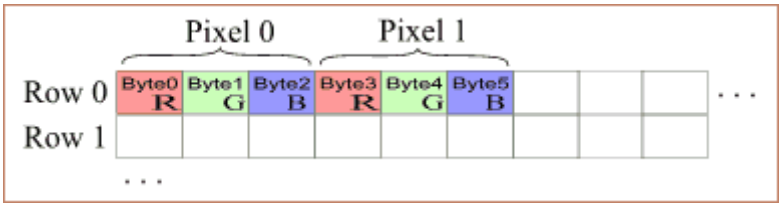
AGG 中文手册

1 渲染内存 Rendering Buffer

我们先从这里开始：在内存中开辟一块存储区，然后将它的内容以最简单的光栅格式写到文件中，也就是 PPM（Portable Pixel Map）格式。虽然 Windows 对这种格式并没有原生的支持，但很多图像浏览器和转换器都能使用这种格式，比如 IrfanView（www.irfanview.com）。所有 AGG 的控制台例子都使用了 P6 256 格式，也就是 RGB，每个字节代码一个颜色。现在假设我们将在下图所示的 RGB-buffer 内存区中工作：



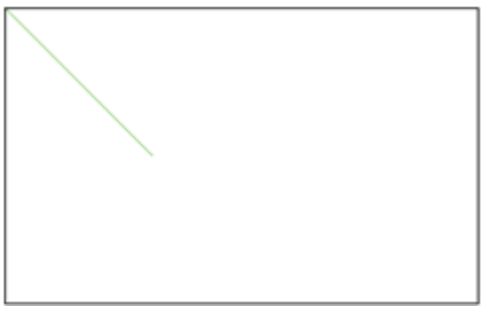
1.1 第一个简单例子

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

enum
{
    frame_width = 320,
    frame_height = 200
};

// Writing the buffer to a .PPM file, assuming it has
// RGB-structure, one byte per color component
//-----------------------------------------------
bool write_ppm(const unsigned char* buf,
               unsigned width,
               unsigned height,
               const char* file_name)
{
```

```
        FILE* fd = fopen(file_name, "wb");
        if(fd)
        {
            fprintf(fd, "P6 %d %d 255 ", width, height);
            fwrite(buf, 1, width * height * 3, fd);
            fclose(fd);
            return true;
        }
        return false;
}

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

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

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

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

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

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

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

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

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

在这个例子中，你甚至不需要链接任何的 AGG 的代码文件，你只需要在你的编译器命令行中设置好 AGG 的包含路径就行了。编译并运行它，你会看到现图所示的结果：



这个例子中几乎所有东西都“手工打制”的，使用的唯一一个现成的类是 rendering_buffer。这个类本身并不知道关于内存中像素格式的任何信息，它只是保存了一个数组，数组中的元素分别指向每行（像

素的开头）。为申请和释放这块存储区是使用者的责任，你可以使用任何可行的方式来申请和释放内存，比如使用系统提供的 API 函数，或是简单的用内存分配器（译注：应该是 new、delete、malloc、free 等），甚至是直接使用一个静态数组。在上面这个例子中，因为每个像素要占用 3 个字节，所以我们申请了 width*height*3 字节的内存，在实际内存中是不存在“行”这种布局方式的，但这样做可以提高程序的性能，而且有时候在使用 API 的时候需要。

1.2 Class rendering_buffer

包含文件： agg_rendering_buffer.h

rendering_buffer 这个类保存了指向每一行像素的指针，基本上这个类做的事就是这些了。看起来好像不是什么了不起的事，不过我们还是继续分析下去。这个类的接口和功能都很简单，它只是模板类 row_ptr_cache 的一个 typedef 而已：

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

row_prt_cache 这个类的接口的功能如下：

```
template<class T> class row_ptr_cache
{
public:
    row_ptr_cache();
    row_ptr_cache(T* buf, unsigned width, unsigned height, int stride);
    void attach(T* buf, unsigned width, unsigned height, int stride);
    T* buf();
    const T* buf()    const;
    unsigned width()  const;
    unsigned height() const;
    int      stride() const;
```

```
        unsigned stride_abs() const;
        T* row_ptr(int, int y, unsigned);
        T* row_ptr(int y);
        const T* row_ptr(int y) const;
        row_data row    (int y) const;
        T const* const* rows() const;
        template<class RenBuf> void copy_from(const RenBuf& src);
        void clear(T value)
    };
```

