Introduction

Overview and Basic Concepts

Brief Overview

Yet Another Invention of the Wheel

Anti-Grain Geometry (**AGG**) is a general purpose graphical toolkit written completely in standard and platform independent **C++**. It can be used in many areas of computer programming where high quality 2D graphics is an essential part of the project. For example, if you render 2D geographic maps **AGG** is a must. **AGG** uses only **C++** and standard C runtime functions, such as **memcpy, sin, cos, sqrt**, etc. The basic algorithms don't even use **C++ Standard Template Library**. Thus, **AGG** can be used in a very large number of applications, including embedded systems.

On the other hand, **AGG** allows you to replace any part of the library, if, for example, it doesn't fit performance requirements. Or you can add another color space if needed. All of it is possible because of extensive using of **C++ template** mechanism.

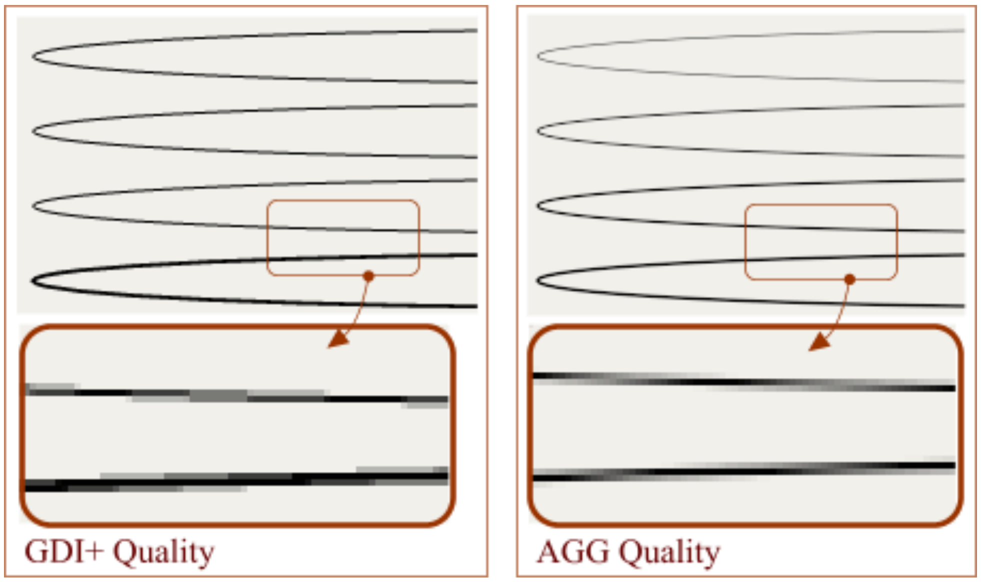
Anti-Grain Geometry is not a solid graphic library and it's not very easy to use. I consider **AGG** as a **"tool to create other tools"**. It means that there's no **"Graphics"** object or something like that, instead, **AGG** consists of a number of loosely coupled algorithms that can be used together or separately. All of them have well defined interfaces and absolute minimum of implicit or explicit dependencies.

Gentle Criticism

Most of the graphic libraries have a single class like **“Graphics”** in GDI+, that has hundred or even thousands of functions. This object can exist implicitly, like in OpenGL. Anyway, all commonly used graphical tool kits, including Java2D, DisplayPDF, SVG, and other very good ones have this kind of a class explicitly or implicitly. That's simple and in some cases quite suitable, but always very restrictive. It works well only in simple cases, at least I haven't seen a graphical library that would completely fit all my needs. Moreover, all that kinds of libraries or standards have a syndrome of giantism. Most of the functionality is never used, but some simple things are impossible to achieve. Herein, the graphical engines (or libraries) typically weigh tons of mega-bytes. If you take the most advanced SVG viewer, Adobe SVG, it works well only with simplest primitives. As soon as you try to use some advanced things, like interactive SVG with different graphical filters, you will have memory leaks, or even crashes. It's not because it has bad design, it's because the proposed possibilities assume extremely complex design. The design itself becomes an **NP-complete** task, which is impossible to perceive by a human mind as impossible to perceive the infinity.

