

## 介绍和推荐 AGG

最近在研发跨平台的 GIS engine, GDI 部分当然不能只用 WINDOWS 的 GDI PLUS, 一个跨平台的 2D Engine 的选择就非常重要。网上查了 N 久, 感觉 AGG 非常全面, 效果也很好, 而且也是跨平台的。非常适合我的问题领域。

### 一、AGG 是什么

AGG, 全名: Anti-Grain Geometry, 是一个开源的、高效的 2D 图形库, 它的网站:

<http://www.antigrain.com/> 。

### 二、AGG 的特点

AGG 的功能与 GDI+ 的功能非常类似, 但提供了比 GDI+ 更灵活的编程接口, 其产生的图形的质量也非常高, 而且它是跨平台的, 其宣传可以在非常多的操作系统上运行, 我至少在 Windows、Wince、Linux 三个平台测试通过。

### 三、AGG 的功能

- 1、支持 ALPHA、GAMMA 等变色处理, 以及用户自定义的变色处理;
- 2、支持任意 2D 图形变换;
- 3、支持 SVG 和 PostScript 描述, 适于网上图形生成;
- 4、支持高质量的图形处理, 支持反走样插值等高级功能;
- 5、支持任意方式的渐变色处理;
- 6、支持所有颜色格式;
- 7、支持对位图的多种处理;

- 8、支持直线的多种处理，类似于 GDI+；
- 9、支持 GPC，即通用多边形裁剪方法；
- 10、支持多种字体输出，包括汉字的处理；

#### 四、AGG 的使用

在设计上，它是师出 Boost 库，在其中使用了大量的现代标准 C++ 语言的语法规则，包括模板、仿函数等处理，但是为了能在更多的平台上使用，它并没有直接使用 Boost 和 STL 库，而是自己实现了部分 STL 功能。

AGG 将图形功能分为几个层次，每一层次都可以由用户自己改动和扩充，作为 AGG 的使用者，可以使用它的全部功能，也可以只使用它的部分功能；作为图形的接口，它允许用户在不同层次上对它进行访问。

以下是一个典型的作图分层：

- 1、定义矢量作图源数据（其定义类似于 PostScript）；
- 2、提供变换管道（包括坐标变换，以及其它可能的数据变换）；
- 3、将数据转为水平线光栅化数据；
- 4、将数据转为带颜色格式的输出缓冲区数据；
- 5、输出位图或像素数组。

AGG 的设计定位是为其它开发工具提供工具，因此，其使用是灵活但不容易，但通过它提供的几十个例子，可以很容易地入门（但精通不是易事）。

为了使用 AGG，有两种方式，一种方式是直接使用它，一种方式是为其定义一个封装接口。现在网上有一些这方面的封装接口例子（比如说，定义一个仿 GDI+ 的封装接口），可以在它的论坛上找到。本人也编写了一个封装类组，但主要是为了我的项目用的，以

后我会提供一个更通用的封装类组。

## 五、使用 AGG 的软件

我前段时间上网查 2D 图形功能，结果发现了许多库都是或多或少地使用了 AGG，于是才对 AGG 感兴趣，我看中了它的跨平台性，以及运行速度和内存占用等。现在提供几个在内部使用 AGG 的比较有名的开源项目：

Inkspace:一个跨平台的 SVG 编辑软件，部分使用了 AGG 的功能（还使用了 ArtLib 的功能，我以后会对 ArtLib 作介绍）。

wxArt2d:一个基于 Wxwindow 的 2D 图形编辑软件，提供了 AGG 可以一种作图选择。

vcf:一个 C++ 框架库，使用 AGG 作为图形输出。

这些库都可以到 <http://www.sf.net/> 上找到。

## 一、概要

AGG 是一个用标准的平台无关的 C++ 写成的通用图形工具包。它可以应用在计算机程序中需要高质量的 2D 图形的许多方面。例如，AGG 可以用于渲染 2D 地图。AGG 只使用了 C++ 和标准 C 的函数，如 memcpy, sin, cos, sqrt 等。基本的算法甚至没有使用 C++ Standard Template Library。因此，AGG 能够在大量的应用软件中使用，包括嵌入式系统中。

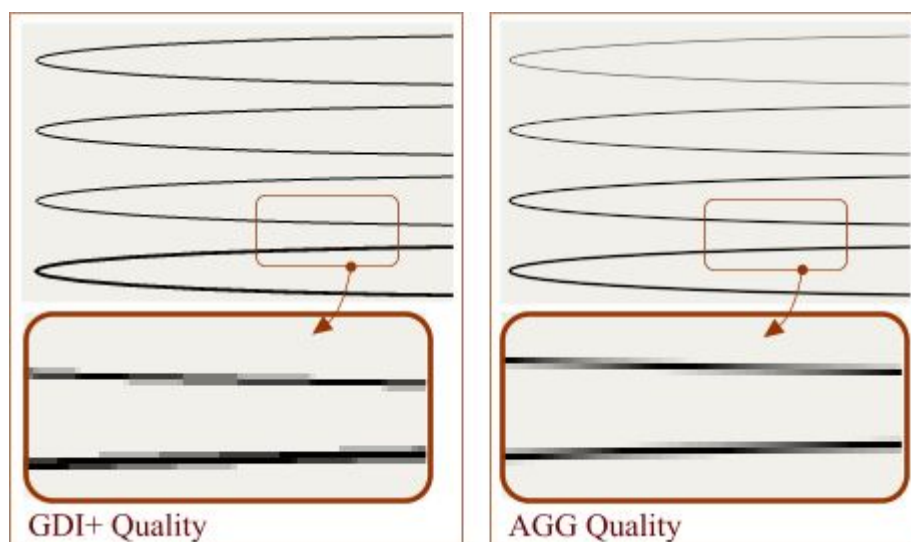
另一方面，AGG 允许对库的一部分进行替换，比如在它不能适应性能的要求时。如果需要，你也能够添加其他的颜色空间。因为 AGG 是基于 C++ 的模板机制的。

AGG 不是一个紧密的图形库，它并不容易使用。这是由于考虑到 AGG 是一个“创建其

他工具的工具”。这意味着，没有“Gphcis”对象或是其他类似的东西。取而代之的是许多组织松散的、能单独或是组合使用的算法。这些算法有着定义良好的接口，它们之间有着尽可能最小的隐式或显式的依赖关系。

## 1.建议

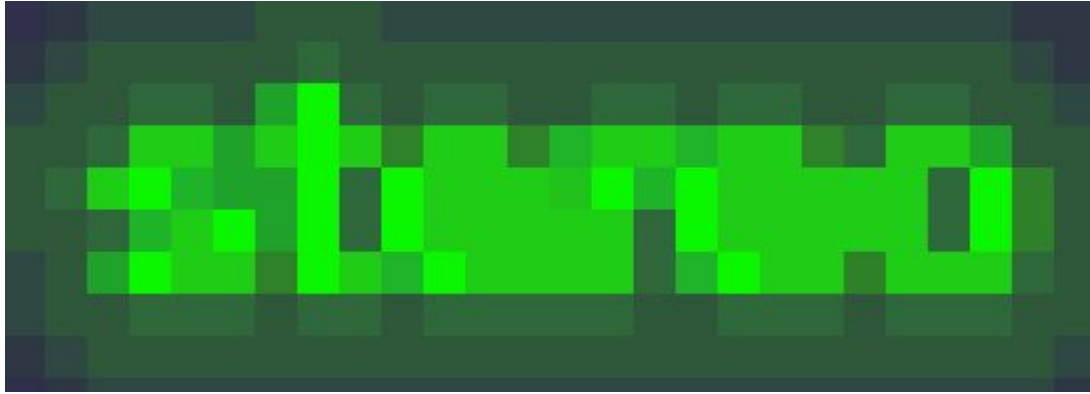
AGG 最主要的目标是打破一下旧的传统（针对 GDI，GDI + 的一些缺点，比如过于庞大，不能修改以满足要求等）。AGG 所要展示的是稳定，轻巧，灵活和自由。与 STL 不同，AGG 是 C++图形库。STL 通常能够很方便的直接在程序中使用。而并不建议象使用 STL 那样使用 AGG。更好的使用 AGG 的方法是，可以针对需要处理的问题对它进行简单的封装后进行使用。这与 GDI + 有什么不同？首先，可以对封装后的 AGG 完全进行控制。AGG 只是提供了一系列基本的算法，而这些算法间有着很少的的关联。可以只定义接口，转换管道和输出的形式。甚至可以替代任何已经存在的图形接口的一部分。例如，可以使用 AGG 的光栅化器来在屏幕上显式图形并直接调用用 GDI 来打印。下图是 GDI+和 AGG 的渲染品质的比较。



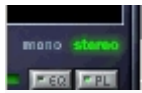
但最重要的是，如果程序设计的足够灵活，这可以使得程序变成完全可移植的。

## 2.反锯齿和亚像素精度

反锯齿是用于在低分辨率设备上显示图象时改善视觉品质的一种广为人知的技术。它基于人类的视觉特性。看看下图，尝试猜测一下是什么意思。

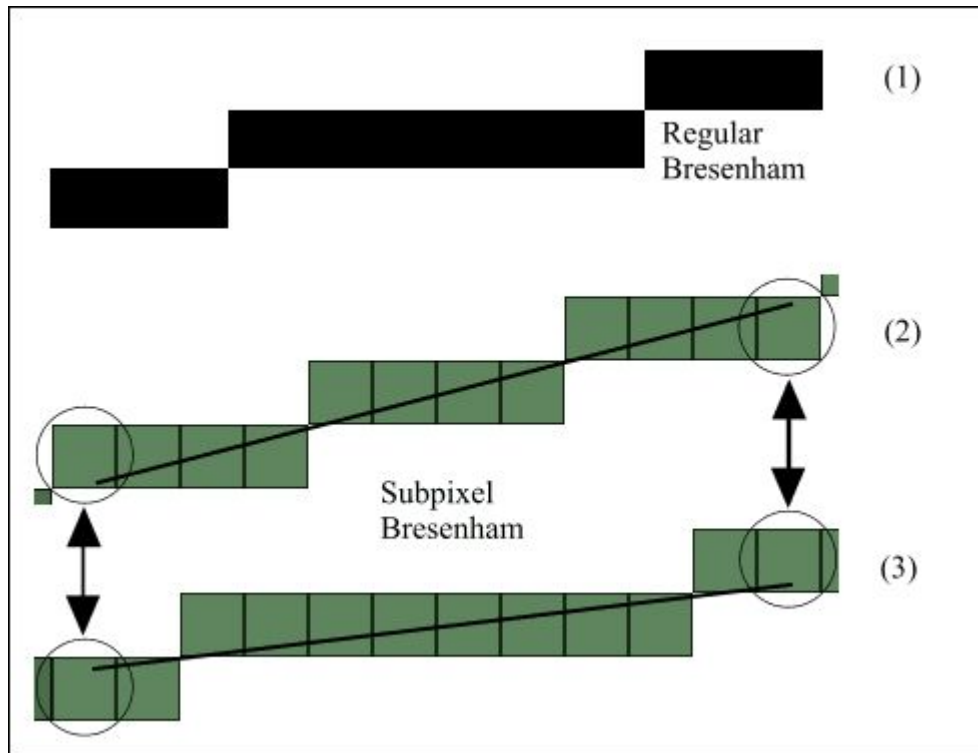


这是一个用反锯齿方式绘制的词。



现在来看看正常大小的同一幅图。能够很容易的分辨出这是“stereo”。然而这是两幅相同的图。只是第一幅比第二幅大。反锯齿并没有让它看起来更清楚，它只是让你的大脑更容易推断出是什么。

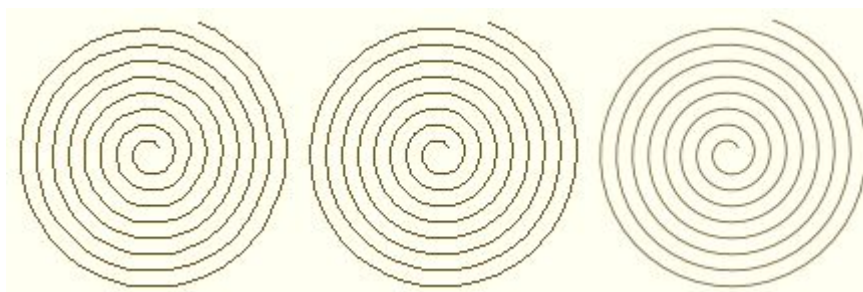
AGG 中不仅仅使用了反锯齿，它还利用了亚像素精度的技术。下图中显示了使用简单的 Bresenham 差值放大后的结果。



如图中（2）和（3），由于利用了亚像素精度，虽然线的起始点和结束点是同样的，但是可以表示两种不同的线。而如果不利用亚像素精度，同样的起始点和结束点则只能表示一种线，如（1）。

如果同时利用亚像素精度和反锯齿，就能够获得更好的效果。

看看下面的图有什么不同：

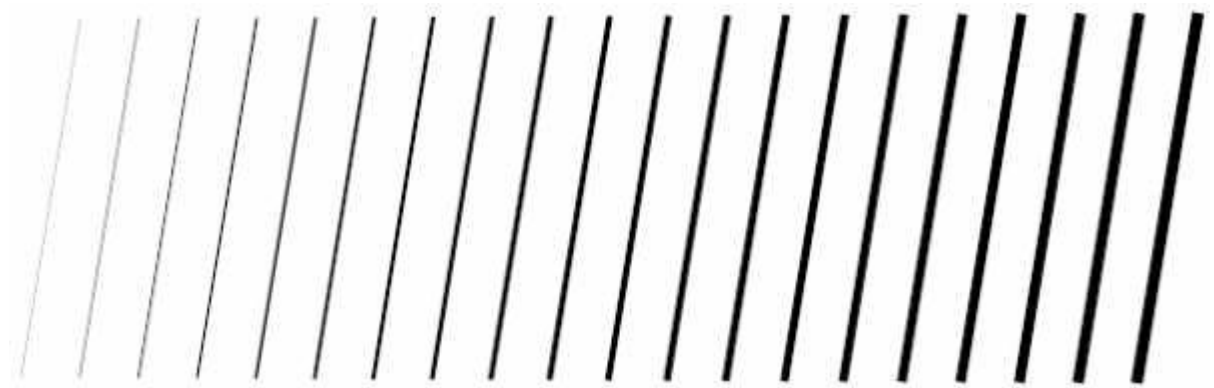


这三个螺旋形可以近似的看成是由短的直线段组成的。左边的这幅图采用的是正常的 Integer Bresenham 方法（使用 Windows GDI 的 LineTo 和 MoveTo 函数也可以得到类似的

效果)。中间的图采用了改进后的精度为  $1/256$  像素的 Integer Bresenham 方法。而右边的图同样使用了  $1/256$  像素的精度，但是同时还使用了反锯齿。请注意，将线段中亚像素真实定位的能力非常重要。如果我们对正常像素坐标使用反锯齿，那么螺旋形看起来会更平滑，但仍然同左边的图一样难看。

亚像素精度甚至比控制线的视觉厚度 (visual thickness) 更加重要。只有当我们有一个好的反锯齿算法时，良好的精确性才能成为可能。另一方面，如果我们可以只用一个点的离散性来确定线的宽度，那么反锯齿就没有什么用处了。反锯齿和亚像素精确度总是协同使用的。

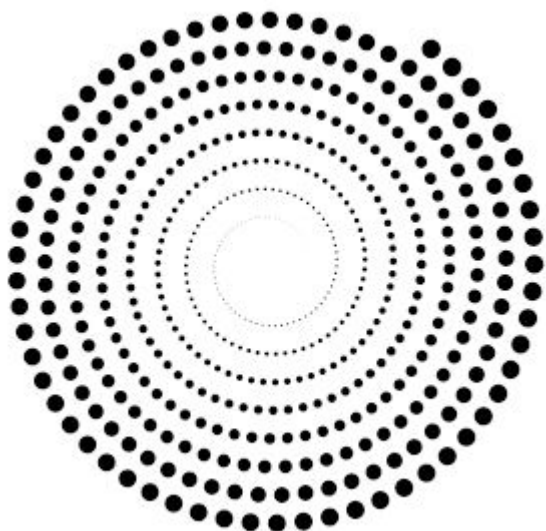
现代显示器的分辨率最多是 120DPI，而利用亚像素精度可以达到 300DPI。下面的图片显示的线的宽度从 0.3 像素开始，依次递增 0.3 像素。



用反锯齿和亚像素精度渲染的线

下面是两个用亚像素精度进行渲染的例子：





用反锯齿和亚像素精度渲染的圆形



可爱的狮子

请注意：那些小的狮子虽然丢失了一些细节，但还是保持了与较大狮子的一致性。



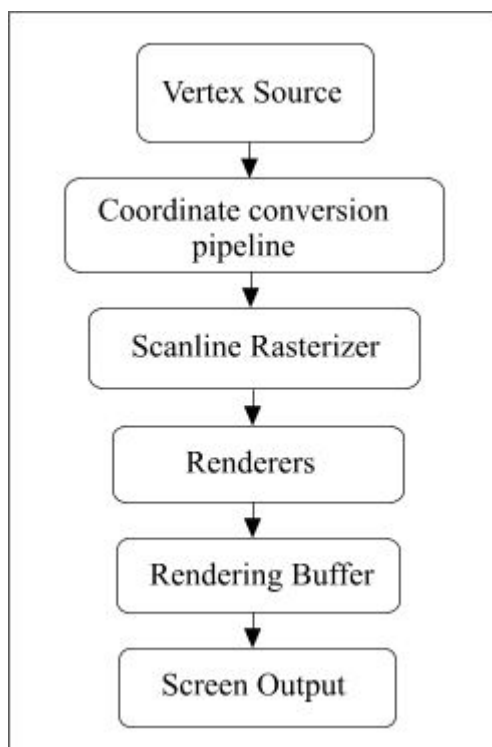
## 二、基本观念

### 1.库的设计

AGG 被设计成一系列松散的算法和类模板。因此所有的部分都能够很容易的组合起来。基于模板的设计使得可以替代库的一部分而不需要修改一行代码。AGG 也被设计的易扩展和灵活。可以在其中很容易的添加新的算法。