这个类的实现里没有使用断言或是验证的措施，所以，使用者有责任在用这个类对象时正确地将它初始化到实际的内存块中，这可以在构造函数中完成，也可以使用 attach() 函数。它们的参数解释如下：

- buf — 指向内存块的指针。
- width — 以像素为单位表示的图像宽度。rendering buffer（渲染内存区）并不知道像素格式和每个像素在内存中大小等信息。这个值会直接存储在 m_width 这个成员变量中，使用 width() 函数就可以获取它的值。
- height — 以像素为单位表示的图像高度（同样也是行数）
- stride — Stride（大步幅，-_-；不知道怎么翻了……），也就是用类型 T 来度量的一行的长度。 rendering_buffer 是一个 typedef，也就是 row_prt_cache<int8u>，所以这个值是以字节数来算的。Stride 决定内存中每一行的实现长度。如果这个值是负的，那么 Y 轴的方向就是反过来的。也就是说 Y 等 0 的点是是内在块的最后一行的。Y == height - 1 则是第一行。stride 的绝对值也很重要，因为它让你可以方便地操作内存中的“行”（就像 windows 中的 BMP）。另外，这个参数允许使用者可以像操作整个内存区块一样，操作其中任意一个“矩形”内存区。

attach()函数会改变缓冲区或是它的参数，它自己会为“行指针”数组重新分配内存，所以你可以在任何时候调用它。当（且仅当）新的 height 值比之前使用过的最大的 height 值还要大时，它才会重新申请内存。

构造的这个对象的开销仅仅是初始化它的成员变量（设置为 0），attach()的开销则是分配 sizeof(ptr)*height 个字节的内存，然后将这些指针指向对应的“行”。

最常使用的函数是 row_prt(y)，这个函数只是简单地返回指向第 y 函数指针，这个指针指向的位置已经考虑到了 Y 轴的方向了。

注意：

渲染内存区（rendering buffer）并不管任何裁减或是边界检查的事，这是更高层的类的责任。

buf(), width(), height(), stride(), stride_abs() 这些函数的意义显而易见，就不解释了。

copy_from()函数会将其它内存的内容拷贝至“本”内存中。这个函数是安全的，如果(两者的)width 和 height 值不相同，那它会尽可能拷贝大一些的区域。一般来讲都用于拷贝相同大小区域。

1.3 Two Modifications of the Example

首先，在创建 rendering buffer 的对象时将 stride 取负值：

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

那么结果将变成这样：

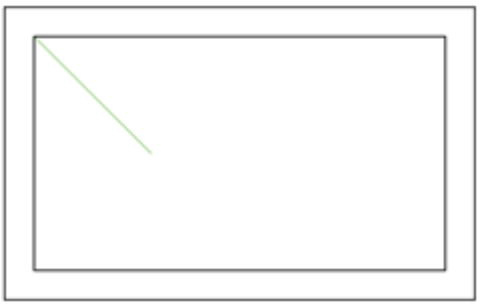
然后，我们试下将 rendering buffer 附着(attach)到被分配的内存区的某部分。这个修改会使得 rendering buffer 两次附着在同一块内存区上，第一次是整个被分配的内存区域，第二次是其中的一部分：