The Proposal

The primary goal of **Anti-Grain Geometry** is to break this ancient as mammoth's manure tradition and show you the beauty of stability, lightness, flexibility, and freedom. The basic concepts can seem not very conventional at the beginning, but they are very close to the ones used in **STL**. But there's a big difference too. **STL** is a general **C++** tool while **AGG** is **C++** graphics. You usually use **STL** in your applications directly as a convenient toolkit. I wouldn't say it's a good idea to use **AGG** in the very same way. A good idea is to create a lightweight, problem oriented wrapper over **AGG** to solve your particular tasks. How can that be different from that very GDI+ then? The first thing is that you have total control upon that wrapper. **Anti-Grain Geometry** just provides you a set of basic algorithms and flexible design with the minimum of implicit or explitit dependencies. You and only you define the interface, conversion pipelines, and the form of output. You can even simulate a part of any existing graphical interface. For example, you can use **AGG** rasterizer to display graphics on the screen and direct Windows GDI calls for printing, incorporating it into a single API. Not convincing? Look at the quality of rendering in **GDI+** and **AGG**:

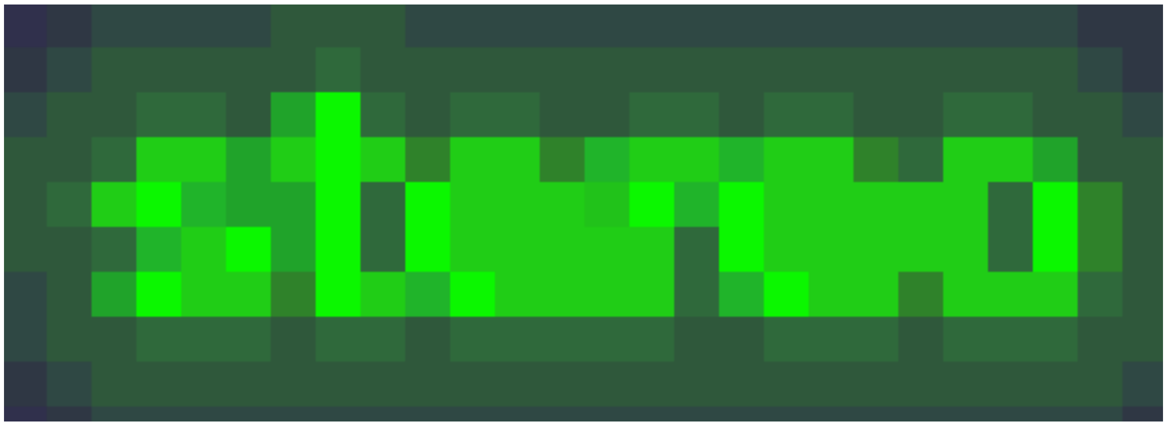


Quality of Rendering

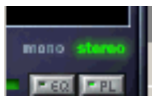
But most of all, your applications become absolutely portable, if your design is smart enough. **AGG** can be also a tool to combine different outputs in a uniform API. Particularly, you can use **AGG** to generate raster images on the server side in your Web-Based applications. And it all can be **cross-platform!**

Anti-Aliasing and Subpixel Accuracy

Anti-Aliasing is a very well known technique used to improve the visual quality of images when displaying them on low resolution devices. It's based on the properties of the human vision. Look at the following picture and try to guess what it means.