AGG 并没有指定任何它的形式，可以自由的使用它的任何一部分。然而，AGG 被当作一个在内存中渲染图象的工具。这是学习 AGG 的一个不错的着手点，可这样并不十分准确。在 AGG 提供的教程中有如何开始学习 AGG 更好的例子。

下面是一个典型的 AGG 渲染途径。



注意在 Vertex Source 和 Screen Output 之间的任一步骤都不是强制性的。例如，你能使用基于 Windows API 光栅化器，这样你就不需要 AGG 的光栅化和渲染了。如果只想绘制线，可以使用 AGG 的 outline 光栅化器，这样可以更快。

对上图的一些说明：

#### λ Vertex Source

Vertex Source 是一些对象。如多边形, polyline 等一系列连续的 2D 顶点。可以通过 MoveTo, LineTo 产生。

#### λ Coordinate conversion pipeline

Coordinate conversion pipeline 由一组坐标转换器组成。它通常作用于用浮点数描绘的矢量数据 (X, Y)。可能会包括一些几何变换, 轮廓生成等。

#### λ Scanline Rasterizer

Scanline Rasterizer 把矢量数据转换成许多水平的扫描线。这些扫描线通常 (不是必须) 带有反锯齿的信息。

#### λ Renderers

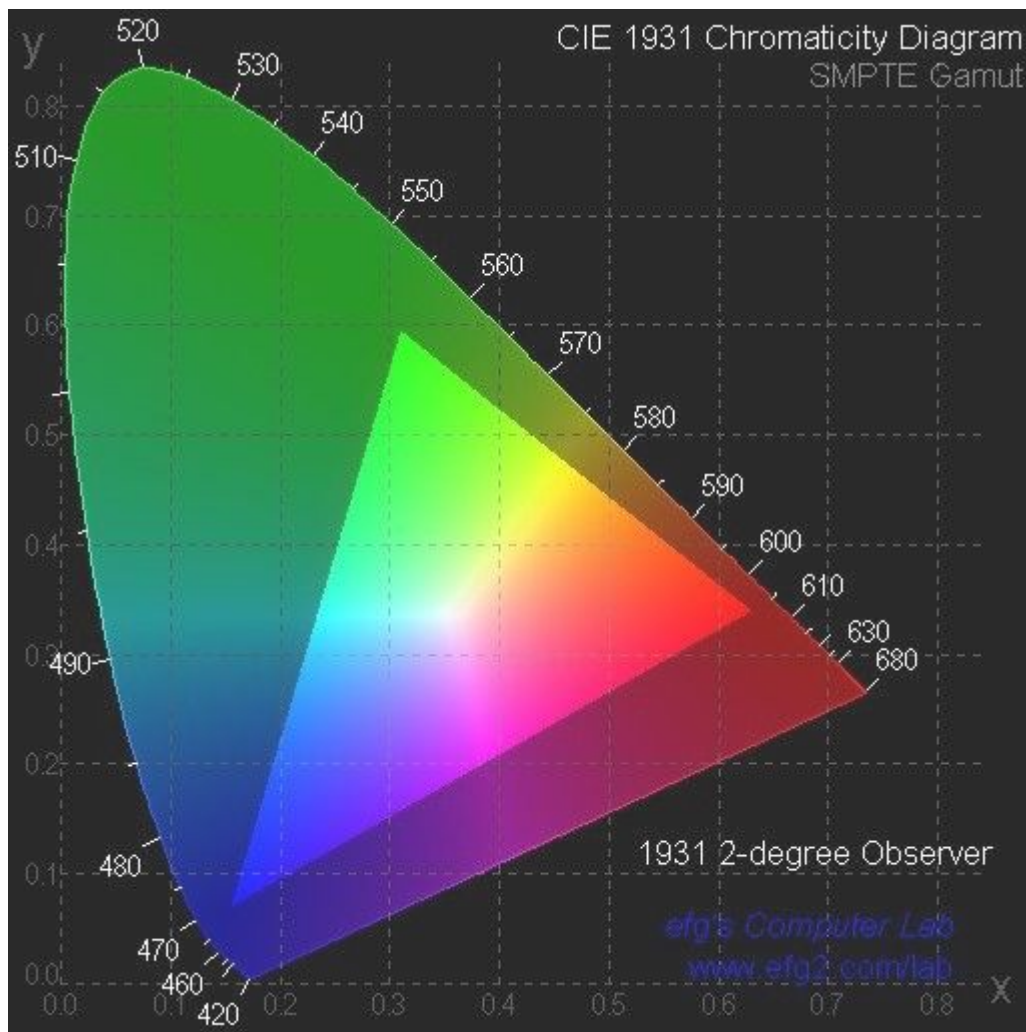
Renderers 对扫描线进行渲染。把渲染的结果写入 rendering buffer。

#### λ Rendering Buffer

Rendering Buffer 是内存中的一个在后面用于显示的缓冲区。通常 (不是必须) 它包含了适应于显示系统的像素格式。如, 24 位的 B-G-R, 32 位的 B-G-R-A 或 15 位的 R-G-B-555 等。

## 2.颜色, 颜色空间和像素格式

在 AGG 中颜色直到要把数据放入 rendering buffer 中时才在渲染过程中出现。一般来说, 没有通常 “color” 意图的结构体或类。AGG 总是对具体的颜色空间进行操作。这儿包括的颜色空间有很多种, 如 RGB, HSV, CMYK 等。这些颜色空间通常有特定的限制。比如, RGB 空间就只是人眼可见颜色的一个很小的子集, 下图中的三角形区域。



### CIE 色品图和 RGB 色域

换句话说，在真实世界中有很多颜色是不能用 RGB, CMYK, HSV 等颜色空间来描述的。除了自然中存在的，任何一个其他颜色空间都是受限制的。因此，为了避免在将来可能出现的限制，没有引入一个如“color”般的对象。

### 3.坐标单位

AGG 主要使用输出设备的坐标。在屏幕上就是像素。但是不同于其他一些库和 API, AGG 支持亚像素精度。这意味着需要用浮点数描述坐标，因为小数部分的值也将会产生影响。为了不限使用者的自由，AGG 中没有嵌入从世界坐标转换到屏幕坐标的转换机制。当不同的应用软件中需要不同的近似方式时，这就显得很重要了。AGG 只是提供了从 view

port 到设备的变换器。可以添加自己的简单的类来以毫米、英寸或是其他物理单位为单位进行转换。

## 评论

### 评论 1

曾计划用 AGG 做文档编辑软件的图形内核，读过一段时间它的代码：设计精巧，结构清晰，颇有学术风格，对模板的理念和应用不亚于 STL。

然而却感觉这么好的东西，地位却非常尴尬：

同样跨平台的有 QT，Cairo；高阶渲染功能未全部实现，已实现的有 bug（可能因为未经过大规模应用验证）；可扩展性还不错，前提是读懂大量带模板的代码。

最致命的是纯软件渲染，不能硬件加速；无三维扩展性，不能用在有三维需要的场合，在 vista 图形引擎都已经三维化的年代，三维恐怕是未来图形系统的入门级要求了，大型图形应用软件恐怕不敢用它做底层。

所以比较适合做轻量级的图形效果控件，像素操作是 GDI 的 10 倍。

从功能上，Cairo 倒是比较平衡的选择。但 C 代码读起挺郁闷不说，扩展也难。

## 评论 2

C++的接口对 C++程序员是比较有吸引力的，作为 GDI 的替代方案，它在速度和质量上确实好很多。我主要用来做 GIS 的图形显示，感觉够用了。就是其中的组件比较零散，学起来比较费劲。

## 用 AGG 实现高质量图形输出

### 使用前 AGG 的准备工作

下载 AGG 库，它的家在 <http://www.antigrain.com>，目前最高版本是 AGG2.5

解压，后面以[AGG]表示 AGG 的解压目录.

把[AGG]/include 加入到 include 搜索目录中

把[AGG]/src 里所有 cpp 加入到项目中（或者用 makefile 一起编译）

另外，AGG 还有一些其它组件，用到时也要把它们(都是些.h 和.cpp 文件)加入项目：

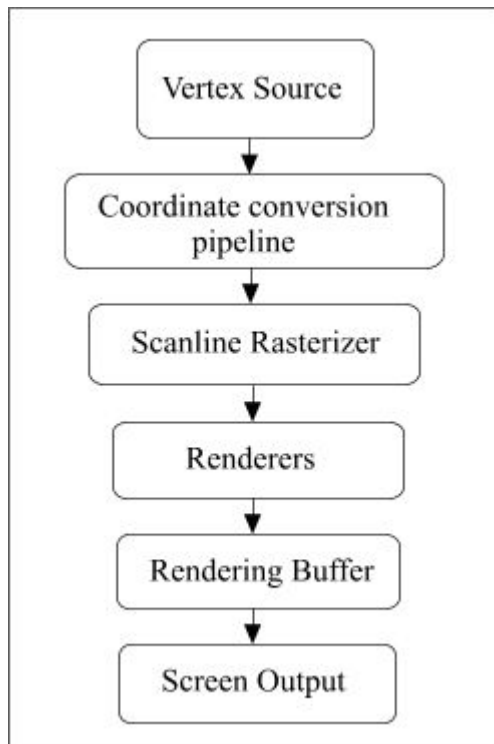
如果要用 AGG 的控件和窗体，要加入[AGG]/src/ctrl/\*.cpp 和

[AGG]/src/platform/<OS>/\*.cpp，头文件在[AGG]/include/ctrl 和[AGG]/include/platform 里

如果要用到 TrueType 字体显示，要加入[AGG]/font\_win32\_tt 目录下的源码和头文件。利用 freetype 库，则是[AGG]/font\_freetype 目录。

如果要用到 Generic Polygon Clipper 库（一个区域剪裁计算库），加入[AGG]/gpc 目录下的源码和头文件。

AGG 图形显示原理见下图：



其中：

**Vertex Source** 顶点源，里面存放了一堆 2D 顶点以及对应的命令，如"MoveTo"、"LineTo"等。

**Coordinate conversion pipeline** 坐标转换管道，它可以变换 Vertex Source 中的顶点，比如矩阵变换，轮廓提取，转换为虚线等。

**Scanline Rasterizer** 把顶点数据（矢量数据）转换成一组水平扫描线，扫描线由一组线段(Span)组成，线段(Span)包含了起始位置、长度和覆盖率（可以理解为透明度）信息。AGG 的抗锯齿（Anti-Aliasing）功能也是在这时引入的。

**Renderers** 渲染器，渲染扫描线(Scanline)中的线段(Span)，最简单的就是为 Span 提供单一颜色，复杂的有多种颜色(如渐变)、使用图像数据、Pattern 等。

**Rendering Buffer** 用于存放像素点阵数据的内存块，这里是最终形成的图像数据。

要理解 AGG 的工作原理，先看一段代码：

```
#include "agg_basics.h"

#include "agg_rendering_buffer.h"

#include "agg_rasterizer_scanline_aa.h"

#include "agg_scanline_u.h"

#include "agg_renderer_scanline.h"

#include "agg_pixfmt_rgb.h"

#include "platform/agg_platform_support.h"

#include "agg_ellipse.h"

#include "agg_conv_contour.h"

#include "agg_conv_stroke.h"

class the_application : public agg::platform_support
{
public:
    the_application(agg::pix_format_e format, bool flip_y) :
        agg::platform_support(format, flip_y)
    {
    }

    virtual void on_draw()
    {
        //Rendering Buffer

        agg::rendering_buffer &rbuf = rbuf_window();
```



```
agg::pixfmt_bgr24 pixf(rbuf);
```

```
// Renderers
```

```
typedef agg::renderer_base<agg::pixfmt_bgr24> renderer_base_type;
```

```
renderer_base_type renb(pixf);
```

```
typedef agg::renderer_scanline_aa_solid<renderer_base_type> renderer_scanline_type;
```

```
renderer_scanline_type rens(renb);
```

```
// Vertex Source
```

```
agg::ellipse ell(100,100,50,50);
```

```
// Coordinate conversion pipeline
```

```
typedef agg::conv_contour<agg::ellipse> ell_cc_type;
```

```
ell_cc_type ccell(ell);
```

```
typedef agg::conv_stroke<ell_cc_type> ell_cc_cs_type;
```

```
ell_cc_cs_type cscell(ccell);
```

```
// Scanline Rasterizer
```

```
agg::rasterizer_scanline_aa<> ras;
```

```
agg::scanline_u8 sl;
```

```
// Draw
```

```
renb.clear(agg::rgba8(255,255,255));
```

```

    for(int i=0; i<5; i++)

    {

        ccell.width(i*20);

        ras.add_path(csccell);

        rens1.color( agg::rgba8(0,0,i*50));

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

    }

}

};

int agg_main(int argc, char* argv[])

{

    the_application app(agg::pix_format_bgr24, false);

    app.caption("AGG Example. Anti-Aliasing Demo");

    if(app.init(600, 400, agg::window_resize))

    {

        return app.run();

    }

    return -1;

}

```

编译这段代码的方法是（以 VC 为例）：

新建空白 GUI 项目（就是有 WinMain 的项目）

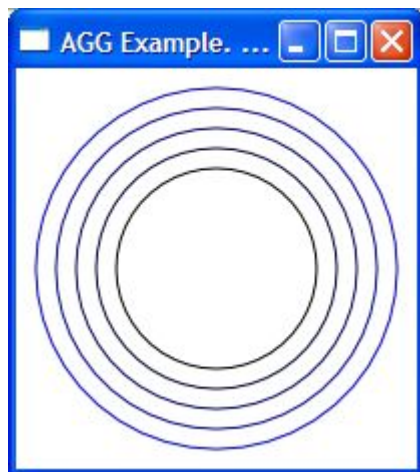
把[AGG]/src 里所有\*.cpp 加入到项目中

把[AGG]/src/platform/Win32/\*.cpp 加入到项目中

Ctrl+C/Ctrl+V 上面的代码

编译!

显示效果:



我们先不管 `agg_main` 及 `agg::platform_support` 的问题，实际上 `agg::platform_support` 只是 AGG 给我们方便显示 AGG 图形用的,真正应用时几乎不会用到(后面会讲到怎样把 AGG 图形画到 HDC 上)。

现在我们只需要知道这个框架可以生成一个窗体，当窗体重画时会调用 `virtual void on_draw()` 就行了。

现在直接从 `on_draw()` 开始看

通过 `rbuf_window()` 方法得到一个 `agg::rendering_buffer`，它就是“**Rendering Buffer**”，是一块用于存放图像的内存块。通过 `pixfmt_bgr24` 包装，我们就可以以像素为单位存取图像。

`agg::renderer_base` 和 `agg::renderer_scanline_aa_solid` 都属于“渲染器 **Renderer**”。

`renderer_base` 为底层渲染器，它支撑起所有的高层渲染器。这里的

`renderer_scanline_aa_solid` 就是一个高层渲染器。

`agg::ellipse` 是“顶点源 **Vertex Source**”，这个顶点源呈现的是一个圆形。

`agg::conv_contour` 和 `agg::conv_stroke` 作为“坐标转换管道 **Coordinate conversion pipeline**”，

`conv_contour` 扩展轮廓线，`conv_stroke` 只显示轮廓线（如果没有 `conv_stroke` 就会显示实心圆，可以去掉试试）。

`agg::rasterizer_scanline_aa` 就是“**Scanline Rasterizer**”啦。

`agg::render_scanlines` 函数执行这个 AGG 工作流程。

## 顶点源(Vertex Source)

顶点源是一种可以产生多边形所需要的“带命令的顶点”的对象。比如三角形顶点源，就应该会产生一个带“MoveTo”命令的点，另外二个带“LineTo”命令的点和最终闭合的“ClosePoly”命令。

## 头文件

```
#include <agg_path_storage.h> //path_storage

#include <agg_ellipse.h> // ellipse

#include <agg_arc.h> // arc

#include <agg_arrowhead.h> // arrowhead

#include <agg_curves.h> // curve3, curve4

#include <agg_gsv_text.h> // gsv_text, gsv_text_outline
```

```
#include <agg_rounded_rect.h> // rounded_rect
```

...

