as outlines without prior stroke conversion. This is faster, but of course not as flexible concerning transformations.

Following is a complete example of building up a pipeline that renders the above ellipse into a `BBitmap`.

```
BRect r(0, 0, 319, 239);
BBitmap* bitmap = new BBitmap(r, 0, B_RGBA32);
```

```
// the AGG rendering buffer representation of the BBitmap
agg::rendering_buffer buffer;
buffer.attach((uint8*)bitmap->Bits(),
              bitmap->Bounds().IntegerWidth() + 1,
              bitmap->Bounds().IntegerHeight() + 1,
              bitmap->BytesPerRow());

// the AGG pixel format matching the format of the BBitmap (B_RGBA32)
typedef agg::pixfmt_bgra32 pixel_format_type;

typedef agg::renderer_base<pixel_format_type> base_renderer_type;

// this renderer uses a solid color for filling paths
typedef agg::renderer_scanline_aa_solid renderer_type;

// the Rasterizer definition
agg::rasterizer_scanline_aa<> rasterizer;
rasterizer.reset();

// constructing the frame buffer pipeline
pixel_format_type pixelFormat(buffer);
base_renderer_type baseRenderer(pixelFormat);
renderer_type renderer(baseRenderer);

// using a packed scanline storage

agg::scanline_p8 scanline;

// add the transformed ellipse outline from above to the Rasterizer
rasterizer.add_path(transformedOutline);

// defining the color used for filling
renderer.color(agg::rgba(0, 0, 0, 1));

// clear out the bitmap with some background color
renderer.clear(agg::rgba(1, 1, 1, 1));

// triggering the actual rendering
agg::render_scanlines(rasterizer, scanline, renderer);
```

Et voilà, we have rendered our ellipse into a `BBitmap`.

The Painter

And now – on for the subjects more immediately interesting to the Haiku Painter and app_server. While reading this far, you might have realized that the AGG template "plug and play" achieves a compile time flexibility, but not a runtime flexibility. It is always possible to create polymorphic wrappers implementing the required interfaces. AGG is optimized for speed and memory usage by default, while leaving it up to the user of the framework to achieve runtime flexibility should it be needed.

In the `Painter` class, I needed runtime flexibility most urgently at the Pixel Format level. `BView` supports many drawing modes, which define how a source pixel is drawn over a destination pixel. Patterns can be used to define masks. The High Color and Low Color of a BView are immediately important to this. There are multiple options how to achieve this behavior using AGG. One option is to use a Rasterizer and Renderer which use Span Generators that generate patterns. I was not sure about the speed issues involved with this approach, so I just went with what appeared to be the most straight forward solution: Making the Pixel Format runtime flexible. If you look into the code, you will notice this is done using the `PixelFormat` class, which forwards all AGG Pixel Format functions to the appropriate function of one of the `DrawingMode` classes. Classes of `DrawingMode` know how to blend colors according to a `BView` drawing_mode and a given pattern, High Color and Low Color.

Bitmaps...

Bitmaps are handled in Painter by constructing a rendering pipeline that uses a "Span Generator" to

determine the fill colors. AGG offers many Span Generators for color gradients, patterns, and bitmaps. Consequently Painter sets up a pipeline with a bitmap Span Generator and a transformation stage to handle scaling of bitmaps, then renders a path enclosing the area in which the bitmap is to appear in the target frame buffer. The bitmap transformation is set up so that the correct part of the bitmap is located within the surrounding path. So cropping bitmaps is also handled. The nearest neighbor interpolator is used for the transformation of the bitmap. As the pipeline would also handle any other affine transformations of the bitmap such as rotation and skew, it is probably slower than the original app_server's implementation, which only handles scaling.

Support for more colorspaces is most likely achieved best by modifying the nearest neighbor interpolator to also handle colorspace conversion. Alternatively, an extra conversion step could be inserted into the pipeline before the transformation stage. The Painter DrawBitmap() function is targeted towards handling all cases of bitmap drawing, but special versions for non-scaled bitmaps are there to improve speed.

... and Fonts

Painter implements text rendering also through AGG, which in turn is based on FreeType to read and convert font glyphs into vector paths. From there on, the normal vector path rasterization pipelines are used. AGG implements font caching to avoid extracting glyphs from the font files more often than necessary. As with bitmaps, text rendering in Painter is currently targeted for the general case. Rendering horizontal text is handled as a special case bypassing the vector pipeline and using Freetype generated bitmaps which leads to significant speedup. In this approach, the font cache is used at the scanline level, which means there is no real vector path rasterization most of the time. AGG provided the necessary code for this already. Because of hinting, which means that the shapes of glyphs are slightly distorted so as to fit onto complete pixels, the result of rendering a glyph is the same most of the time, so it can be cached in its bitmap form.

More Speed

So how can Painter be improved? Many cases of acceleration are already covered, including hardware acceleration where the driver interface has support for it. This is done by bypassing the vector rendering for simple drawing jobs like rendering horizontal or vertical lines, filling or inverting rectangles and so on - provided the current drawing_mode is suitable. Although Painter special cases the unscaled drawing of B_RGB(A)32 and B_CMAP8 bitmaps, it would result in great speed ups if drawing scaled bitmaps had a special implementation as well.

It is worth noting that Painter as the drawing backend of the app_server is not necessarily guilty of it's yet disappointing slowness. When compared to BeOS R5 and later versions, that is. One possible cause for the slow performance (by BeOS standards) may be our implementation of *redraw cycles*. Currently, it is done on a per window basis, while it seems to be on a per view basis in BeOS. Our implementation could lead to unnecessary drawing of already outdated view clipping regions, while that is more likely to be avoided with a more fine grained "per view" approach of update sessions. But with these considerations, I am leaving the domain of Painter and enter that of the app_server at large, so I better stop here.

Concluding this article, it should hopefully be much easier to understand the Painter code and design. Should there be any questions, I am more than happy to answer them! Just drop me a line at superstippi at gmx dot com!