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

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

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

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

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

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

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

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

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

    delete [] buffer;
```

```
    return 0;
}
```

最后描画出来的结果是这样：



最后一处修改是：

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

运行结果如下：

在最后的一个例子里，我们只是使 stride 取了负值，而指针则和上个例子一样，仍然指向内存区的起启处。

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

2 Pixel Format Renderers

首先，我们创建一个更“文明”（译注：显得更高级一点）的例子：

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

enum
{
    frame_width = 320,
    frame_height = 200
};

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

// Draw a black frame around the rendering buffer
```

```
//----------------------------------------------------

template<class Ren>

void draw_black_frame(Ren& ren)

{

    unsigned i;

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

    for(i = 0; i < ren.height(); ++i)

    {

        ren.copy_pixel(0,                i, c);

        ren.copy_pixel(ren.width() - 1, i, c);

    }

    for(i = 0; i < ren.width(); ++i)

    {

        ren.copy_pixel(i, 0,                 c);

        ren.copy_pixel(i, ren.height() - 1, c);

    }

}




int main()

{

    //--------------------

    // Allocate the buffer.

    // Clear the buffer, for now "manually"

    // Create the rendering buffer object

    // Create the Pixel Format renderer
```

```
// Do something simple, draw a diagonal line
// Write the buffer to agg_test.ppm
// Free memory

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

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

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

agg::pixfmt_rgb24 pixf(rbuf);

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

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

delete [] buffer;
return 0;
```

```
}
```

这个例子看起来和前面的没什么不一样的，但其实他们差别很大，看看这个声明：

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

这里我们创建了一个底层的像素渲染对象（pixel rendering object）并将它附着到渲染内存区（rendering buffer）上，它是这样定义的：

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

类模板 pixel_formats_rgb24 掌握了内存中具体的像素格式信息。唯一的模板参数可以是 order_rgb24 或是 order_rgb23，它们定义了颜色字节(color channels)的顺序。

与 rendering buffer 不同的是，这些类使用整型的像素坐标进行操作，因为它们知道怎么计算对于特定点 X 的偏移。你可能会说，如果在 rendering buffer 中保存像素的宽度值的话会更容易，但是在实践中会有很多限制。别忘了，像素宽度可能比一个字节还小，比如在打印机渲染高解析度的 B&W 图像的时候就是这样。因此，我们需要将这个功能分离出来，rendering_buffer 这个类就用于加速对“行”的访问，而 pixel format renderers 就负责如何解析“行”是什么。

现在，AGG 里下面这些文件实现了各种不同的像素格式：

- agg_pixfmt_gray8.h：每个字节表示一个像素灰度。这种像素格式允许你将它和 rgb24 或是 rgb23 中的某个颜色成分放在一起工作。它有两个模板参数：Step 和 Offset，为方便起见，还有下面这些 typedef ：
    - typedef pixfmt_gray8_base<1, 0> pixfmt_gray8;
    - typedef pixfmt_gray8_base<3, 0> pixfmt_gray8_rgb24r;
    - typedef pixfmt_gray8_base<3, 1> pixfmt_gray8_rgb24g;
    - typedef pixfmt_gray8_base<3, 2> pixfmt_gray8_rgb24b;
    - typedef pixfmt_gray8_base<3, 2> pixfmt_gray8_bgr24r;
    - typedef pixfmt_gray8_base<3, 1> pixfmt_gray8_bgr24g;
    - typedef pixfmt_gray8_base<3, 0> pixfmt_gray8_bgr24b;
    - typedef pixfmt_gray8_base<4, 0> pixfmt_gray8_rgba32r;

- typedef pixfmt_gray8_base<4, 1> pixfmt_gray8_rgba32g;
- typedef pixfmt_gray8_base<4, 2> pixfmt_gray8_rgba32b;
- typedef pixfmt_gray8_base<4, 3> pixfmt_gray8_rgba32a;
- typedef pixfmt_gray8_base<4, 1> pixfmt_gray8_argb32r;
- typedef pixfmt_gray8_base<4, 2> pixfmt_gray8_argb32g;
- typedef pixfmt_gray8_base<4, 3> pixfmt_gray8_argb32b;
- typedef pixfmt_gray8_base<4, 0> pixfmt_gray8_argb32a;
- typedef pixfmt_gray8_base<4, 2> pixfmt_gray8_bgra32r;
- typedef pixfmt_gray8_base<4, 1> pixfmt_gray8_bgra32g;
- typedef pixfmt_gray8_base<4, 0> pixfmt_gray8_bgra32b;
- typedef pixfmt_gray8_base<4, 3> pixfmt_gray8_bgra32a;
- typedef pixfmt_gray8_base<4, 3> pixfmt_gray8_abgr32r;
- typedef pixfmt_gray8_base<4, 2> pixfmt_gray8_abgr32g;
- typedef pixfmt_gray8_base<4, 1> pixfmt_gray8_abgr32b;
- typedef pixfmt_gray8_base<4, 0> pixfmt_gray8_abgr32a;