## 类型

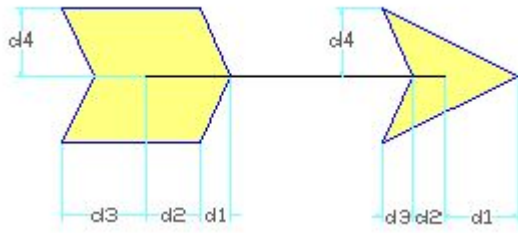
自定义类	所有实现了 void rewind(unsigned path_id);和 unsigned vertex(double* x, double* y);的类。
ellipse	圆，输入为中心点坐标和 XY 轴半径，本文所用的 <a href="#">例子</a> 就 使用了这个顶点源
arc	弧线，输入为中心点坐标和 XY 轴半径，以及起始和终止角(rad)，顺时针/逆时针方向
curve3	贝塞尔曲线，输入为起点坐标、第一控制点坐标、终点坐标
curve4	贝塞尔曲线，输入为起点坐标、第一控制点坐标、第二控制点坐标、终点坐标
gsv_text	使用 AGG 自带字模的文字输出（只支持 ASCII 码），使用 start_point 方法指定文字位置，text 方法指定 文字，flip 指定是否上下倒转，size 指定文字大小，适合与 conv_stroke 或 gsv_text_outline 配合。
gsv_text_outline	可变换文字，输入为 gsv_text 和变换矩阵(默认为 trans_affine，后文会提到)。width 方法设置文 字宽度
rounded_rect	圆角方形，输入为左上角右下角坐标和圆角半径

路径存储器, 可以用 join\_path 方法加入多个顶点源。而且 path\_storage  
**path\_storage** 本身支持 move\_to, line\_to, curve 和 arc\_to 等画线功能

**arrowhead** 箭头, 它是作为标记点来用的

其中的 arrowhead 颇为特殊, 它一般作为线段的标记点, 具体用法是这样的:

```
arrowhead ah;  
  
ah.head(d1,d2,d3,d4); //定义箭头  
  
ah.tail(d1,d2,d3,d4); //定义箭尾  
  
VertexSource VS; //其它顶点源  
  
// 使用顶点转换器, 并指定 Markers 类型为 vcgen_markers_term  
  
// 顶点转换器可以是 conv_dash、conv_stroke 或 conv_marker_adaptor, 见后文《坐标转换管道》  
  
// vcgen_markers_term:以端点作为标记点  
  
conv_stroke<VertexSource, vcgen_markers_term> csVS(VS);  
  
...draw_term  
  
// 用 conv_marker 指定 ah 作为线段 marker 点的标记  
  
conv_marker<vcgen_markers_term, arrowhead> arrow(csVS.markers(), ah);  
  
ras.add_path(csVS);  
  
ras.add_path(arrow); //marker 要紧随其后加入  
  
...render  
  
ah.head()和 ah.tail()方法中的 d1,d2,d3,d4 参数的意义见下图
```



例，画一条简单的箭头直线(基于[此处代码](#))

在 `on_draw()` 方法最后加上下列代码：

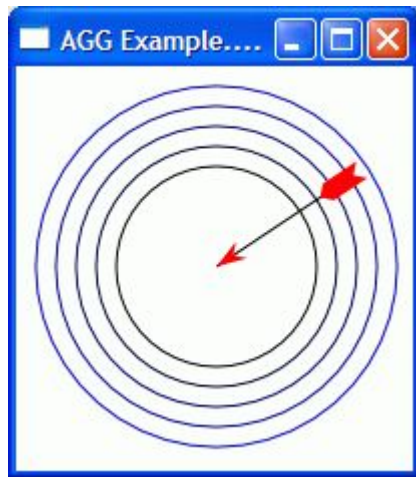
```

1.  agg::arrowhead ah;
2.  ah.head(0,10,5,5);
3.  ah.tail(10,10,5,5);
4.  // 用 path_storage 生成一条直线
5.  agg::path_storage ps;
6.  ps.move_to(160,60);
7.  ps.line_to(100,100);
8.  // 转换
9.  agg::conv_stroke<agg::path_storage, agg::vcgen_markers_term> csps(ps);
10. agg::conv_marker<agg::vcgen_markers_term, agg::arrowhead>
11.     arrow(csps.markers(), ah);
12. // 画线
13. ras.add_path(csps);
14. agg::render_scanlines_aa_solid(ras,sl,renb,agg::rgba8(0,0,0));
15. // 画箭头
16. ras.add_path(arrow);
17. agg::render_scanlines_aa_solid(ras,sl,renb,agg::rgba8(255,0,0));

```

得到的图形是:





注意要加头文件:

```
#include "agg_conv_marker.h"

#include "agg_arrowhead.h"

#include "agg_path_storage.h"

#include "agg_vcgen_markers_term.h"
```

试验代码，自定义一个顶点源(基于[此处代码](#))

为了对顶点源有更深入的了解，我们要自己实现一个顶点源，这个顶点源只是很简单的一个三角形。

前面说过，只要实现了 `void rewind(unsigned path_id);` 和 `unsigned vertex(double* x, double* y);` 方法就可以作为顶点源。

**rewind** 方法指示顶点源回到第一个顶点；**vertex** 方法取出当前顶点然后把当前顶点下移，返回值是当前顶点所带的命令。所谓的命令是一个 `enum path_commands_e` 类型，定义如下：

```
1.  enum path_commands_e
2.  {
3.      path_cmd_stop    = 0,    //----path_cmd_stop
4.      path_cmd_move_to  = 1,    //----path_cmd_move_to
5.      path_cmd_line_to  = 2,    //----path_cmd_line_to
6.      path_cmd_curve3   = 3,    //----path_cmd_curve3
7.      path_cmd_curve4   = 4,    //----path_cmd_curve4
8.      path_cmd_curveN   = 5,    //----path_cmd_curveN
9.      path_cmd_catrom    = 6,    //----path_cmd_catrom
10.     path_cmd_ubspline = 7,    //----path_cmd_ubspline
```

```

11. path_cmd_end_poly = 0x0F, //----path_cmd_end_poly
12. path_cmd_mask     = 0x0F //----path_cmd_mask
13. };

```

`path_commands_e` 还能和 `path_flags_e` 组合:

```

1. enum path_flags_e
2. {
3.     path_flags_none = 0, //----path_flags_none
4.     path_flags_ccw  = 0x10, //----path_flags_ccw
5.     path_flags_cw   = 0x20, //----path_flags_cw
6.     path_flags_close = 0x40, //----path_flags_close
7.     path_flags_mask = 0xF0 //----path_flags_mask
8. };

```

`vertex()`返回的命令中含有 `path_cmd_stop` 时表示结束。

```

1. // 等边三角形
2. class triangle{
3. public:
4.     triangle(double cx, double cy, double r)//中心点, r 为中心点到边的长度
5.     {
6.         // 直接准备好三个点
7.         m_step = 0;
8.         m_pt[0].x = cx; m_pt[0].y = cy-r;
9.         m_pt[1].x = cx+r*0.866; m_pt[1].y = cy+r*0.5;
10.        m_pt[2].x = cx-r*0.866; m_pt[2].y = cy+r*0.5;
11.        //AGG 把方向作为区分多边形内部和外部的依据, 可以试试 m_pt[1]和 m_pt[2]对调
12.    }
13.    void rewind(unsigned)
14.    {
15.        m_step = 0;
16.    }
17.    unsigned vertex(double* x, double* y)
18.    {
19.        switch(m_step++)
20.        {
21.        case 0:
22.            //第一步, move_to
23.            *x = m_pt[0].x;
24.            *y = m_pt[0].y;
25.            return agg::path_cmd_move_to;
26.        case 1:
27.        case 2:

```

```

28.     //第二、三步, line_to
29.     *x = m_pt[m_step-1].x;
30.     *y = m_pt[m_step-1].y;
31.     return agg::path_cmd_line_to;
32. case 3:
33.     // 第四步, 闭合多边形
34.     return agg::path_cmd_end_poly|agg::path_flags_close;
35. default:
36.     // 第五步, 结束
37.     return agg::path_cmd_stop;
38. }
39. }
40. private:
41.     agg::point_d m_pt[3];
42.     unsigned m_step;
43. };

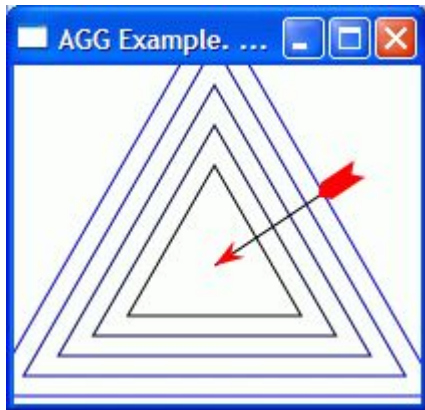
```

在 on\_draw()方法里把

`agg::ellipse ell(100,100,50,50);` 改成 `triangle ell(100,100,50);`

`typedef agg::conv_contour<agg::ellipse> ell_cc_type;`改成 `typedef agg::conv_contour<triangle>`  
`ell_cc_type;`

得到的图形是:



除了文字输出功能（`gsv_text` 只能输出 ASCII 文字），上面这些顶点源提供的图形丰富程度已经超过了系统 API。文字输出功能 将以单独的篇幅讲述

## Coordinate conversion pipeline 坐标转换管道

坐标转换管道用于改变顶点源产生的顶点，包括坐标、命令、产生新顶点等。如对顶点进行矩阵变换、插入顶点形成虚线之类的功能。

变换矩阵(trans\_affine)

在认识转换管道之前，先了解一下 AGG 的变换矩阵。通过顶点坐标与矩阵的运行，我们可以得到新的坐标。关于图像的矩阵运算，MSDN 里 有一篇[关于 GDI+矩阵运算的文章](#)，很值得一看

头文件

```
#include <agg_trans_affine.h>
```

类型

```
trans_affine
```

成员变量

这六个变量组成一个 2\*3 的矩阵，与坐标计算后得到一个新的坐标。

比如对点(x,y)进行变换，则新的点(x',y') 为:

```
double sx, shy, shx,
```

```
sy, tx, ty;
```

$$x' = x*sx + y*shx + tx;$$
$$y' = x*shy + y*sy + ty;$$

成员方法

```
void transform(double* x, double* y) const;
```

用上面的公式转换 x,y 坐标

```
const trans_affine& scale(double s);
```

缩放

```
const trans_affine& scale(double x, double y);
```

```
const trans_affine& rotate(double a);
```

旋转，弧度单位（pi/180）

```
const trans_affine& translate(double x, double y)
```

平移

```
trans_affine operator * (const trans_affine& m);
```

矩阵乘法

```
const trans_affine& invert();
```

取反矩阵

坐标转换管道中有个叫 **conv\_transform** 的转换器，它能利用矩阵对源顶点进行变换，我们先在这里玩玩吧^^

实验代码(基于[此处](#)代码)

加入头文件 `#include "agg_conv_transform.h"`

把 `on_draw()` 方法的里从 “`// Vertex Source`” 到 “`// Scanline Rasterizer`” 之间的代码改写成：

```
// Vertex Source
```

```
agg::ellipse ell(0,0,50,50); //圆心在中间
```

```
// Coordinate conversion pipeline
```

```
agg::trans_affine mtz;
```

```
mtz.scale(0.5,1); // x 轴缩小到原来的一半
```

```
mtz.rotate(agg::deg2rad(30)); // 旋转 30 度
```

```
mtz.translate(100,100); // 平移 100,100
```

```
typedef agg::conv_transform<agg::ellipse> ell_ct_type;
```

```
ell_ct_type ctell(ell, mtx); // 矩阵变换
```

```
typedef agg::conv_contour<ell_ct_type> ell_cc_type;
```

```
ell_cc_type ccell(ctell); // 轮廓变换
```

```
typedef agg::conv_stroke<ell_cc_type> ell_cc_cs_type;
```

```
ell_cc_cs_type cscell(ccell); // 转换成多义线
```

得到的图形是:



注：**trans\_affine** 不仅仅用于源顶点的变换，在 AGG 库中有不少地方都能看到它。比如后面会讲到的线段(span)生成器，通过变换矩阵，就能够 自由变换填充于多边形之内的图案。

坐标转换管道

头文件

```
#include <agg_conv_stroke.h> // conv_stroke
```

```
#include <agg_conv_dash.h> // conv_dash
```

```
#include <agg_conv_marker.h> // conv_marker
```

```
#include <agg_conv_curve.h> // conv_curve
```

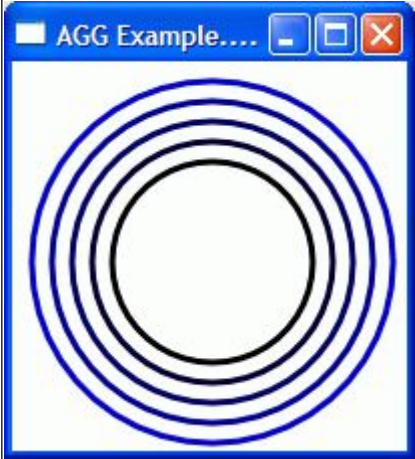
```
#include <agg_conv_contour.h> // conv_contour
```

```
#include <agg_conv_smooth_poly1.h> // conv_smooth_poly1.h
```

```
#include <agg_conv_bspline.h> // conv_bspline
```

```
#include <agg_conv_transform.h> // conv_transform
```

类型(演示程序基于[此处代码](#))

<pre>template&lt;class VertexSource, class Markers = null_markers&gt; struct conv_stroke;</pre>	<p>变成连续线 构造参数为 VertexSource width 属性决定线宽。</p>	<p>在<a href="#">例程</a>的 ell_cc_cs_type cscell(ccell); 后面加上 cscell.width(3);线宽就会 变成 3。</p> 
<pre>template&lt;class VertexSource, class Markers = null_markers&gt; struct conv_dash;</pre>	<p>虚线 构造参数为 VertexSource 用 add_dash 设置虚线</p>	<pre>// Coordinate conversion pipeline typedef agg::conv_contour&lt;agg::ellipse&gt;</pre>



长度和间隔

与 conv\_stroke 套用

```
ell_cc_type;
```

```
ell_cc_type ccell(ell);
```

```
typedef agg::conv_dash<ell_cc_type>
```

```
ell_cd_type;
```

```
ell_cd_type cdccell(ccell);
```

```
cdccell.add_dash(5,5);
```

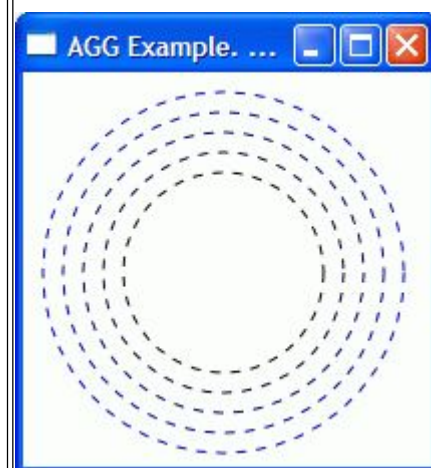
```
typedef
```

```
agg::conv_stroke<ell_cd_type>
```

```
ell_cc_cs_type;
```

```
ell_cc_cs_type cscell(cdccell);
```

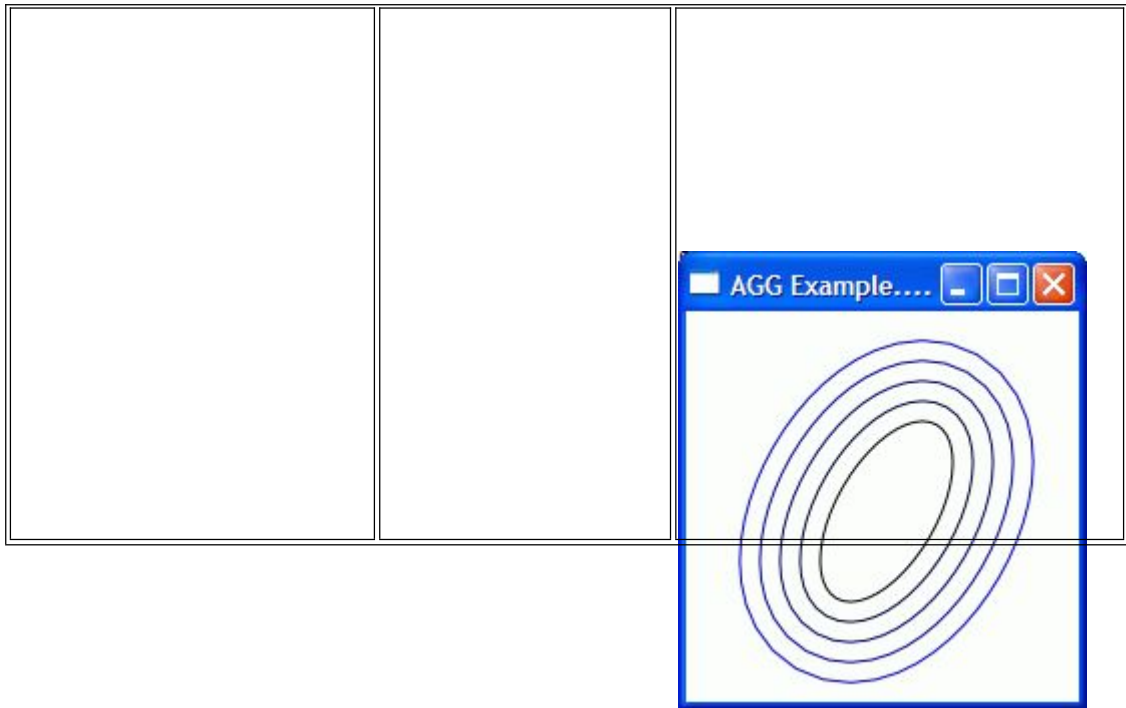
```
...
```



<pre>template&lt;class MarkerLocator, class MarkerShapes&gt; class conv_marker;</pre>	<p>建立标记</p>	<p>请参考 <a href="#">arrowhead 示例代码</a></p> 
<pre>template&lt;class VertexSource&gt; struct conv_contour;</pre>	<p>轮廓变换</p> <p>构造参数为</p> <p>VertexSource</p> <p>width 属性决定扩展或收缩轮廓。</p>	<p>见 <a href="#">例 程代码</a></p>
<pre>template&lt;class VertexSource&gt; struct conv_smooth_poly1_curve;</pre>	<p>圆滑过渡多边形各顶点（贝塞尔）</p> <p>构造参数为</p> <p>VertexSource</p> <p>smooth_value 属性决定圆滑度(默认为 1)</p>	<p>在 <a href="#">例 程</a> on_draw()方法最后加入下面代码</p> <pre>triangle t(100,100,50);//自定义顶点源 agg::conv_smooth_poly1_curve&lt;triangle&gt; cspct(t);</pre>