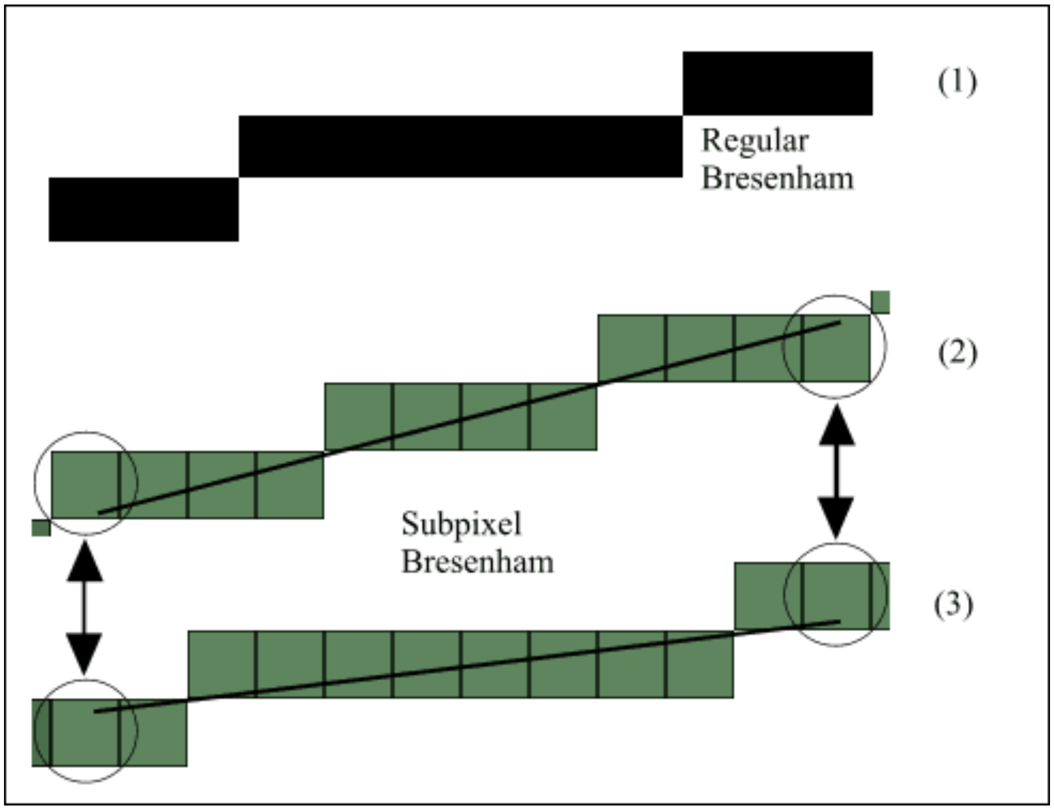


Well, it's a word drawn with **Anti-Aliasing**. In terms of Kotelnikov-Shannon's theorem, the maximal frequency of the image is far above of the Shannon limit.



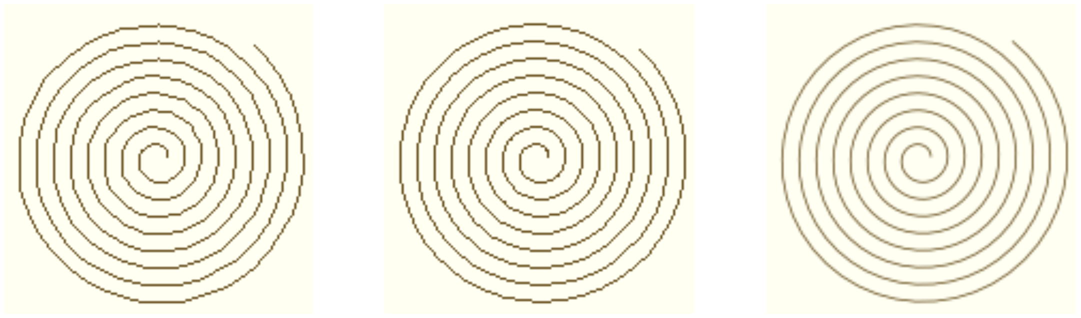
Now look at the same picture that has normal size and **within the context**. You easily recognize word **“stereo”**. However, the pictrures are exactly the same. The first one is just an enlarged version of the last one. This very property allows us to reconstruct missing information on the basis of accumulated experience. **Anti-Aliasing** doesn't make you see better, it basically makes you brain work better and reconstruct missing details. The result is great. It allows us to draw much more detailed maps for example.

But the point is not only in **Anti-Aliasing** itself. The point is we can draw primitives with **Subpixel Accuracy**. It's especially important for the visual thickness of the lines. First, let us see that even with simple Bresenham line interpolator we can achieve a better result if we use **Subpixel Accuracy**. The following picture shows enlarged results of the simple Bresenham interpolator.