- agg_pixfmt_rgb24.h, 每个像素占 3 个字节，有 RGB 或是 BGR 两种颜色顺序。定义了下面这些像素类型：
  - typedef pixel_formats_rgb24<order_rgb24> pixfmt_rgb24;
  - typedef pixel_formats_rgb24<order_bgr24> pixfmt_bgr24;
- agg_pixfmt_rgb555.h, 每个像素占 15 bits。 每 5 个 bits 代表一种颜色成分，最低位没用到。
- agg_pixfmt_rgb565.h, 每个像素占 16 bits。红色占 5 bits, 绿色占 6 bits, 蓝色 5 bits。
- agg_pixfmt_rgba32.h, 每个像素占 4 个字节, RGB 再加 Alpha 通道值。下面有不同的颜色顺序类型:
  - typedef pixel_formats_rgba32<order_rgba32> pixfmt_rgba32;
  - typedef pixel_formats_rgba32<order_argb32> pixfmt_argb32;
  - typedef pixel_formats_rgba32<order_abgr32> pixfmt_abgr32;
  - typedef pixel_formats_rgba32<order_bgra32> pixfmt_bgra32;

像素格式的类定义了它们原始的颜色空间和颜色类型，像下面这样：

```
typedef rgba8 color_type;
```

对于 pixfmt_gray8_nnn，这些都是 gray8，这种机制允许你写自己的像素和颜色格式，举个例子，HSV、CMYK 等等，AGG 的其它部分可以完全无误地和你自己定义的新的像素格式一起工作。

注意

区分清楚 color type 和 buffer 代表的原始的颜色空间是非常重要的。比如说，你可以假设你正在使用 CMYK 进行工作，但使用的是 RGB 的 buffer （你只需要写一个简单的转换函数，就可以从 CMYK 结构 中创建 rgba8 的对象）。但这种转换只是一个近似，可能会因此出现颜色上的失真，因为有些 CMYK 中的颜色无法用 RGB 来表示，反之亦然。如果想完全使用某种颜色空间的表现能力，你可能要写一个为这种颜色空间写一个可以避免中间转换的 pixel format renderer 。

2.1 Creation

重要!

像素格式相关的类并不进行任何的裁剪操作，也就是说直接使用这些类进行工作一般来说不太安全。裁剪是上层类的功能。采用这样设计的理由很简单：要让用户设计自定义的像素格式类越简单越好。像素格式可能会五花八门，但裁剪操作的代码一般都没有什么区别。

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

像素格式渲染器（pixecl format renderers）的构造函数需要一已经创建并良好初始化的 rendering_buffer 对象的引用。这个构建工作的开销很小，基本上只是初始化一个指针。

2.2 Member Functions

像素格式渲染器（pixecl format renderers）必须要实现以下这些接口：

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

返回内存区的宽和高（以像素数来衡量）

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

返回(x,y)坐标处的像素的颜色

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

将带颜色的像素拷入缓存区中。如果是本身 RGB 像素格式，那么它就不考虑 rgba8 拷贝源中的存在的 alpha 通道。如果本身是 RGBA，那么它就简单地把所有值都拷贝过来，包括 R、G、B，以及 alpha 通道值。