		<pre> ras.add_path(cspct);  agg::render_scanlines_aa_solid(  ras,sl,renb,agg::rgba8(255,0,0)); </pre>  <p>三角形就变得圆头圆脑啦^_^</p>
<pre> template&lt;class VertexSource&gt; struct conv_bspline; </pre>	<p>圆滑过渡多义线各顶点（贝塞尔）</p> <p>构造参数为</p> <p>VertexSource</p> <p>interpolation_step 属性</p> <p>决定步长。</p>	<p>在<a href="#">例程</a> on_draw()方法最后加入下面代码</p> <pre> triangle t(100,100,50);  agg::conv_bspline&lt;triangle&gt; cspct(t);  ras.add_path(cspct);  agg::render_scanlines_aa_solid(  ras,sl,renb,agg::rgba8(255,0,0)); </pre>

		 <p>三角形也能变得圆头圆脑</p>
<pre>template&lt;class VertexSource, class Curve3 = curve3, class Curve4 = curve4&gt; class conv_curve;</pre>	<p>可识别 VertexSource 中的曲线信息</p> <p>构造参数为 VertexSource。</p> <p>conv_smooth_poly1_curve</p> <p>就是基于它实现的。</p>	<p>例程里的顶点都没有曲线信息，算了，</p> <p>到后面讲到文字输出时会用到它的。</p>
<pre>template&lt;class VertexSource, class Transformer = trans_affine&gt; class conv_transform;</pre>	<p>矩阵变换</p> <p>用变换矩阵重新计算顶点位置</p> <p>构造参数为 VertexSource 和变换矩阵</p>	<p>见变换矩阵一节的例子</p>



### Scanline Rasterizer 光栅化

Scanline Rasterizer 能够把顶点数据转换成一组水平扫描线，扫描线由一组线段(Span)组成，线段(Span)包含了起始位置、长度和覆盖率（可以理解 为透明度）信息。AGG 的抗锯齿（Anti-Aliasing）功能也是在这时引入的。

#### 扫描线 Scanline

扫描线是一种保存 span 的容器，span 用于表示一小条（水平方向）细线。图像中同一行的 span 组成一个 Scanline。

#### 头文件

```
#include <agg_scanline_u.h> // scanline_u8,scanline32_u8
```

```
#include <agg_scanline_p.h> // scanline_p8,scanline32_p8
```

```
#include <agg_scanline_bin.h> // scanline_bin,scanline32_bin
```

类型

scanline_bin,scanline32_bin	不携带 AA 信息的 span 容器。scanline32_bin 中的 32 代表坐标位数，一般 16 位已经足够了，所以前一版本用得更多些（下同）
scanline_u8,scanline32_u8	unpacked 版的 span 容器，用每个 span 来保存各自的线段信息
scanline_p8,scanline32_p8	packed 版的 span 容器，相同属性的 span 会合并成一个

成员类型

struct span;	线段数据，其中的成员变量有：x 起始位置，len 长度，*covers 覆盖率
typename iterator,const_iterator;	span 迭代器
typename cover_type;	span 中 covers 类型(覆盖率)

成员方法

iterator <b>begin</b> ();  unsigned <b>num_spans</b> ();	用于遍历 span,begin()取得指向第一个 span 的迭代器  num_spans()取得容器中 span 的数目
--	---

void <b>reset</b> (int min_x, int max_x);	设置容器大小
void <b>add_span</b> (int x, unsigned len, unsigned cover)	加入一条线段
void <b>add_cell</b> (int x, unsigned cover)	加入一个点
void <b>add_cells</b> (int x, unsigned len, const cover_type* covers)	加入一组点
void <b>finalize</b> (int y)  int <b>y</b> ();	Scanline 容器对应的 Y 坐标

Rasterizer

怎么翻译呢？光栅化？光栅制造机？嗯~~算了，还是直接叫它 Rasterizer(雷死特拉倒)吧  
 \_-!!!

Rasterizer 就是把相当于矢量数据的一堆顶点和命令转换成一行行的扫描线的设备，它就像粉刷工人对照着图纸把彩漆刷到墙上一 样。可以说是 AGG 里最重要的类型之一，套用建翔兄的话就是：

立功了！立功了！不要给 GDI 任何的机会！伟大的 AGG 的 Rasterizer 类！他了继承开源社区的光荣传统！达芬奇、Linus、 唐寅，在这一刻灵魂附体！

Rasterizer 是关键对象！他代表了 AGG 伟大的设计理念！在这一刻！他不是一个人的战斗！他不是一个人！面对着全世界人民的目 光和期待，他深知责任的重大，0.001 秒种之后将会是什么样的图像？



头文件

```
#include <agg_rasterizer_scanline_aa.h>
```

类型

```
template<class Clip = rasterizer_sl_clip_int>
```

```
class rasterizer_scanline_aa;
```

成员方法

<pre>template&lt;class GammaF&gt;  void <b>gamma</b>(const GammaF&amp; gamma_function);</pre>	<p>设置 gamma 值。</p> <p>GammaF 为一种仿函数</p> <p>AGG 自带有 gamma_power、gamma_none、gamma_threshold、 gamma_linear、gamma_multiply</p>
<pre>bool <b>rewind_scanlines</b>();</pre>	<p>跳到第一个 scanline 位置，同时设置 sorted 为 true。</p> <p>这时再加入其它顶点会先清空现有顶点</p>
<pre>bool <b>navigate_scanline</b>(int y);</pre>	<p>跳到 y 行</p>
<pre>bool <b>sweep_scanline</b>(Scanline&amp;);</pre>	<p>把当前行画入 Scanline，当下移一行</p>
<pre>void <b>reset</b>();</pre>	<p>清空</p>
<pre>void <b>move_to</b>(int x, int y);</pre>	<p>简单的画线功能，单位为 1/poly_subpixel_scale</p>

void <b>line_to</b> (int x, int y);	(poly_subpixel_scale 一般为 256)
void <b>move_to_d</b> (double x, double y); void <b>line_to_d</b> (double x, double y);	简单的画线功能，单位为像素
void <b>add_path</b> (VertexSource& vs,unsigned path_id=0)	加入顶点

## Renderers 渲染器

渲染器负责表现扫描线 Scanline 中的每个线段（span）。在渲染器之前，AGG 图形中的线段是没有颜色值的，只是位置、长度和 覆盖率（透明度）。渲染器赋予线段色彩，最终成为一幅完整的图像。

渲染器被分成底中高三层。其中底层负责像素包装，由 **PixelFormat Renderer** 实现；中层是基础层，在 **PixelFormat Renderer** 的基础上提供更多方法，是所有高层渲染器依赖的基础，由 **Base Renderer** 实现；高层负责渲染 Scanline 中的线段，由 **Scanline Renderer** 等实现。

Scanline Renderer

头文件

```
#include <agg_renderer_scanline.h>
```

类型

```
template<class BaseRenderer> class renderer_scanline_aa_solid; //实色 AA 渲染
```

```
template<class BaseRenderer> class renderer_scanline_bin_solid; //实色原始渲染
```

```
template<class BaseRenderer, class SpanAllocator, class SpanGenerator>
```

```
class renderer_scanline_aa; // 自定义 AA 渲染
```

```
template<class BaseRenderer, class SpanAllocator, class SpanGenerator>
```

```
class renderer_scanline_bin; // 自定义原始渲染
```

以及自己写的实现了 void prepare() 和 template<class Scanline> void render(const Scanline& sl) 方法的类

另外，头文件 `agg_renderer_scanline.h` 中的 `render_scanlines` 函数很重要，它是 AGG 显示流程的实现。

```
void render_scanlines(Rasterizer& ras, Scanline& sl, Renderer& ren);
```

从 Rasterizer 生成逐行的 Scanline，然后交给 Scanline Renderer 渲染。

这里还要提一下 `render_scanlines_aa_solid`、`render_scanlines_aa`、

`render_scanlines_bin_solid`、`render_scanlines_bin` 这几个函数。它们的作用

和 `render_scanlines` 一样，只是跳过了 Scanline Renderer 环节，直接向 Base Renderer 渲染。

```
void render_scanlines_aa_solid(Rasterizer& ras, Scanline& sl,
```

```
BaseRenderer& ren, const ColorT& color)
```

```
template<class Rasterizer, class Scanline, class BaseRenderer,

class SpanAllocator, class SpanGenerator>

void render_scanlines_aa(Rasterizer& ras, Scanline& sl, BaseRenderer& ren,

SpanAllocator& alloc, SpanGenerator& span_gen);
```

实验代码(基于[此 处代码](#))

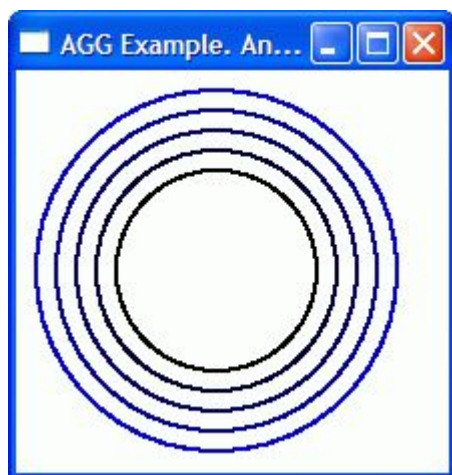
把 on\_draw()方法里原

```
typedef agg::renderer_scanline_aa_solid<renderer_base_type> renderer_scanline_type;
```

改成

```
typedef agg::renderer_scanline_bin_solid<renderer_base_type> renderer_scanline_type;
```

得到的图形是:



去掉 renderer\_scanline\_type 以及所有的 rensl 相关语句，把

```
agg::render_scanlines(ras,sl,rensl);
```

改成

```
agg::render_scanlines_aa_solid(ras,sl,renb,agg::rgba8(0,0,i*50));
```

同样可以得到我们想要的图形

Basic Renderers

头文件

```
#include <agg_renderer_base.h>
```

```
#include <agg_renderer_mclip.h>
```

类型

```
template<class PixelFormat> class renderer_base;
```

```
template<class PixelFormat> class renderer_mclip;
```

构造函数

```
renderer_base(pixfmt_type& ren); 参数 ren 指定底层的 PixelFormat Renderer
```

成员方法

<code>pixfmt_type&amp; <b>ren</b>();</code>	返回底层的 PixelFormat Renderer
<code>unsigned <b>width</b>() const;</code> <code>unsigned <b>height</b>() const;</code>	宽高

void <b>reset_clipping</b> (bool visibility);	设置是否可见  clipping  box=visibility?(0,0,width-1,height-1):(1,1,0,0)
bool <b>clip_box</b> (int x1, int y1, int x2, int y2);	设置 clipping box，renderer_base 专有
void <b>add_clip_box</b> (int x1, int y1, int x2, int y2);	添加 clipping box，renderer_mclip 专有
bool <b>inbox</b> (int x, int y) const;	x,y 点是否在 clipping box 内，renderer_base 专有
void <b>first_clip_box</b> ();  bool <b>next_clip_box</b> ();	切换 clipping box，renderer_mclip 专用
const rect& <b>clip_box</b> () const;  int <b>xmin</b> ()     const;  int <b>ymin</b> ()     const;  int <b>xmax</b> ()     const;  int <b>ymax</b> ()     const;  const rect& <b>bounding_clip_box</b> () const;  int <b>bounding_xmin</b> ()     const;  int <b>bounding_ymin</b> ()     const;  int <b>bounding_xmax</b> ()     const;	返回 clipping box 大小

<code>int     <b>bounding_ymax</b>()     const;</code>	
<code>void <b>clear</b>(const color_type&amp; c);</code>	以颜色 c 填充所有区域
<code>void <b>copy_pixel</b>(int x, int y, const color_type&amp; c);  void <b>blend_pixel</b>(int x, int y, const color_type&amp; c, cover_type cover);  color_type <b>pixel</b>(int x, int y) const;  void <b>copy_h</b>(<del>void</del><b>line</b>(int x1, int y, int x2, const color_type&amp; c);  void <b>blend_h</b>(<del>void</del><b>line</b>(int x1, int y, int x2,                   const color_type&amp; c, cover_type cover);  void <b>blend_solid_h</b>(<del>void</del><b>span</b>(int x, int y, int len,                   const color_type&amp; c, const cover_type* covers);  void <b>blend_color_h</b>(<del>void</del><b>span</b>(<i>_no_slip</i>)int x, int y, int len,                   const color_type* colors, const cover_type* covers);</code>	见后文的 PixelFormat Renderer

<pre>void <b>copy_from</b>(const rendering_buffer&amp; from,     const rect* rc=0,     int x_to=0,     int y_to=0);</pre>	<p>从 from 复制一个矩形区域过来，rc 指定源区域，x_to,y_to 指定目标位置</p>
---	--

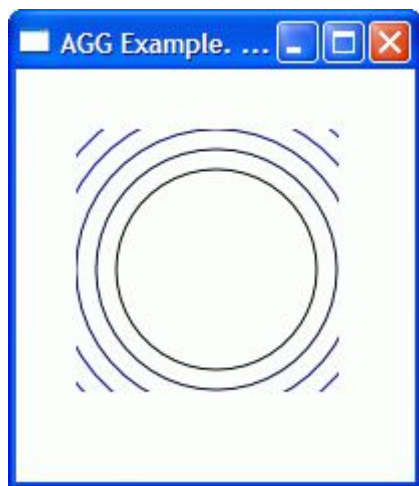
实验代码(基于[此 处代码](#))

在 on\_draw()方法的 renb.clear(agg::rgba8(255,255,255));语句后面加上：

```
renb.clear(agg::rgba8(255,255,255));
```

```
renb.clip_box(30,30,160,160); // 设置可写区域
```

得到的图形是:



PixelFormat Renderer

PixelFormat Renderer 的作用是以指定的颜色空间来包装原始的 **Rendering Buffer**(见后文)，AGG 把它归类于底层 Renderer。

Rendering Buffer 是以字节为单位的，而 PixelFormat Renderer 则是以像素为单位的。



头文件

```
#include "agg_pixfmt_rgb.h
```

```
#include "agg_pixfmt_gray.h"
```

类型

```
pixfmt_gray8
pixfmt_rgb24
pixfmt_bgr24
pixfmt_rgba32
pixfmt_bgr24_gamma
...
```

构造函数

```
pixfmt_base(rbuf_type& rb);
```

rb 参数为 Rendering Buffer 类型

类型定义

	<p>像素类型</p> <p>需要了解的是在 AGG 中像素也是一个功能完善的类,常用的有 rgba、rgba8、gray8。</p>
<pre>typedef color_type;</pre>	<p>rgba 里每个颜色分量用 double 表示, 范围从 0~1。其它像素类后面的数字代表每个颜色分量占用的位数。大部分像素类都可以从 rgba 构造。</p> <p>同时, 像素类还有 gradient 等牛 X 的颜色计算方法。</p>

typedef <b>value_type</b> ;	单个颜色分量的类型
typedef <b>order_type</b> ;	<p>颜色排序方式，我们可以通过里面的枚举值 R G B A 得到各颜色分量所在位置，常用的有 order_rgb, order_bgr, order_rgba。</p> <p>这是 order_rgb 的定义： struct order_rgb { enum rgb_e { R=0, G=1, B=2, rgb_tag }; };</p>

### 成员方法

unsigned <b>width</b> ()  unsigned <b>height</b> ()	宽高
color_type <b>pixel</b> (int x, int y);  void <b>copy_pixel</b> (int x, int y, const color_type& c);	取得、设置指定点的颜色
void <b>blend_pixel</b> (int x, int y, const color_type& c, int8u cover);	设置指定点颜色，与原颜色有混合效果，强度由 cover 指定
void <b>copy_hline</b> (int x, int y, unsigned len, const color_type& c);  void <b>copy_vline</b> (int x, int y, unsigned len, const color_type& c);	从 x,y 开始画一条长度为 len 的线，颜色为 c，同样有 blend_版本
void <b>blend_solid_h</b> ( <del>r</del> ) <b>span</b> (int x, int y, unsigned len,	类似 hline 和 vline 版本，color 版指定一组

<pre> const color_type&amp; c, const int8u* covers);  void blend_color_h(span(int x, int y, unsigned len,  const color_type* colors, const int8u* covers); </pre>	<p>颜色，依次着色。covers 指定覆盖率</p>
---	-----------------------------

实验代码(基于[此处代码](#))

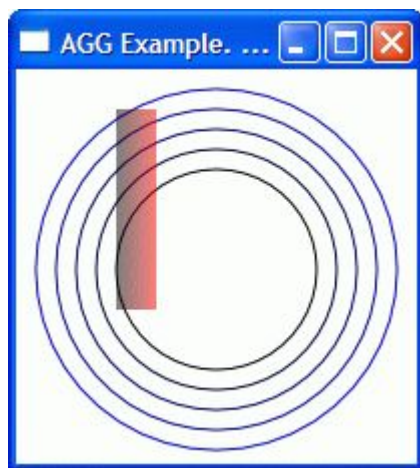
在 on\_draw()方法的最后加上：

//从 50,20 开始，画 20 条长度为 100 的竖线，颜色从黑渐变到红，覆盖率为 128(半透明)

```
for(int i=0; i<20; i++)
```

```
    pixf.blend_vline(50+i,20,100,agg::rgba(i/20.0,0,0),128);
```