A Bresenham Line Rendered with Subpixel Accuracy

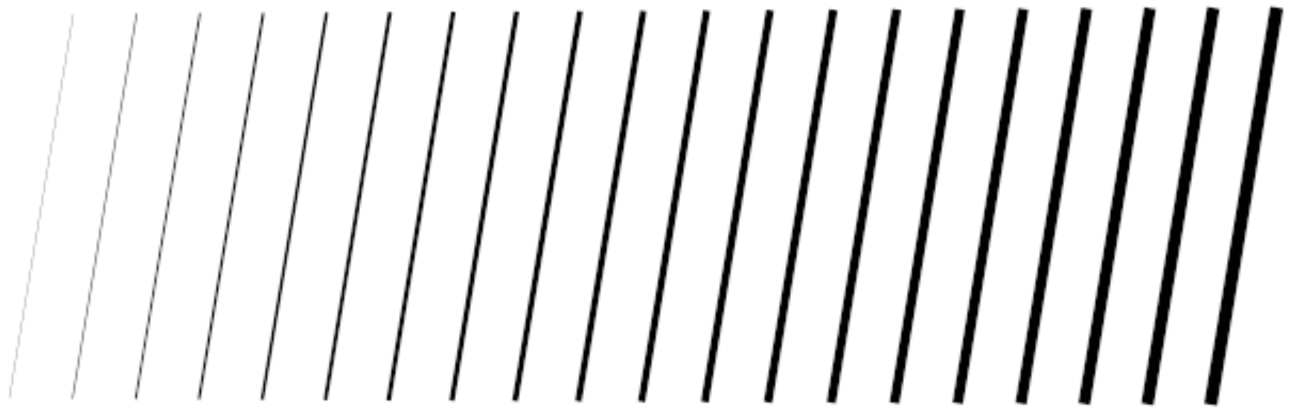
Consider cases (2) and (3). The thin black lines are what we need to interpolate. If we use **Subpixel Accuracy** we will really have two different sets of pixels displayed, despite of the fact that the begins and ends of both lines fall into the same pixels. And the lines have really different tangents, which is very important. If we use a classical Bresenham, without considering the **Subpixel Accuracy** we will see result (1) in all cases. That's especially important to approximate curves with short line segments. But if we use **Anti-Aliasing** plus **Subpixel Accuracy** we can do much better. Look at that difference.



Here all three spirals are approximated with short straight line segments. The left one is drawn using regular integer Bresenham, when the coordinates are rounded off to pixels (you will have a similar result if you use Winwows GDI MoveTo/LineTo, for example). The one in the middle uses a modified integer Bresenham with precision of 1/256 of a pixel. And the right one uses the same 1/256 accuracy, but with **Anti-Aliasing**. Note that it's very important to have a possibility of real subpixel positioning of the line segments. If we use regular pixel coordinates with **Anti-Aliasing**, the spiral will look smooth but still, as ugly as the one on the left.

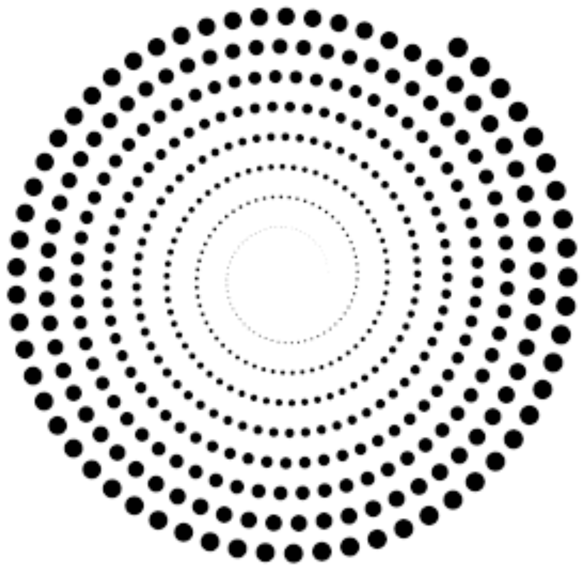
The **Subpixel Accuracy** is even more important to control the visual thickness of the lines. It's possible only if we have good algorithms of **Anti-Aliasing**. On the other hand, there's no much sense of **Anti-Aliasing** if can set the line width with the discretness of one pixel only. **Anti-Aliasing** and **Subpixel Accuracy** always work in cooperation.