得到的图形是:



Rendering Buffer 渲染缓存

Rendering Buffer 是一个内存块，用于保存图像数据。这是 AGG 与显示器之间的桥梁，我们要显示 AGG 图形实际上就是识别这个内存块并使用系统的 API 显示出来 而已（实际上几乎不需要做转换工作，因为无论是 Windows 还是 Linux，API 所用的图像存储格式与 Rendering Buffer 都是兼容的）。

头文件：

```
#include "agg_rendering_buffer.h"
```

类型：

```
rendering_buffer
```

构造函数：

```
rendering_buffer(int8u* buf, unsigned width, unsigned height, int stride);
```

参数分别表示内存块指针，宽、高、每行的步幅（当步幅<0 时，表示上下颠倒）

成员方法：

<code>void <b>attach</b>(int8u* buf, unsigned width, unsigned height, int stride);</code>	参数与构造函数相同
<code>int8u* <b>buf</b>();</code>	返回内存块指针
<code>unsigned <b>width</b>() const;</code> <code>unsigned <b>height</b>() const;</code>	返回宽、高、每行步幅

<pre>int    <b>stride</b>() const;  unsigned <b>stride_abs</b>() const;</pre>	
<pre>int8u* <b>row_ptr</b>(int y)</pre>	返回指向第 y 行起点的指针
<pre>void <b>clear</b>(int8u value)</pre>	以 value 值填充整个内存块
<pre>template&lt;class RenBuf&gt; void <b>copy_from</b>(const RenBuf&amp; src)</pre>	从另一 rendering_buffer 中复制数据

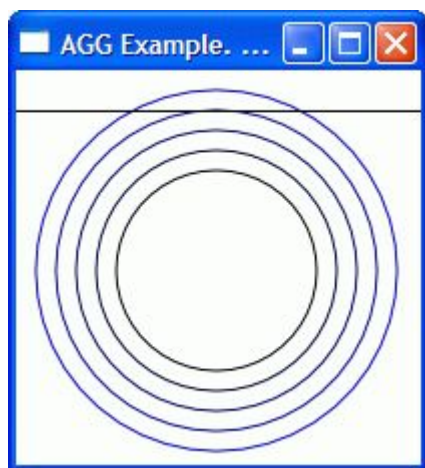
实验代码(基于[此处代码](#))

在 on\_draw()方法的最后加上：

```
agg::int8u* p = rbuf.row_ptr(20);//得到第 20 行指针
```

```
memset(p,0,rbuf.stride_abs());//整行以 0 填充
```

得到的图形是：



AGG 与 GDI 显示

Rendering Buffer 的图像存储方式和 Windows 的 BMP 是一样的，所以让 AGG 处理 BMP 是很简单的事情，下面的代码演示了怎样在 HDC 上显示 AGG

```
#include <agg_rendering_buffer.h>

#include <agg_pixfmt_rgba.h>

#include <agg_renderer_base.h>

#include <agg_rasterizer_scanline_aa.h>

#include <agg_scanline_p.h>

...

// 首先让系统生成一个 32 位的 bmp 缓存

BITMAPINFO bmp_info;

bmp_info.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);

bmp_info.bmiHeader.biWidth = width;

bmp_info.bmiHeader.biHeight = height;

bmp_info.bmiHeader.biPlanes = 1;

bmp_info.bmiHeader.biBitCount = 32;

bmp_info.bmiHeader.biCompression = BI_RGB;

bmp_info.bmiHeader.biSizeImage = 0;

bmp_info.bmiHeader.biXPelsPerMeter = 0;

bmp_info.bmiHeader.biYPelsPerMeter = 0;

bmp_info.bmiHeader.biClrUsed = 0;

bmp_info.bmiHeader.biClrImportant = 0;

HDC mem_dc = ::CreateCompatibleDC(hdc);

void* buf = 0;

HBITMAP bmp = ::CreateDIBSection(
```

```

mem_dc,

&bmp_info,

DIB_RGB_COLORS,

&buf,

0,

0

);

// 把 bmp 与 mem_dc 关联，这样 AGG 就可以和原生 GDI 一起工作了

HBITMAP temp = (HBITMAP)::SelectObject(mem_dc, bmp);

//=====================================================

// 以下是 AGG 代码

agg::rendering_buffer rbuf;

// 32 位位图，每行字节数为 width*4。

// BMP 是上下倒置的，为了和 GDI 习惯相同，最后一个参数是负值。

rbuf.attach((unsigned char*)buf, width, height, -width*4);

// 像素格式和 renderer_base

agg::pixfmt_bgra32 pixf(rbuf);

agg::renderer_base<agg::pixfmt_bgra32> renb(pixf);

renb.clear(agg::rgba8(255, 255, 255, 255));

// Scanline renderer

agg::renderer_scanline_aa_solid<agg::renderer_base<agg::pixfmt_bgra32>> ren(renb);

// Rasterizer & scanline

agg::rasterizer_scanline_aa<> ras;

agg::scanline_p8 sl;

// 多义线（三角形）

```

```

ras.move_to_d(20.7, 34.15);

ras.line_to_d(398.23, 123.43);

ras.line_to_d(165.45, 401.87);

// 设置颜色后渲染

ren.color(agg::rgba8(80, 90, 60));

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

//=====

// 把 bmp 显示到 hdc 上，如果图片中有 Alpha 通道，可以使用 AlphaBlend 代替 BitBlt。

::BitBlt(

    hdc,

    rt.left,

    rt.top,

    width,

    height,

    mem_dc,

    0,

    0,

    SRCCOPY

);

// 释放资源

::SelectObject(mem_dc, temp);

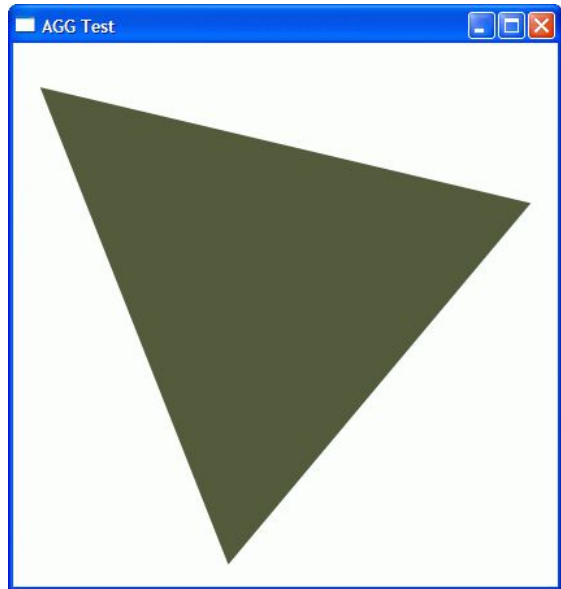
::DeleteObject(bmp);

::DeleteObject(mem_dc);

```

得到的图形是:





使用 AGG 提供的 pixel\_map 类

如果你觉得上面的方法还是有点烦的话( 这个要怪 MS 的 API 太麻烦 ), 可以考虑用 AGG 友情提供的 pixel\_map 类, 用它操作 BMP 方便多了。(要把 [AGG]/src/platform/win32/agg\_win32\_bmp.cpp 加入一起编译)

```
#include <agg_rendering_buffer.h>

#include <agg_pixfmt_rgba.h>

#include <agg_renderer_base.h>

#include <agg_rasterizer_scanline_aa.h>

#include <agg_scanline_p.h>

#include <platform/win32/agg_win32_bmp.h>

...

CRect rc;

GetClientRect(&rc);

agg::pixel_map pm;

pm.create(rc.right,rc.bottom,agg::org_color32);
```

```
//=====

// 以下是 AGG 代码

agg::rendering_buffer rbuf;

rbuf.attach(pm.buf(), pm.width(), pm.height(), -pm.stride());

// 像素格式和 renderer_base

agg::pixfmt_bgra32 pixf(rbuf);

agg::renderer_base<agg::pixfmt_bgra32> renb(pixf);

renb.clear(agg::rgba8(255, 255, 255, 255));

// Scanline renderer

agg::renderer_scanline_aa_solid<agg::renderer_base<agg::pixfmt_bgra32> > ren(renb);

// Rasterizer & scanline

agg::rasterizer_scanline_aa<> ras;

agg::scanline_p8 sl;

// 多义线（三角形）

ras.move_to_d(20.7, 34.15);

ras.line_to_d(398.23, 123.43);

ras.line_to_d(165.45, 401.87);

// 设置颜色后渲染

ren.color(agg::rgba8(80, 90, 60));

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

//=====

pm.draw(hdc);
```

## 线段生成器(Span Generator)

我们前面举的例子使用的都是简单的单一实色，如蓝色的圆、黑色的线等。这是因为在例子里我们一直使用 `renderer_scanline_aa_solid` 或 `render_scanlines_aa_solid`。

在上篇文章(<http://www.cppprog.com/2009/0821/150.html>)的[渲染器一节](#)中除了

`renderer_scanline_aa_solid` 外，还提到有一个 `renderer_scanline_aa`，这里再写一遍它的声明：

```
template<class BaseRenderer, class SpanAllocator, class SpanGenerator>
```

```
    class renderer_scanline_aa;
```

另外，还有一个函数版本：

```
template<class Rasterizer, class Scanline, class BaseRenderer,
```

```
        class SpanAllocator, class SpanGenerator>
```

```
void render_scanlines_aa(Rasterizer& ras, Scanline& sl, BaseRenderer& ren,
```

```
                        SpanAllocator& alloc, SpanGenerator& span_gen);
```

`renderer_scanline_aa`（还有一个兄弟版本 `renderer_scanline_bin`）可以按指定的图案或不同的颜色（如渐变）填充顶点源里的多边形。其中的模板参数 `SpanAllocator` 用于准备 span，我们直接使用 `agg::span_allocator` 就行。这里的 `SpanGenerator` 就是本节要说的线段生成器，它决定了最终用什么东西填到 `rendering_buffer` 里。

线段生成器品种很多，常用的在致可以分成图案类和色彩类两大部分：图案类线段生成器使用已有图像作为 span 来源；色彩类线段生成器使用指定的颜色作为 span 来源。

### 图案类线段生成器

#### 头文件

```
#include <agg_span_image_filter_gray.h>
```

```
#include <agg_span_image_filter_rgb.h>
```

```
#include <agg_span_image_filter_rgba.h>
```

```
#include <agg_span_pattern_gray.h>
```

```
#include <agg_span_pattern_rgb.h>
```

```
#include <agg_span_pattern_rgba.h>
```

## 类型

```
template<class Source, class Interpolator>
```

```
span_image_filter_[gray|rgb|rgba]
```

```
template<class Source, class Interpolator>
```

```
span_image_filter_[gray|rgb|rgba]_2x2
```

```
template<class Source, class Interpolator>
```

```
span_image_filter_[gray|rgb|rgba]_bilinear
```

```
template<class Source, class Interpolator>
```

```
span_image_filter_[gray|rgb|rgba]_bilinear_clip
```

```
template<class Source, class Interpolator>
```

```
span_image_filter_[gray|rgb|rgba]_nn
```

```
template<class Source, class Interpolator>
```

```
span_image_resample_[gray|rgb|rgba]
```

```
template<class Source>
```

```
span_image_resample_[gray|rgb|rgba]_affine
```

```
template<class Source>
```

```
class agg::span_pattern_[gray|rgb|rgba]
```

上面这些线段生成器类的模板参数都比较相似：**Source** 用于指定图像来源，可以是 PixelFormat renderer 或 agg::image\_accessor\_clip( 由不同的线段生成器类决定 ); **Interpolator** 是一种插值器，用于填充图像间隙。我们先写一段示例代码，先看一下线段生成器的作用，也为后面的各种实验做准备。

示例代码，使用 span\_image\_filter\_rgb\_bilinear\_clip

还是基于这个代码(<http://www.cppprog.com/2009/0816/146.html>)，加入下面的头文件

```
#include <platform/win32/agg_win32_bmp.h>

#include "agg_span_allocator.h"

#include "agg_span_image_filter_rgb.h"
```

在 on\_draw() 方法的最后加上下面这些代码

```
...

// 以图像填充

agg::pixel_map pm_img;

if(pm_img.load_from_bmp("d:/spheres.bmp"))

{

    // pm_img 里的图案作为填充来源

    agg::rendering_buffer rbuf_img(

        pm_img.buf(),

        pm_img.width(), pm_img.height(),

        -pm_img.stride());

    agg::pixfmt_bgr24 pixf_img(rbuf_img); // 我用的 bmp 是 24 位的

    // 线段分配器

    typedef agg::span_allocator<agg::rgba8> span_allocator_type; // 分配器类型
```

```

span_allocator_type span_alloc; // span_allocator

// 插值器

typedef agg::span_interpolator_linear<> interpolator_type; //插值器类型

agg::trans_affine img_mtx; // 变换矩阵

interpolator_type ip(img_mtx); // 插值器

// 线段生成器

typedef agg::span_image_filter_rgb_bilinear_clip<agg::pixfmt_bgr24,

interpolator_type > span_gen_type; // 这个就是 Span Generator

span_gen_type span_gen(pixf_img, agg::rgba(0,1,0), ip);

// 组合成渲染器

agg::renderer_scanline_aa<

    renderer_base_type,

    span_allocator_type,

    span_gen_type

> my_renderer(renb, span_alloc, span_gen);

// 插值器的矩阵变换

img_mtx.scale(0.5);

img_mtx.translate(40,40);

img_mtx.invert(); //注意这里

// 用我们的渲染器画圆

ras.add_path(ell);

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

}

```

其中的 `d://spheres.bmp`[\(下载\)](#)是我预先放在 D 盘里的 24 位 bmp 图像，作为填充的来源。

显示效果:



在第 19 行的 `span_gen_type` 之前，所有的事情都在为定义这个线段生成器做准备。

首先是用 `pixel_map` 读取 bmp 文件，然后生成 `rendering_buffer` 和 `pixfmt_bgr24` 作为这个线段生成器的 "Source"。

然后是线段分配器，这个没什么特殊要求的话用 `span_allocator` 就可以了。

接着是插值器类型，插值器也有几个类型（后面会介绍），它的构造函数需要一个变换矩阵对象，于是我们得为它装备一个。

现在，终于可以组合成一个我们的线段生成器了。这里使用的是

`span_image_filter_rgb_bilinear_clip`，它的 Source 是 `PixelFormat Renderer`，如本例的 `pixfmt_bgr24`。

`span_image_filter_rgb_bilinear_clip` 的构造函数有三个参数，分别是 Source 对象，填充来源范围之外的颜色和插值器对象。

我们可以改变插值器的矩阵来变换填充图像，象这里的 `img_mtx.scale(0.5)`和

`img_mtx.translate(40,40)`。要注意的是，插值器的矩阵运算是从目标位置向源位置计算的

(即根据目标位置变换得到对应的填充源位置), 所以想对源图像变换的话, 要记得最后调用矩阵的 `invert()` 方法取反。

最后, 画圆。由于 `ell` 是 `ellipse` 对象, 没有被 `conv_stroke` 转换的 `ellipse` 对象是实心的(多边形而不是多义线), 于是填充之。

## 插值器 `Interpolator`

插值器的作用是连接目标位置和源位置, 比如要填充一个  $8 \times 8$  的图形, 对应的填充源是一个  $4 \times 4$  的图像, 一种简单的线性插值器就要根据目标的位置线性计算得到源对应的位置, 如目标点(4,4)、(4,5)、(5,4)、(5,5)这几个位置点对应到源的(2,2)点上。

## 头文件

```
#include <agg_span_interpolator_linear.h>
```

```
#include <agg_span_interpolator_persp.h>
```

```
#include <agg_span_interpolator_trans.h>
```

## 类型

```
template<class Transformer = trans_affine, unsigned SubpixelShift = 8>
```

```
class agg::span_interpolator_linear
```

```
template<class Transformer = trans_affine, unsigned SubpixelShift = 8>
```

```
class agg::span_interpolator_linear_subdiv
```

```
template<unsigned SubpixelShift = 8>
```

```
class agg::span_interpolator_persp_exact
```

```
template<unsigned SubpixelShift = 8>
```

```
class agg::span_interpolator_persp_lerp
```



```
template<class Transformer, unsigned SubpixelShift = 8>
```

```
class agg::span_interpolator_trans
```

不同的插值器对于不同的变换有各自的优势，对于大部分应用来说，

span\_interpolator\_linear 是比较简单高效的。

实验代码，使用 span\_interpolator\_persp\_lerp

把[上面的演示代码](#)里的 `interpolator_type` 改成 `span_interpolator_persp_lerp`，这是一个透视变

换的插值器，输入为源四个角的坐标和目标的四个角上的坐标。

```
...
```

```
// 插值器
```

```
//typedef agg::span_interpolator_linear<> interpolator_type; //插值器类型
```

```
//agg::trans_affine img_mtx; // 变换矩阵不需要了
```

```
//interpolator_type ip(img_mtx); // 插值器
```

```
typedef agg::span_interpolator_persp_lerp<> interpolator_type; //插值器类型
```

```
const double offset = 50; // 偏移
```

```
const double scale = 0.5; // 缩放倍数
```

```
double src[8]={offset+0,offset+0,
```

```
    offset+pm_img.width()*scale,offset+0,
```

```
    offset+pm_img.width()*scale,offset+pm_img.height()*scale,
```

```
    offset+0,offset+pm_img.height()*scale
```

```
}; //源四角坐标，按偏移和缩放倍数改变一哈
```

```
double dst[8]={0,0,
```

```
    pm_img.width(),0,
```

```
    pm_img.width()+100,pm_img.height(),
```

```

100,pm_img.height()-100

}; //目标四角坐标，左右乱扯一哈

interpolator_type ip(src,dst);

...

```

最后别忘了把 img\_mtx 相关代码注释掉:

```

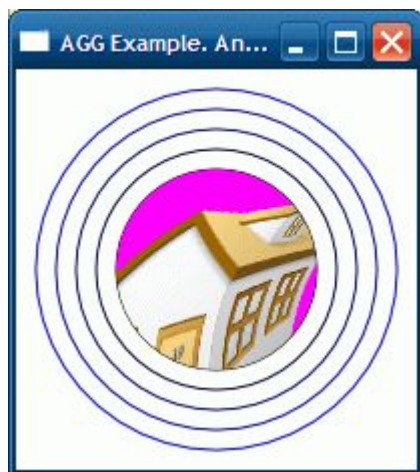
//img_mtx.scale(0.5);

//img_mtx.translate(40,40);

//img_mtx.invert(); //注意这里

```

显示效果:



变换器 Transformer

注意一下我们前面用的 [span interpolator linear](#)，以及[曾经使用过的 conv transform](#)，默认的

模板参数 Transformer 为 **trans\_affine**。我们已经知道 trans\_affine 是一个 2\*3 的变换矩阵。

在 AGG 中，变换器不仅仅只有矩阵，这里将要介绍的就是其它的一些变换器。

头文件

```

#include <agg_trans_affine.h>

```

```
#include <agg_trans_bilinear.h>

#include <agg_trans_single_path.h>

#include <agg_trans_double_path.h>

#include <agg_trans_perspective.h>

#include <agg_trans_viewport.h>

#include <agg_trans_warp_magnifier.h>
```

## 类型

```
agg::trans_affine

agg::trans_bilinear

agg::trans_single_path

agg::trans_double_path

agg::trans_perspective

agg::trans_viewport

agg::trans_warp_magnifier
```

实验代码，使用 trans\_warp\_magnifier

同样把[示例代码](#)中的插值器部分改成下列代码

```
...

// 插值器

//typedef agg::span_interpolator_linear<> interpolator_type; //插值器类型

//agg::trans_affine img_mtx; // 变换矩阵

//interpolator_type ip(img_mtx); // 插值器

typedef agg::span_interpolator_trans<

    agg::trans_warp_magnifier // 使用 trans_warp_magnifier

> interpolator_type; //插值器类型
```

```
agg::trans_warp_magnifier mag;

interpolator_type ip(mag);

mag.magnification(0.5);

mag.center(100,100);

mag.radius(50);

...
```

建议把后面的 `ras.add_path(ell)` 改成 `ras.add_path(ccell)`，画得大一点好看清效果，呵呵

显示效果



### 图像访问器 Image Accessor

也许有不少同学看到开头的线段生成器一节时，已经尝试修改示例代码中的

`span_image_filter_rgb_bilinear_clip` 了（比如改成 `span_image_filter_rgb_bilinear`）。不过编

译时会出错，这是因为大部分的线段生成器类接受的 Source 模板不是 `PixelFormat`

`Renderer`，而是 `Image Accessor` 即图像存取器。

头文件

```
#include <agg_image_accessors.h>
```

## 类型

```
template<class PixFmt>
```

```
class agg::image_accessor_clip // 图像以外的地方用指定颜色填充
```

```
template<class PixFmt>
```

```
class agg::image_accessor_clone // 图像以外的地方以图像边缘填充
```

```
template<class PixFmt>
```

```
class agg::image_accessor_no_clip // 图像以外不可读取，否则引发异常
```

```
template<class PixFmt, class WrapX, class WrapY>
```

```
class agg::image_accessor_wrap // 平铺图像，平铺方式由 WrapX 和 WrapY 指定
```

## 实验代码

把[示例代码](#)中的 span\_image\_filter\_rgb\_bilinear\_clip 部分改成下面的代码

```
...

// 线段生成器

//typedef agg::span_image_filter_rgb_bilinear_clip<agg::pixfmt_bgr24,

//interpolator_type > span_gen_type; // 这个就是 Span Generator

//span_gen_type span_gen(pixf_img, agg::rgba(0,1,0), ip);

// 图像访问器

typedef agg::image_accessor_clone<agg::pixfmt_bgr24> image_accessor_type;

image_accessor_type    accessor(pixf_img);

// 使用 span_image_filter_rgb_bilinear

typedef agg::span_image_filter_rgb_bilinear<

    image_accessor_type,

    interpolator_type > span_gen_type;

span_gen_type span_gen(accessor, ip);
```

...

建议把后面的 `ras.add_path(ell)` 改成 `ras.add_path(ccell)`

显示效果



`image_accessor_wrap` 类要指定 `WrapX` 和 `WrapY`，可选的有：

`wrap_mode_reflect`

`wrap_mode_reflect_auto_pow2`

`wrap_mode_pow2`

`wrap_mode_repeat`

`wrap_mode_repeat_auto_pow2`

`wrap_mode_repeat_pow2`

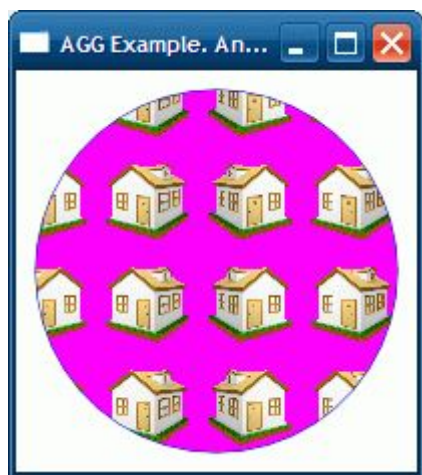
比如我们把本例中的 `image_accessor_type` 定义改成

```
//typedef agg::image_accessor_clone<agg::pixfmt_bgr24> image_accessor_type;
```

```
typedef agg::image_accessor_wrap<agg::pixfmt_bgr24,
```

```
agg::wrap_mode_reflect,agg::wrap_mode_repeat> image_accessor_type;
```

显示效果是



(为了突出效果，用矩阵 `img_mtx` 把源缩小了)

图像过滤器(Image Filter)

在一些线段生成器里，比如 `span_image_filter_[gray|rgb|rgba]`，

`span_image_resample_[gray|rgb|rgba]`等类，它们的构造函数还有一个 “`const`

`image_filter_lut &filter`” 参数，这个参数用于变换图像的像素值。它们的名称都以

`image_filter` 作为前缀，AGG 中称为 Image Filter(图像过滤器)。

头文件

```
#include <agg_image_filters.h>
```

类型

```
image_filter_bilinear;
```

```
image_filter_blackman;
```

```
image_filter_blackman[36|64|100|144|196|256];
```

```
image_filter_kaiser;
```

```
image_filter_lanczos;
```

```
image_filter_lanczos[36|64|100|144|196|256];
```

```
image_filter_mitchell;
```

...还有很多呢...

## 实验代码

把上面的 `span_image_filter_rgb_bilinear` 改成 `span_image_resample_rgb_affine`

...

```
//typedef agg::image_accessor_clone<agg::pixfmt_bgr24> image_accessor_type;
```

```
typedef agg::image_accessor_wrap<agg::pixfmt_bgr24,
```

```
agg::wrap_mode_reflect,agg::wrap_mode_repeat> image_accessor_type;
```

```
image_accessor_type    accessor(pixf_img);
```

```
//typedef agg::span_image_filter_rgb_bilinear<
```

```
//    image_accessor_type,
```

```
//    interpolator_type > span_gen_type;
```

```
//span_gen_type span_gen(accessor, ip);
```

```
typedef agg::span_image_resample_rgb_affine<image_accessor_type> span_gen_type;
```

```
span_gen_type span_gen(accessor, ip, agg::image_filter_sinc36());
```

...

## 显示效果





## 色彩类线段生成器

### 头文件

```
#include <agg_span_solid.h>

#include <agg_span_gradient.h>

#include <agg_span_gradient_alpha.h>

#include <agg_span_gouraud_gray.h>

#include <agg_span_gouraud_rgba.h>
```

### 类型

```
template<class ColorT>

    class agg::span_solid;

template<class ColorT, class Interpolator, class GradientF, class ColorF>

    class agg::span_gradient;

template<class ColorT, class Interpolator, class GradientF, class AlphaF>

    class agg::span_gradient_alpha;

template<class ColorT>
```

```
class agg::span_gouraud_[gray|rgba];
```

如果你是从上面的图案类线段生成器看到这里的话，那么色彩类的就相对简单得多了。同样，我们先写一个示例代码，也方便以后做实验。

## 示例代码

同样基于这个代码(<http://www.cppprog.com/2009/0816/146.html>),

```
//Rendering Buffer
agg::rendering_buffer &rbuf = rbuf_window();
agg::pixfmt_bgr24 pixf(rbuf);

// Renderers
typedef agg::renderer_base<agg::pixfmt_bgr24> renderer_base_type;
renderer_base_type renb(pixf);

typedef agg::renderer_scanline_aa_solid<renderer_base_type> renderer_scanline_type;
renderer_scanline_type rensl(renb);

// Vertex Source
agg::ellipse ell(100,100,50,50);

// Scanline Rasterizer
agg::rasterizer_scanline_aa<> ras;
agg::scanline_u8 sl;
```

加入下面的头文件

```
#include "agg_span_allocator.h"
```

```
#include "agg_span_gradient.h"
```

在 on\_draw()方法的最后加上下面这些代码

```
// 色彩类线段生成器 demo
```

```
// 线段分配器
```

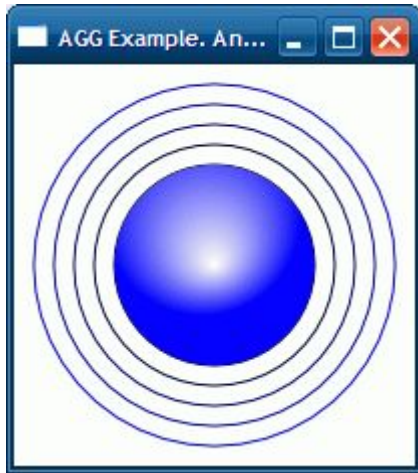
```
typedef agg::span_allocator<agg::rgba8> span_allocator_type;//分配器类型
```

```

span_allocator_type span_alloc; // span_allocator
// 插值器
typedef agg::span_interpolator_linear<> interpolator_type; //插值器类型
agg::trans_affine img_mtx; // 变换矩阵
interpolator_type ip(img_mtx); // 插值器
// 渐变方式
typedef agg::gradient_radial_focus gradientF_type;
gradientF_type grF(1, 0.1, 0.5);
// 渐变颜色
typedef agg::gradient_linear_color<agg::rgba8> colorF_type;
colorF_type colorF(agg::rgba(1,1,1), agg::rgba(0,0,1)); //白色到蓝色
// 线段生成器
typedef agg::span_gradient<agg::rgba8,
interpolator_type,
gradientF_type,
colorF_type> span_gen_type;
span_gen_type span_gen(ip,grF,colorF,0,50);
// 组合成渲染器
agg::renderer_scanline_aa<
renderer_base_type,
span_allocator_type,
span_gen_type
> my_renderer(renb, span_alloc, span_gen);
// 矩阵变换
img_mtx.translate(100,100);
img_mtx.invert(); //注意这里
// 使用我们的渲染器画圆
ras.add_path(ell);
agg::render_scanlines(ras,sl,my_renderer);

```

显示效果



`span_gradient` 是一个模板类（废话，AGG 的大部分类都是），前两个模板参数 `ColorT` 和 `Interpolator` 一个是颜色类型一个是插值器没什么好说的了。关键是后面两个：`GradientF` 用于指定渐变的方式，如水平渐变、垂直渐变、圆形渐变等；`ColorF` 指定渐变的颜色。

渐变方式选择了 `agg::gradient_radial_focus`，这是一个可指定焦点的圆形渐变方式。

渐变色使用 `agg::gradient_linear_color` 设置起始颜色和终止颜色。

`span_gradient` 的构造函数前三个分别是插值器、渐变方式、渐变颜色，后面两个数字表示渐变的起始和终止位置。不同的渐变方式起始和终止位置的意义是不同的，如在圆形填充里起始和终止表示中心和边缘；水平渐变则表示从左到右。

插值器的矩阵变换把这个渐变中心移到(100,100)点上，同样要记得调用 `invert()` 方法反转。

## 渐变颜色

前面说到 `span_gradient` 的模板参数 `ColorF` 指定渐变的颜色，我们使用的是 `gradient_linear_color`，那么有哪些类可以作为 `ColorF` 呢？

AGG 文档里说只要实现了 “`operator []()`” 和 “`size()`” 的类就可以作为 `ColorF`，嗯，`std::vector<rgba8>` 也行哈。

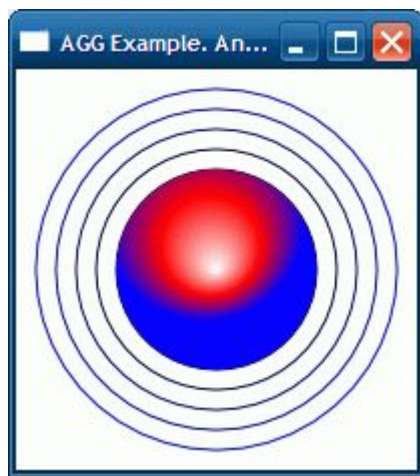
实验代码，使用 `std::vector<rgba8>` 实现多颜色渐变

把[示例代码](#)的渐变颜色部分改成这样：

```
...
// 渐变颜色
//typedef agg::gradient_linear_color<agg::rgba8> colorF_type;
//colorF_type colorF(agg::rgba(1,1,1), agg::rgba(0,0,1));//白色到蓝色
```

```
typedef std::vector<agg::rgba8> colorF_type;
colorF_type colorF(256);
agg::rgba begin_color(1,1,1), mid_color(1,0,0), end_color(0,0,1);
for(int i=0; i<128; i++) //前 128 从白到红
colorF[i] = begin_color.gradient(mid_color,i/128.0);
for(int i=0; i<128; i++) //后 128 从红到蓝
colorF[i+128] = mid_color.gradient(end_color,i/128.0);
```

显示效果



这里指定的 vector 容量 256 指的是用于的颜色，想要更平滑过渡的话可以使用更多的颜色数。

除了用 vector 实现多种颜色的渐变外，我们还可以用 AGG 提供的一个 **gradient\_lut** 类，用它可以方便很多。

gradient\_lut 的头文件是 `#include <agg_gradient_lut.h>`

类声明为

```
template<class ColorInterpolator, unsigned ColorLutSize = 256>
```

```
class agg::gradient_lut
```

其中的 ColorInterpolator 负责生成两种颜色的中间色，直接使用 AGG 自带的 `agg::color_interpolator` 就行。

通过 gradient\_lut 的 **add\_color**(double offset, color\_type color)方法添加多种颜色，其中的 **offset** 表示添加的颜色所处的偏移位置，取值为 0~1 之间。

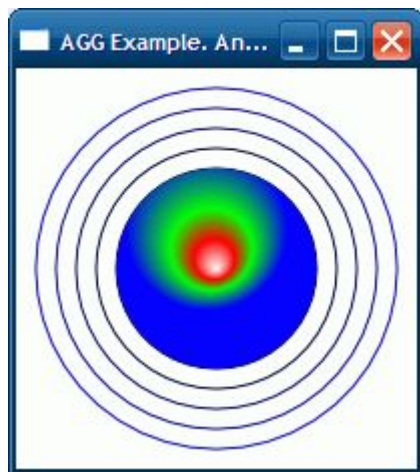
添加完所有颜色后调用 **build\_lut**()方法让 gradient\_lut 内部生成颜色数组。

实验代码，使用 gradient\_lut 实现多颜色渐变

把[示例代码](#)的渐变颜色部分改成这样：

```
...
// 渐变颜色
//typedef agg::gradient_linear_color<agg::rgba8> colorF_type;
//colorF_type colorF(agg::rgba(1,1,1), agg::rgba(0,0,1));//白色到蓝色
typedef agg::gradient_lut<
agg::color_interpolator<agg::rgba8>
> colorF_type;
colorF_type colorF;
colorF.add_color(0, agg::rgba(1,1,1));
colorF.add_color(0.2, agg::rgba(1,0,0));
colorF.add_color(0.4, agg::rgba(0,1,0));
colorF.add_color(0.8, agg::rgba(0,0,1));
colorF.build_lut();
...
```

显示效果



渐变方式

除本例中的 gradient\_radial\_focus 以外，AGG 还提供了很多渐变方式，它们都定义在#include <agg\_span\_gradient.h>头文件之中。