Modern displays have resolutions of at most 120 DPI, while **Subpixel Accuracy** is actual up to 300 DPI. The following picture shows lines with thickness starting from 0.3 pixels and increasing by 0.3 pixel.

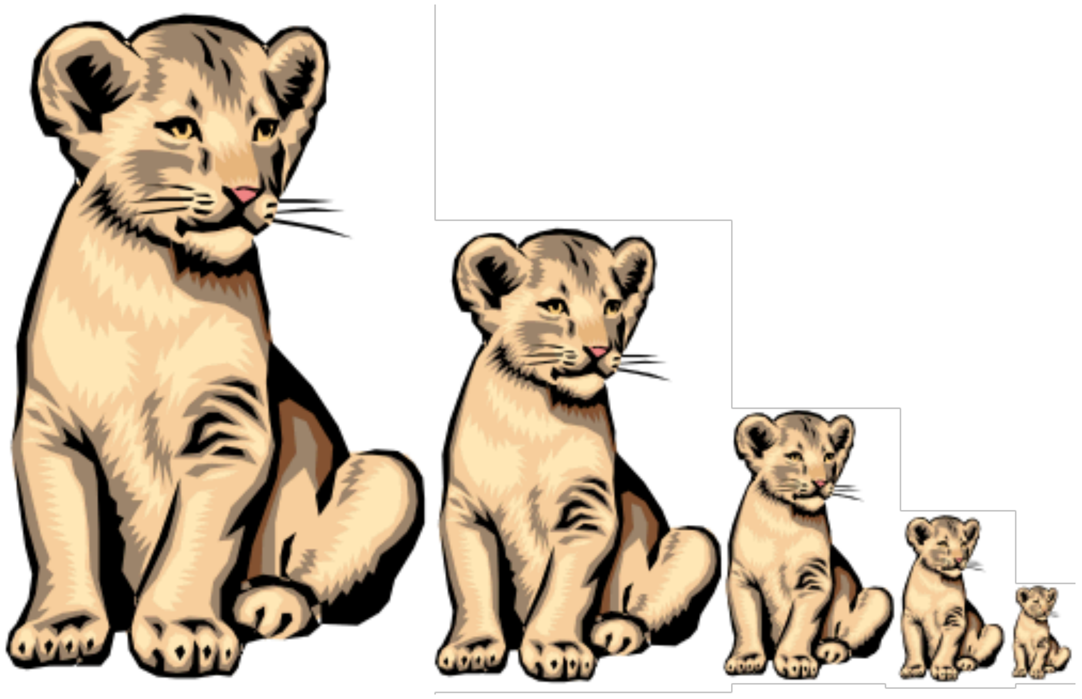


Lines Rendered with Anti-Aliasing and Subpixel Accuracy

There are two more examples of rendering with **Subpixel Accuracy**.



Circles Rendered with Anti-Aliasing and Subpixel Accuracy



Cute Lions

Note that the appearance of the small ones remains consistent despite of lost details.

Basic Concepts

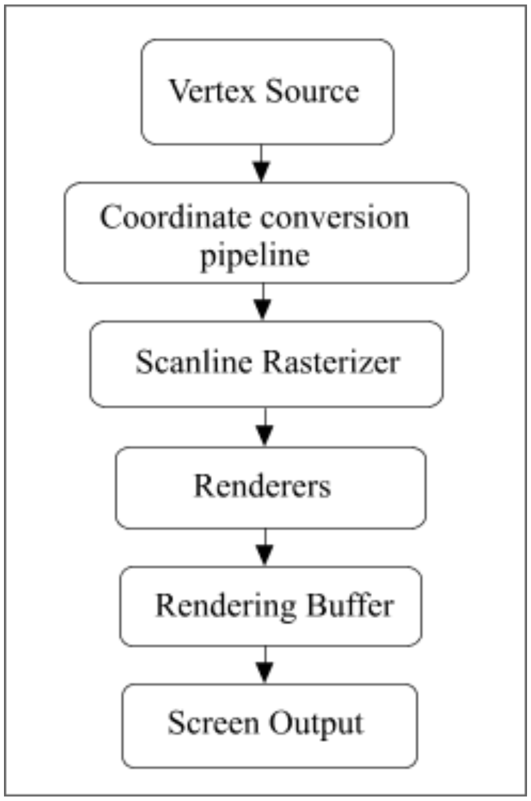
Design of the Library

Anti-Grain Geometry is designed as a set of loosely coupled algorithms and class templates united with a common idea, so that all the components can be easily combined. Also, the template based design allows you to replace any part of the library without the necessity to modify a single byte in the existing code.