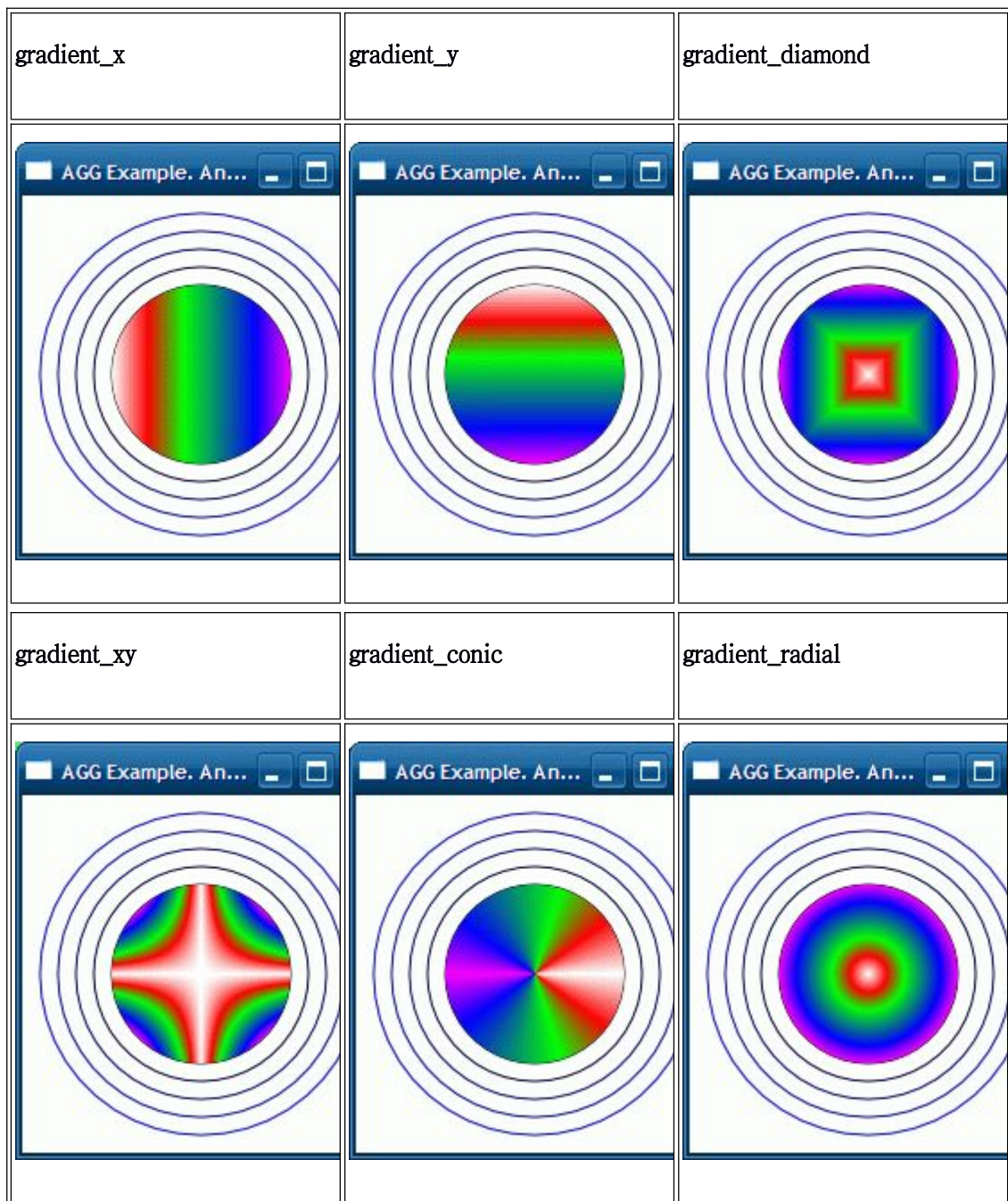
修改[演示代码](#)的渐变方式是很简单的，如：

```

...
// 渐变方式
//typedef agg::gradient_radial_focus gradientF_type;
//gradientF_type grF(1, 0.1, 0.5);
typedef agg::gradient_x gradientF_type;
gradientF_type grF;
...

```

这里是其中的一部分 AGG 自带渐变方式以及显示效果



本节的最后，再介绍一下其它几个色彩类的线段生成器

**span\_solid** 没什么好说的，实色填充而已

**span\_gradient\_alpha** 是透明度渐变，参数和 **span\_gradient** 差不多，区别是 **ColorF** 改成了 **AlphaF**，“operator []()”返回值也由颜色结构变为的透明度数值。

**span\_gouraud\_rgba** 高氏三角着色，需指定三角形的三个顶点和三种颜色，用法见下例

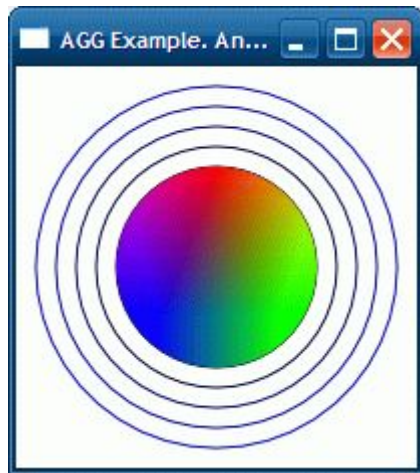
```
// 色彩类线段生成器 demo
// 线段分配器
typedef agg::span_allocator<agg::rgba8> span_allocator_type; // 分配器类型
span_allocator_type span_alloc; // span_allocator

typedef agg::span_gouraud_rgba<agg::rgba8> span_gen_type;
span_gen_type span_gen;
// 三种颜色
span_gen.colors(
    agg::rgba(1,0,0),
    agg::rgba(0,1,0),
    agg::rgba(0,0,1)
);
// 三角形三个顶点
span_gen.triangle(
    100,50,
    130,125,
    70,125,0
);
agg::renderer_scanline_aa<
renderer_base_type,
span_allocator_type,
span_gen_type
> my_renderer(span_alloc, span_gen);

ras.add_path(ell);
agg::render_scanlines(ras,sl,my_renderer);
```

显示效果





## 组合类线段生成器

### 头文件

```
#include <agg_span_converter.h>
```

### 类型

```
template<class SpanGenerator, class SpanConverter>
```

```
class agg::span_converter;
```

span\_converter 的作用是组合两种生成器，比如先由图案类线段生成器产生图案，然后由色彩类线段生成器产生半透明色叠加在图案上。

下面的演示代码演示了怎样组合 span\_image\_filter\_rgb\_bilinear\_clip 和 span\_gradient\_alpha 两种生成器

演示代码，同样基于[这个代码](#)，加入下面的头文件

```
#include "agg_span_allocator.h"
```

```
#include "agg_span_gradient_alpha.h"
```

```
#include "agg_span_converter.h"
```

```
#include "span_image_filter_rgb_bilinear_clip.h"
```

在 on\_draw() 方法的最后加上下面这些代码

```

agg::pixel_map pm_img;
if(pm_img.load_from_bmp("d://spheres.bmp"))
{
    // pm_img 里的图案作为填充来源
    agg::rendering_buffer rbuf_img(
        pm_img.buf(),
        pm_img.width(), pm_img.height(),
        -pm_img.stride());
    agg::pixfmt_bgr24 pixf_img(rbuf_img);// 我用的 bmp 是 24 位的
    // 线段分配器
    typedef agg::span_allocator<agg::rgba8> span_allocator_type;//分配器类型
    span_allocator_type span_alloc; // span_allocator
    // 插值器
    typedef agg::span_interpolator_linear<> interpolator_type; //插值器类型
    agg::trans_affine img_mtx; // 变换矩阵
    interpolator_type ip_img(img_mtx); // 插值器

    agg::trans_affine alpha_mtx; // 变换矩阵
    interpolator_type ip_alpha(alpha_mtx); // 插值器

    // 渐变方式
    typedef agg::gradient_x gradientF_type;
    gradientF_type grF;

    typedef std::vector<agg::int8u> alphaF_type;
    alphaF_type alphaF(256);
    for(int i=0; i<256; i++) alphaF[i] = i;

    // Alpha 线段生成器
    typedef agg::span_gradient_alpha<agg::rgba8,
        interpolator_type,
        gradientF_type,
        alphaF_type> alpha_span_gen_type;
    alpha_span_gen_type alpha_span_gen(ip_alpha,grF,alphaF,0,150);

    // 图案线段生成器
    typedef agg::span_image_filter_rgb_bilinear_clip<agg::pixfmt_bgr24,
        interpolator_type > pic_span_gen_type;
    pic_span_gen_type pic_span_gen(pixf_img, agg::rgba(0,1,0), ip_img);

```

```

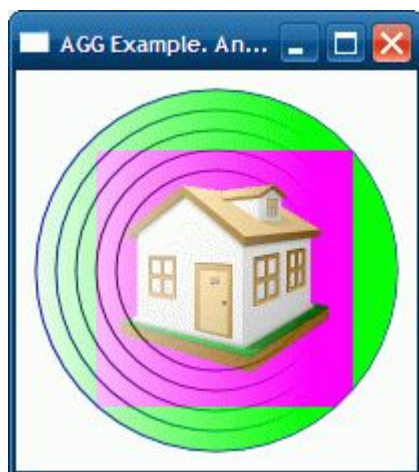
// 使用 span_converter 组合成新的线段生成器
typedef agg::span_converter<pic_span_gen_type, alpha_span_gen_type> span_gen_type;
span_gen_type span_gen(pic_span_gen,alpha_span_gen);

// 组合成渲染器
agg::renderer_scanline_aa<
    renderer_base_type,
    span_allocator_type,
    span_gen_type
> my_renderer(renb, span_alloc, span_gen);
// 插值器的矩阵变换
img_mtx.scale(0.5);
img_mtx.translate(40,40);
img_mtx.invert(); //注意这里

// 用我们的渲染器画圆
ras.add_path(ccell);
agg::render_scanlines(ras,sl,my_renderer);
}

```

显示效果



## AGG 的字符输出

字符输出，对于 AGG 来说，这个功能可以处于[显示流程](#)的 不同位置。比如字体引擎可直接处于“Scanline Rasterizer”层向渲染器提供已处理完毕的扫描线，也可以处于“Vertex Source 顶点源”层提供字体的顶点数据。

下面，我们开始学习 AGG 不同的字符输出方式。如没有特殊说明，所以示例代码都基于[此处代码](#)

### *方式一、使用 gsv\_text 对象*

gsv\_text 属于顶点源层的对象，它的用法也很简单，直接看下例：

引用头文件：`#include <agg_gsv_text.h>`

在 `on_draw()` 方法的最后加入下面的代码

```
// gsv_text 类
agg::gsv_text txt;
agg::conv_stroke<agg::gsv_text> cstxt(txt);
// 设置大小及是否反转
txt.flip(true);
txt.size(18);
// 设置位置和文字
txt.start_point(20,100);
txt.text("cpp");
// 以红色输出上面的文字
ras.add_path(cstxt);
agg::render_scanlines_aa_solid(ras,sl,renb,agg::rgba(1,0,0));
// 设置新的位置和文字
txt.start_point(20+txt.text_width(),100);
txt.text("prog.com");
// 以蓝色输出上面的文字
```

```
ras.add_path(cstxt);
agg::render_scanlines_aa_solid(ras,sl,renb,agg::rgba(0,0,1));
```

显示效果



注：*gsv\_text* 的 *flip()* 方法指出是否上下反转输出，这里设置了 *flip* 是因为在 Windows 下 *agg::platform\_support* 的 *rbuf\_window()* 其实是一个 DIB 缓存，它的方向是从下到上的。

*gsv\_text* 必须用 **conv\_stroke** 转换才能正确输出文字，否则会被当作多边形处理。为了方便，AGG 提供了 **gsv\_text\_outline** 包装，它实现了 *conv\_stroke* 和坐标转换，代码很短：

```
template<class Transformer = trans_affine> class gsv_text_outline
{
public:
    gsv_text_outline(gsv_text& text, const Transformer& trans) :
        m_polyline(text),
        m_trans(m_polyline, trans){}
```

```

void width(double w){

    m_polyline.width(w);

}

void transformer(const Transformer* trans){

    m_trans->transformer(trans);

}

void rewind(unsigned path_id) {

    m_trans.rewind(path_id);

    m_polyline.line_join(round_join);

    m_polyline.line_cap(round_cap);

}

unsigned vertex(double* x, double* y) {

    return m_trans.vertex(x, y);

}

private:

conv_stroke<gsv_text> m_polyline;

conv_transform<conv_stroke<gsv_text>, Transformer> m_trans;

};

```

我们使用 `gsv_text_outline` 重写上面的代码：

```
#include <agg_gsv_text.h>
```

```

#include <agg_trans_single_path.h>

...

////////////////////////////////////

// gsv_text 及 gsv_text_outline

agg::gsv_text txt;

agg::trans_single_path tran_path; //使用 trans_single_path 作为变换器

agg::gsv_text_outline<agg::trans_single_path> txtol(txt,tran_path);

// 设置变换器

tran_path.add_path(ell);

////////////////////////////////////

// 设置大小及是否反转

txt.flip(true);

txt.size(24);

// 设置位置和文字

txt.start_point(0,0);

txt.text("cpp");

// 以红色输出上面的文字

ras.add_path(txtol);

agg::render_scanlines_aa_solid(ras,sl,renb,agg::rgba(1,0,0));

// 设置新的位置和文字

```

```
txt.start_point(0+txt.text_width(),0);
```

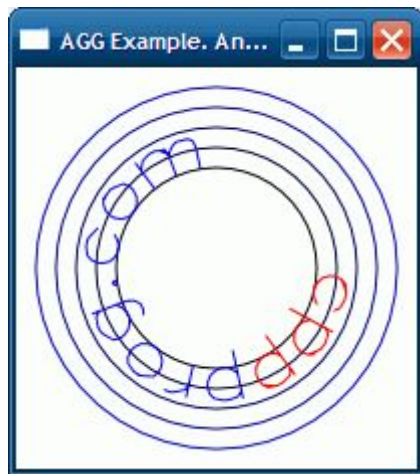
```
txt.text("prog.com");
```

```
// 以蓝色输出上面的文字
```

```
ras.add_path(txtol);
```

```
agg::render_scanlines_aa_solid(ras,sl,renb,agg::rgba(0,0,1));
```

显示效果



gsv\_text 的使用很简单，不过要了解的一点是：它只能输出 ASCII 可显示字符，对于汉字是无能为力的（你可以试试输出"C++编程"）。如果要输出汉字，我们得继续寻找其它字符输出方式。

### 方式二、使用字体引擎(Font Engine)

AGG 的字体引擎利用 WinAPI:GetGlyphOutline 或 FreeType 库得到字体数据（字模），它可以处于“Scanline Rasterizer”层或“顶点源”层。要使用字体引擎，要把相应的字体引擎源码(agg\_font\_win32\_tt.cpp 或 agg\_font\_freetype.cpp)加入项目一起编译。

头文件



```
#include <agg_font_win32_tt.h>
```

```
#include <agg_font_freetype.h>
```

注意，它们都有自己的文件夹，不是在 agg 的 include 文件夹里。

## 类型

agg::font\_engine\_win32\_tt\_int16

agg::font\_engine\_win32\_tt\_int32

agg::font\_engine\_freetype\_int16

agg::font\_engine\_freetype\_int32

显然,前两个利用 WinAPI 实现,后两个利用 FreeType 库实现。类型后面的\_int16 或\_int32

后缀用于指定坐标单位，一般 int16 已经可以满足要求。

成员类型定义：

typedef path_adaptor_type	把字体数据包装成顶点源的类
typedef gray8_adaptor_type	把字体数据包装成 Scanline Rasterizer 的类
typedef mono_adaptor_type	把字体数据包装成 Scanline Rasterizer 的类，但无 AA 效果

成员属性：

double: height	字体高度，单位为 Point(和 Word 里的单位一样)
----------------	-------------------------------

double: width	字体宽度，单位为 Point*2.4。0 表示规则大小(height/2.4)
bool: italic	斜体
bool: flip_y	上下翻转
bool: hinting	字体修正
unsigned: resolution	字体解析度，单位为 dpi

成员方法:

void transform(const trans_affine& affine);	按矩阵变换
bool create_font(const char* typeface_, glyph_rendering ren_type);	font_engine_win32_tt_*专有方法 建立字体，typeface_为字体名，ren_type 稍后再说
bool load_font(const char* font_name, unsigned face_index, glyph_rendering ren_type, const char* font_mem = 0, const long font_mem_size = 0);	font_engine_freetype_*专有方法 建立字体，font_name 是字体文件名或字体名
bool prepare_glyph(unsigned glyph_code)  unsigned data_size() const  void write_glyph_to(int8u* data) const	得到字体数据(字模)所需方法

字体引擎的 `create_font()` 方法和 `load_font()` 方法需要一个 `glyph_rendering` 类型的 **ren\_type** 参数，它决定了字体数据的形式。三个成员类型定义：`path_adaptor_type`、`gray8_adaptor_type` 和 `mono_adaptor_type` 所包装的字体数据是不一样的，只有与 `ren_type` 参数对应才能生成正确的 AGG 显示节点。

`glyph_rendering` 是一个枚举类型，定义是：

```
enum agg::glyph_rendering{

    glyph_ren_native_mono, //对应 mono_adaptor_type

    glyph_ren_native_gray8, //对应 gray8_adaptor_type

    glyph_ren_outline, //对应 path_adaptor_type

    glyph_ren_agg_mono, //对应 mono_adaptor_type

    glyph_ren_agg_gray8 //对应 gray8_adaptor_type

};
```

示例代码 1 – 从顶点源层输出文字

```
typedef agg::font_engine_win32_tt_int16 fe_type;
typedef fe_type::path_adaptor_type vs_type;
// 字体引擎
fe_type fe( ::GetDC(::GetActiveWindow()) ); //注意，实际应用时要释放 HDC
fe.height(36.0);
fe.flip_y(true);
fe.hinting(true);
// 注意后面的 glyph_rendering ren_type 参数
fe.create_font("黑体",agg::glyph_ren_outline);
// 字体串
wchar_t *s = L"C++编程";
// 存放字体数据
std::vector<agg::int8u> data;
// 顶点源
vs_type vs;
// 注意这里，使用 conv_curve 转换
```

```

agg::conv_curve<vs_type> ccvs(vs);

// 字符输出的位置
int x=20,y=100;
for(;*s;s++)
{
    // 让字体引擎准备好字体数据
    if(!fe.prepare_glyph(*s)) continue;
    // 把字体数据放到容器里
    data.resize( fe.data_size() );
    fe.write_glyph_to( &data[0] );
    // 从字体数据中得到顶点源
    vs.init(&data[0], data.size(), x, y);
    // 移动输出位置
    x += fe.advance_x();
    y += fe.advance_y();
    // 输出
    ras.add_path(ccvs);
    agg::render_scanlines_aa_solid(ras,sl,renb,agg::rgba(0,0,1));
}

```

由于字体顶点源可能会包含带 Curve 命令的顶点，所以要用 **conv\_curve** 来 转换。你可以试试去掉这层转换，字符'C' 就不会那么平滑了。

## 示例代码 2 – 从 Scanline Rasterizer 层输出文字

```

// 字体引擎类型定义
typedef agg::font_engine_win32_tt_int16 fe_type;
typedef fe_type::gray8_adaptor_type ras_type;
typedef ras_type::embedded_scanline sl_type;

// 字体引擎
fe_type fe( ::GetDC(::GetActiveWindow()) ); //注意，实际应用时要释放 HDC
fe.height(36.0);
fe.flip_y(true);
fe.hinting(true);

// 注意后面的 glyph_rendering ren_type 参数
fe.create_font("黑体",agg::glyph_ren_agg_gray8);

// 字体串
wchar_t *s = L"C++编程";

```

```

// 存放字体数据
std::vector<agg::int8u> data;

// Rasterizer 和 Scanline
ras_type ras_font;
sl_type sl_font;

// 字符输出的位置
int x=20,y=100;

for(;*s;s++)
{
    // 让字体引擎准备好字体数据
    if(!fe.prepare_glyph(*s)) continue;
    // 把字体数据放到容器里
    data.resize( fe.data_size() );
    fe.write_glyph_to( &data[0] );
    // 从字体数据中得到 Rasterizer
    ras_font.init(&data[0], data.size(), x, y);
    // 移动输出位置
    x += fe.advance_x();
    y += fe.advance_y();
    // 输出
    agg::render_scanlines_aa_solid(ras_font,sl_font,renb,agg::rgba(0,0,1));
}

```

显示效果



## AGG 与 FreeType 库

Linux、FreeBSD 等开源操作系统里一般使用 FreeType 来显示文字,Windows 下的一些软件如 Foxit 也有 FreeType 的身影。

AGG 的 `font_engine_freetype_int16` 字体引擎就使用 FreeType 来取得字模,在 Windows 里,在使用 `font_engine_freetype_int16` 之前,我们得先编译好 FreeType:

1. 从 [www.freetype.org](http://www.freetype.org) 下载 FreeType2,偶下载的是目前最新的 freetype-2.3.9,解压。
2. 以 VC2005 Express 为例,直接打开/builds/win32/vc2005 里的 freetype.sln 编译即可。( *不过这个版本的 freetype.sln 好像有点问题,要用文本编辑器打开,把第一行"Microsoft Visual Studio Solution File, Format Version 10.00"后面的"10.00"改成"9.00"才行*)
3. 对于其它编译器,如 C++Builder,直接把/docs/INSTALL.ANY 里列出的文件加入项目即可(不过加入这些东西还真是比较麻烦,可以[点这里下载](#) C++Builder 的 FreeType2 库编译工程)。
4. 把编译后的库文件(在/objs/win32 里,注意编译版本)加入项目,并设置 include 路径为 freetype- 2.3.9/include 就可以了

## **AGG 使用 FreeType 的源代码:**

```
typedef agg::font_engine_freetype_int16 fe_type;
typedef fe_type::path_adaptor_type vs_type;

fe_type fe;
if(!fe.load_font("c://windows/fonts/simhei.ttf",0,agg::glyph_ren_outline)) return;
fe.height(36.0);
fe.flip_y(true);
fe.hinting(true);

wchar_t *s = L"C++编程";
std::vector<agg::int8u> data;
vs_type vs;
agg::conv_curve<vs_type> ccvs(vs);
int x=20,y=100;
for(;*s;s++)
{
    if(!fe.prepare_glyph(*s)) continue;
    data.resize( fe.data_size() );
```

```

fe.write_glyph_to( &data[0] );
vs.init(&data[0], data.size(), x, y);
x += fe.advance_x();
y += fe.advance_y();
ras.add_path(ccvs);
agg::render_scanlines_aa_solid(ras,sl,renb,agg::rgba(0,0,1));
}

```

### 方式三、使用字体缓存管理器

每次都重新读字模是很费时的，比如前面的例子，"C++" 里的两个'+' 就读两次字模，效率可以想象。

一个好的办法是把已读出来的字模缓存起来，下次再遇到这个字时就不用从字体引擎里读取了，AGG 提供的 **font\_cache\_manager** 类就是 负责这项工作的。

头文件

```
#include "agg_font_cache_manager.h"
```

类型

```
template<class FontEngine> class font_cache_manager;
```

模板参数 FontEngine 指定管理器所用的字体引擎。另外构造参数也是 FontEngine。

成员方法

<pre>const glyph_cache* glyph(unsigned glyph_code);</pre>	<p>获得字模并缓存，glyph_cache 类的定义是：</p> <pre>struct glyph_cache {</pre>
---	---

	<pre>unsigned glyph_index;  int8u* data;  unsigned data_size;  glyph_data_type data_type;  rect_i bounds;  double advance_x;  double advance_y;  };</pre>
<pre>path_adaptor_type&amp; path_adaptor();</pre>	字体引擎的 path_adaptor_type 实例
<pre>gray8_adaptor_type&amp; gray8_adaptor();  gray8_scanline_type&amp; gray8_scanline();</pre>	字体引擎的 gray8_adaptor_type 实例以及对应的 Scanline
<pre>mono_adaptor_type&amp; mono_adaptor();  mono_scanline_type&amp; mono_scanline();</pre>	字体引擎的 mono_adaptor_type 实例以及对应的 Scanline
<pre>void init_embedded_adaptors(const glyph_cache* gl,      double x, double y,      double scale=1.0);</pre>	初始化上面的 adaptor 成员实例(与字体引擎的 ren_type 设置相关)



bool add_kerning(double* x, double* y);	调整坐标
---	------

示例代码 1 – 作为 Rasterizer 渲染：

显示效果

```
agg::font_engine_win32_tt_int16 font(dc);

agg::font_cache_manager<

agg::font_engine_win32_tt_int16

> font_manager(font);

font.height(72.0);

font.width(0);

font.italic(true);

font.flip_y(true);

font.hinting(true);

font.transform(agg::trans_affine_rotation(agg::deg2rad(4.0)));

font.create_font("宋体",agg::glyph_ren_agg_gray8);

double x=10, y=72; //起始位置

wchar_t *text = L"C++编程网";

// 画所有字符

for(*text;text++)

{

    //取字模
```

```

const agg::glyph_cache* glyph = font_manager.glyph(*text);

if(glyph)

{

    // 初始化 gray8_adaptor 实例

    font_manager.init_embedded_adaptors(glyph, x, y);

    agg::render_scanlines_aa_solid(font_manager.gray8_adaptor(),

    font_manager.gray8_scanline(),

    renb, agg::rgba8(0, 0, 0));

    // 前进

    x += glyph->advance_x;

    y += glyph->advance_y;

}

}

```

## C++编程网

示例代码 2 – 作为顶点源渲染：

```

typedef agg::font_engine_win32_tt_int16 fe_type;

fe_type font(GetDC(0));

typedef agg::font_cache_manager<fe_type> fcman_type;

fcman_type font_manager(font);

font.height(72.0);

```

```
font.width(0);

font.italic(true);

font.flip_y(true);

font.hinting(true);

font.transform(agg::trans_affine_rotation(agg::deg2rad(4.0)));

font.create_font("宋体",agg::glyph_ren_outline);

double x=10, y=72; //起始位置

wchar_t *text = L"C++编程网";

// 画所有字符

for(;*text;text++)

{

    const agg::glyph_cache* glyph = font_manager.glyph(*text);

    if(glyph)

    {

        // 准备*_adaptor

        font_manager.init_embedded_adaptors(glyph, x, y);

        // 先用 conv_curve

        typedef agg::conv_curve<

            fcman_type::path_adaptor_type

        > cc_pa_type;

        cc_pa_type ccpath(font_manager.path_adaptor());
```

```

// 画轮廓

typedef agg::conv_stroke<cc_pa_type> cs_cc_pa_type;

cs_cc_pa_type cscppath(ccpath);

agg::rasterizer_scanline_aa<> ras;

agg::scanline_u8 sl;

ras.add_path(cscppath);

agg::render_scanlines_aa_solid(ras, sl, renb, agg::rgba8(0, 0, 0));

// 前进

x += glyph->advance_x;

y += glyph->advance_y;

}

}

```

显示效果



## **AGG 成果**

作为本文的结尾，这里放上一个用 AGG 生成不规则文字窗体的代码。它综合了我们之前学到的 AGG 字体引擎、坐标转换、颜色渐变等几大模块。由于 AGG 的抗锯齿特性，使用生成的窗体看上去边缘过渡非常自然，几乎看不到“毛边”。

先放上最终生成的窗体的效果：



貌似比网页左上角的 logo 还要好看那么一点点-\_-

```
#define _WIN32_WINNT 0x0501

#include <windows.h>
#include <agg_array.h>
#include <agg_pixfmt_rgba.h>
#include <agg_scanline_u.h>
#include <agg_renderer_scanline.h>
#include <../font_win32_tt/agg_font_win32_tt.h>
#include <agg_font_cache_manager.h>
#include <agg_span_solid.h>
#include <agg_span_interpolator_linear.h>
#include <agg_span_gradient.h>
#include <agg_span_allocator.h>
#include <agg_conv_transform.h>
#include <agg_ellipse.h>
#include <agg_trans_single_path.h>

typedef agg::font_engine_win32_tt_int16 fe_type;
typedef agg::font_cache_manager<fe_type> fcman_type;
typedef agg::renderer_base<agg::pixfmt_bgra32> renb_type;

// 使用指定的顶点源和线段生成器输出文字
template<class VS, class SpanGenerator>
void AggDrawText(renb_type &renb,
    fcman_type &font_manager,
    VS &vs, SpanGenerator &span_gen,
    const wchar_t *txt)
{
    using namespace agg;

    span_allocator<rgba8> span_alloc;
    rasterizer_scanline_aa<> ras;
    scanline_u8 sl;
```

```

double x=0, y=0;
for(const wchar_t *p = txt; *p; p++)
{
    const glyph_cache* gc = font_manager.glyph(*p);
    if(gc)
    {
        font_manager.init_embedded_adaptors(gc, x, y);

        ras.add_path(vs);
        agg::render_scanlines_aa(ras, sl, renb, span_alloc, span_gen);

        x += gc->advance_x;
        y += gc->advance_y;
    }
}

// 向 renb 的指定位置和半径输出 http://www.cppprog.com ,有环绕效果
void DrawUrl(HDC dc, renb_type &renb,
    double ox, double oy, double rx, double ry)
{
    using namespace agg;
    //字体引擎
    fe_type font(dc);
    fcman_type font_manager(font);
    font.height(18.0);
    font.flip_y(true);
    font.hinting(true);
    if(!font.create_font("Comic Sans MS",agg::glyph_ren_outline)) return;
    //坐标转换管道
    typedef conv_curve<
        fcman_type::path_adaptor_type
    > cc_pa_type;
    cc_pa_type ccpath(font_manager.path_adaptor());

    typedef conv_transform<cc_pa_type,
        trans_single_path> ct_cc_pa_type;
    trans_single_path trans_path;
    ct_cc_pa_type ctpath(ccpath, trans_path);

```

```

ellipse ell(0,0,rx,ry);
trans_affine ellmtx;
conv_transform<ellipse> ctell(ell, ellmtx);
ellmtx.rotate(agg::pi);
ellmtx.translate(ox,oy);
trans_path.add_path(ctell);
// 线段生成器
span_solid<rgba8> ss;
ss.color(rgba(1,0,0));

AggDrawText(renb, font_manager, ctpath, ss, L"http://www.cppprog.com");
}
// 向 renb 的指定位置输出“C++编程”几个字，有镜象效果
void DrawName(HDC dc, renb_type &renb, double x, double y)
{
    using namespace agg;
    // 字体引擎
    fe_type font(dc);
    fcman_type font_manager(font);

    font.height(30.0);
    font.flip_y(true);
    font.hinting(true);
    if(!font.create_font("黑体",agg::glyph_ren_outline)) return;
    // 坐标转换管道
    typedef conv_curve<
        fcman_type::path_adaptor_type
    > cc_pa_type;
    cc_pa_type ccpath(font_manager.path_adaptor());

    typedef conv_transform<cc_pa_type> ct_cc_pa_type;
    trans_affine mtx;
    ct_cc_pa_type ctpath( ccpath, mtx );

    mtx.translate(50,50);
    //线段生成器
    span_solid<rgba8> ss;
    ss.color(rgba(0,0,0));

```

```

AggDrawText(renb, font_manager, ctpath, ss, L"C++编程");
// 改变坐标转换矩阵(镜像)

mtx.reset();
mtx.flip_y();
mtx.translate(50,60);

// 渐变线段生成器

typedef span_interpolator_linear<> interpolator_type;
trans_affine img_mtx;
interpolator_type ip(img_mtx);

typedef gradient_y gradientF_type;
gradientF_type grF;

typedef gradient_linear_color<rgba8> colorF_type;
colorF_type colorF(rgba(0,0,0), rgba(0,0,0,0));

typedef span_gradient<rgba8,
interpolator_type,
gradientF_type,
colorF_type> span_gen_type;
span_gen_type span_gen(ip,grF,colorF,30,80);

AggDrawText(renb, font_manager, ctpath, span_gen, L"C++编程");
}
// 调用 DrawUrl 和 DrawName 向 renb 输出文字
void DrawIt(HDC dc, renb_type &renb)
{
// 以透明色填充
renb.clear(rgba(0,0,0,0));
// 输出文字
DrawUrl(dc, renb, 100, 50, 80, 40);
DrawName(dc, renb, 50, 50);
}
// 使用 AGG 处理图片后与 hwnd 关联
void SetLayoutWin(HWND hwnd)
{
// 起始位置和窗体大小
POINT ptWinPos = {500,200};
SIZE sizeWindow = {200, 100};

```



```

// 建立 DIB
BITMAPINFO bmp_info;
::ZeroMemory(&bmp_info, sizeof(bmp_info));
bmp_info.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
bmp_info.bmiHeader.biWidth = sizeWindow.cx;
bmp_info.bmiHeader.biHeight = sizeWindow.cy;
bmp_info.bmiHeader.biPlanes = 1;
bmp_info.bmiHeader.biBitCount = 32;
bmp_info.bmiHeader.biCompression = BI_RGB;

HDC hdcTemp = GetDC(0);
HDC mem_dc = ::CreateCompatibleDC(hdcTemp);
ReleaseDC(0, hdcTemp);

void* buf = NULL;
HBITMAP bmp = ::CreateDIBSection(
    mem_dc,
    &bmp_info,
    DIB_RGB_COLORS,
    &buf,
    0, 0
);

// 把 bmp 与 mem_dc 关联，这样 AGG 就可以和原生 GDI 一起工作了
HBITMAP temp = (HBITMAP)::SelectObject(mem_dc, bmp);
{
    // AGG 处理
    agg::rendering_buffer rbuf(
        (unsigned char*)buf,
        sizeWindow.cx, sizeWindow.cy,
        -sizeWindow.cx*4);
    agg::pixfmt_bgra32 pixf(rbuf);
    renb_type renb(pixf);
    DrawIt(mem_dc, renb);
}

// 把画好的 mem_dc 与 hwnd 关联到一起
BLENDFUNCTION m_Blend={AC_SRC_OVER,0,255,AC_SRC_ALPHA};
POINT ptSrc = {0, 0};
BOOL bRet = UpdateLayeredWindow(hwnd, 0, &ptWinPos,

```

```

        &sizeWindow, mem_dc, &ptSrc,
        0, &m_Blend, ULW_ALPHA);
// 回收
::DeleteObject(bmp);
::DeleteDC(mem_dc);
}

// Windows 消息处理
LRESULT CALLBACK WndProc (HWND hwnd, UINT umsg, WPARAM wParam,
        LPARAM lParam)
{
    switch (umsg)
    {
        case WM_CLOSE:
            DestroyWindow (hwnd);
            return 0;
        case WM_DESTROY:
            PostQuitMessage (0);
            return 0;
        case WM_NCHITTEST:
            return HTCAPTION;
    }
    return DefWindowProc (hwnd, umsg, wParam, lParam);
}

int APIENTRY WinMain(HINSTANCE hInstance,
        HINSTANCE hPrevInstance,
        LPTSTR lpCmdLine,
        int nCmdShow)
{
    WNDCLASS wc={
        0,WndProc,
        0,0,
        hInstance,
        NULL,LoadCursor(NULL, IDC_ARROW),
        (HBRUSH)(COLOR_WINDOW+1),
        0,"AGGWIN"
    };
    ::RegisterClass(&wc);

```

```
HWND hWnd = ::CreateWindowEx(WS_EX_LAYERED,"AGGWIN", NULL,  
WS_OVERLAPPEDWINDOW,  
    CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);  
  
if (!hWnd) return -1;  
SetLayoutWin(hWnd);  
::ShowWindow(hWnd, nCmdShow);  
  
// 主消息循环:  
MSG msg;  
while (GetMessage(&msg, NULL, 0, 0))  
{  
    TranslateMessage(&msg);  
    DispatchMessage(&msg);  
}  
  
return (int) msg.wParam;  
}
```