Also **AGG** is designed keeping in mind extensibility and flexibility. Basically I just wanted to create a toolkit that would allow me (and anyone else) to add new fancy algorithms very easily.

AGG does not dictate you any style of its use, you are free to use any part of it. However, **AGG** is often associated with a tool for rendering images in memory. That is not quite true, but it can be a good starting point in studying. The tutorials describe the use of **AGG** starting from the low level functionality that deals with frame buffers and pixels. Then you will gradually understand how to abstract different parts of the library and how to use them separately. Remember, the raster picture is often not the only thing you want to obtain, you will probably want to print your graphics with highest possible quality and in this case you can easily combine the “vectorial” part of the library with some API like Windows GDI, having a common external interface. If that API can render multi-polygons with non-zero and even-odd filling rules it's all you need to incorporate **AGG** into your application. For example, Windows API PolyPolygon perfectly fits these needs, except certain advanced things like gradient filling, Gouraud shading, image transformations, and so on. Or, as an alternative, you can use all **AGG** algorithms producing high resolution pixel images and then to send the result to the printer as a pixel map.

Below is a typical brief scheme of the **AGG** rendering pipeline.



Typical Scheme of the Rendering Pipeline

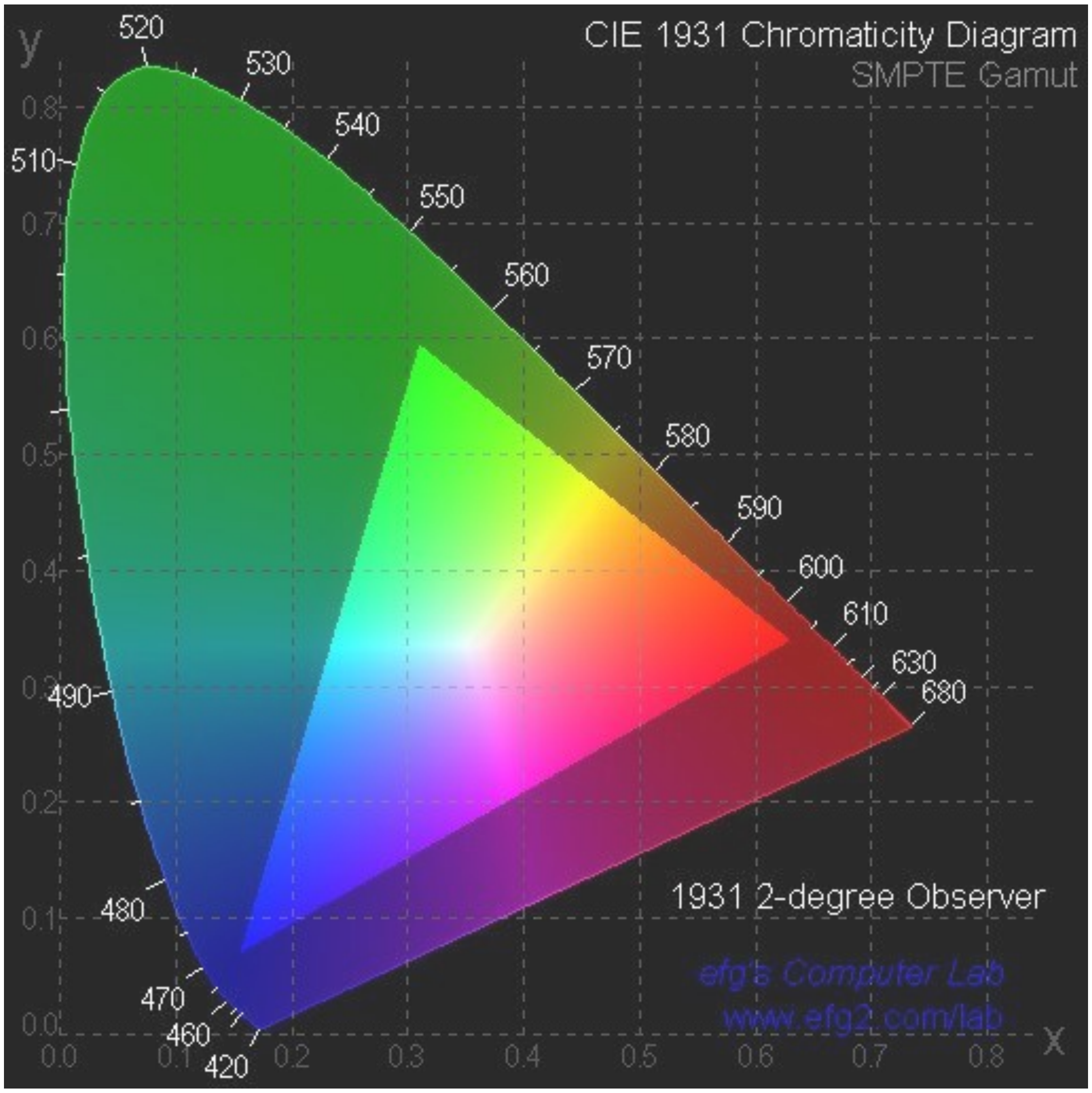
Please note that any component between the "Vertex Source" and "Screen Output" is not mandatory. It all depends on your particular needs. For example, you can use your own rasterizer, based on Windows API. In this case you won't need the AGG rasterizer and renderers. Or, if you need to draw only lines, you can use the AGG **outline** rasterizer that has certain restrictions but works faster. The number of possibilities is endless.

Here:

- **Vertex Source** is some object that produces polygons or polylines as a set of consecutive 2D vertices with commands like "MoveTo", "LineTo". It can be a container or some other object that generates vertices on demand.
- **Coordinate conversion pipeline** consists of a number of coordinate converters. It always works with vectorial data (X,Y) represented as floating point numbers (double). For example, it can contain an affine transformer, outline (stroke) generator, some marker generator (like arrowheads/arrowtails), dashed lines generator, and so on. The pipeline can have branches and you also can have any number of different pipelines. You also can write your own converter and include it into the pipeline.
- **Scanline Rasterizer** converts vectorial data into a number of horizontal scanlines. The scanlines usually (but not obligatory) carry information about **Anti-Aliasing** as "coverage" values.
- **Renderers** render scanlines, sorry for the tautology. The simplest example is solid filling. The renderer just adds a color to the scanline and writes the result into the rendering buffer. More complex renderers can produce multi-color result, like gradients, Gouraud shading, image transformations, patterns, and so on.
- **Rendering Buffer** is a buffer in memory that will be displayed afterwards. Usually but not obligatory it contains pixels in format that fits your video system. For example, 24 bits B-G-R, 32 bits B-G-R-A, or 15 bits R-G-B-555 for Windows. But in general, there're no restrictions on pixel formats or color space if you write your own low level class that supports that format.

Colors, Color Spaces, and Pixel Formats

Colors in **AGG** appear only in renderers, that is, when you actually put some data to the rendering buffer. In general, there's no general purpose structure or class like **"color"**, instead, **AGG** always operates with concrete color space. There are plenty of color spaces in the world, like RGB, HSV, CMYK, etc., and all of them have certain restrictions. For example, the RGB color space is just a poor subset of colors that a human eye can recognize. If you look at the full **CIE Chromaticity Diagram**, you will see that the RGB triangle is just a little part of it.



CIE Chromaticity Diagram and the RGB Gamut

In other words there are plenty of colors in the real world that cannot be reproduced with RGB, CMYK, HSV, etc. Any color space except the one existing in Nature is restrictive. Thus, it was decided not to introduce such an object like **"color"** in order not to restrict the possibilities in advance. Instead, there are objects that operate with concrete color spaces. Currently there are agg::rgba and agg::rgba8 that operate with the most popular **RGB** color space (strictly speaking there's RGB plus Alpha). The RGB color space is used with different pixel formats, like 24-bit RGB or 32-bit RGBA with different order of color components. But the common property of all of them is that they are essentially RGB. Although, **AGG** doesn't explicitly support any other color spaces, there is at least a potential possibility of adding them. It means that all class and function templates that depend on the **"color"** type are parameterized with the **"ColorT"** argument.

Coordinate Units

Basically, **AGG** operates with coordinates of the output device. On your screen there are pixels. But unlike many other libraries and APIs **AGG** initially supports **Subpixel Accuracy**. It means that the coordinates are represented as **doubles**, where fractional values actually take effect. **AGG** doesn't have an embedded conversion mechanism from world to screen coordinates in order not to restrict your freedom. It's very important where and when you do that conversion, so, different applications can require different approaches. **AGG** just provides you a transformer of that kind, namely, that can convert your own view port to the device one. And it's your responsibility to include it into the proper place of the pipeline. You can also write your own very simple class that will allow you to operate with millimeters, inches, or any other physical units.

Internally, the rasterizers use integer coordinates of the format 24.8 bits, that is, 24 bits for the integer part and 8 bits for the fractional one. In other words, all the internal coordinates are multiplied by 256. If you intend to use **AGG** in some embedded system that has inefficient floating point processing, you still can use the rasterizers with their integer interfaces. Although, you won't be able to use the floating point coordinate pipelines in this case.

AGG Building and Coding Notes

Anti-Grain Geometry doesn't have any rich and automated environvents to build. **AGG** mantra is **"It just compiles and works"**. It doesn't have any installation packages either. Maybe it's not very good from the point of view of automated configuring and making of applications (like it's commonly used on Unix), but all you need to do is just add **AGG** source files into your distribution package as if they were your files. As a benefit of this approach, you won't have any problems with configuration files and endless **#ifdef…#elif…#endif**. This is possible because **AGG** has absolute minimum of external dependencies. For Unix there are the simplest possible **Makefiles** to build the **.a** library, for Windows there's no library created at all. All the demo examples just include the necessary source files. This practice allows for more convenient debugging process; in fact, almost all the examples are actually used to implement and debug the algorithms. It also advantages stability of the library, because all the algorithms suffer very deep testing in the conditions near to operational.

NOTE
If you want to use **AGG** in Windows Visual C++ environment, please note that there's no **"stdafx.h"** file used. It's **Microsoft** specific and not a part of C/C++ standard libraries, but **Microsoft** just enforces to use it. To successfully use **AGG** in Visual C++ projects don't forget to turn off the **"Precompiled Headers"** option for all **AGG** source files. Besides, if you link **AGG** with static **MFC** you will probably have duplicating `new` and `delete` operators when linking. It's not because of **AGG**, it's because of **MFC**. You will have the very same problem when you try to use any other **C++** code that calls `new/delete` and doesn't include **stdafx.h**. To resolve this situation follow the **Microsoft recommendations** or just search in **Google** for **"148652 LNK2005"**.

As it was mentioned above, **AGG** uses **C++** template mechanism very actively. However, it uses only well known and proven language constructions. A good compatibility is one of the primary aspirations. **C++** gurus can be suprised that **AGG** doesn't use **STL**, for example. It's done intentionally, in order not to have extra dependencies where the necessity of **STL** containers is very little. Of course, it doesn't prevent you from using **STL** or any other popular tools in a higher level. **AGG** is designed to have absolute minumum of potential conflicts with existing **C++** libraties, tools and technologies.

About this Manual

As it was said before **AGG** provides many different levels of functionality, so that you can use it in many different ways. For example, you may want to use **AGG** rasterizers without the scanline renderers. But for the sake of consistency and graduality we will start from the very beginning and describe all the functionality with examples. This approach might be slower than some “Quick Start”, but it will allow you to understand the conceps of the design. It is really useful because you will know how to replace certain classes and algorithms with your own, or how to extend the library. Particularly, the scanline renderers are platform independent, but not the fastest. You may want to write your own, optimized ones, but oriented to some hardware archtecture, like SSE2.