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

这个函数将带颜色信息的像素 c 与缓存区(x,y)处的像素进行混合(blending)。现在我们来解释一下“混合”的概念。混合(blending) 是抗锯齿(anti-aliasing)的关键特性。在 RGBA 的颜色空间中，我们使用 rgba8 结构体来代表颜色。这个结构体有一个数据成员 int8u a ，它就是 alpha 通道。不过，在这个函数里，我们还看到一个参数 cover ，表示像素的覆盖值大小，比如，这个像素被多边形所“覆盖”的部分的大小（译注：这涉及到亚像素精度，因为一个像素可以分为 256*256 份，所以这个像素并不一定全部被“覆盖”，详细可参考 AGG 对于亚像素的说明）。其实你可以把它看成是另一个 alpha（或者应该叫它 Beta? :)）。这么做有两个原因，首先，颜色类型（color type）不一定非要包含 alpha 值）。就算颜色类型带有 alpha 值，它的类型也不一定非要与抗锯齿算法中使用的颜色类型一致。假设你现在使用的是 "Hi-End" RGBA 颜色空间，这种颜色空间使用 4 个取值范围是[0,1]浮点型来表示，alpha 通道值也使用浮点数————对于这种情况来说，混合时使用一个 byte 实在太少了，但在去锯齿时却非常够用。所以，cover 值就是为去锯齿而使用的一个统一的备用 alpha 值。在大部分情况来说，用 cover_type 来定义它，但在光栅化处理器（rasterizers）中是直接显示地使用 int8u 类型的。这是故意这么定义的，因为如果需要加强 cover_type 的能力时，会使得所有已经存在的像素格式光栅化处理器（pixel format rasterizres）变得与 cover_type 不兼容。它们确实是不兼容的，在进行颜色混合时，如果中间值使用 32-bit 值来暂存的话，那么最大只能使用 8-bit 的覆盖值(coverage value）和 8-bit 的 alpha 值(alpha) 。如果使用 16-bit 的值的话，就要用 64-bit 的中间值暂存，这对于 32-bit 的平台来说会有非常昂贵的开销。

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

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

使用某种颜色描画一条水平或是垂直的线。

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

采用混合颜色的模式描画一带某种颜色的水平（或垂直线）线。之所以要分开 copy 和 blend 两个版本，是因为考虑到效率问题。虽然可以使用一个 if/else （其实在 blend 版的描画函数中就有）来区分，但对于某些场合，比如要描画很多小型标识（markers）时，这会很影响效率，这种场景在不同的散点图描画程序（scatter plot applicatioin）中常常遇到。

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

混合描画一条水平或是垂直的 solid-color 的 span， Span 与 hline 和 vline 几乎是一样的，但它拥有一个存有 coverage value 的数组。这两个函数在渲染实心的去锯齿多边形时会用到。

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

混合描画水平或是垂直的颜色 span ，这两个函数用于不同的 span 产生器中，比如说 gradient，image，patterns，Gouraud interpolation 等等。函数接受一个颜色数组参数，这个颜色数组必须与所使用的像素格式兼容。比如说，所有 AGG 中已经有的 RGB 像素格式都与 rgb8 类型是兼容的。 covers 参数是一个 coverage value 的数组，这与 blend_solid_hspan 中的是一样的。这是参数可选，可以设置为 0 。

下面这个例子是描画阳光的光谱。rgba 类包含有 4 个浮点数的颜色部分（包括 alpha），这个类有一个静态函数 from_wavelength ，以及相应的构造函数。rgba8 可以用 rgba 来构造（在 AGG 中这是一个常见的策略，也就是任何的颜色类型都可以用 rgba 来构造）。

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

enum
{
    frame_width = 320,
    frame_height = 200
};

// [...write_ppm is skipped...]
int main()
{
    //--------------------
    // Allocate the buffer.
    // Clear the buffer, for now "manually"
    // Create the rendering buffer object
    // Create the Pixel Format renderer
    // Create one line (span) of type rgba8.
    // Fill the buffer using blend_color_span
    // Write the buffer to agg_test.ppm
    // Free memory
    unsigned char* buffer = new unsigned char[frame_width * frame_height * 3];
```

```
    memset(buffer, 255, frame_width * frame_height * 3);
    agg::rendering_buffer rbuf(buffer,
        frame_width,
        frame_height,
        frame_width * 3);
    agg::pixfmt_rgb24 pixf(rbuf);
    agg::rgba8 span[frame_width];
    unsigned i;
    for(i = 0; i < frame_width; ++i)
    {
        agg::rgba c(380.0 + 400.0 * i / frame_width, 0.8);
        span[i] = agg::rgba8(c);
    }
    for(i = 0; i < frame_height; ++i)
    {
        pixf.blend_color_hspan(0, i, frame_width, span, 0);
    }
    write_ppm(buffer, frame_width, frame_height, "agg_test.ppm");
    delete [] buffer;
    return 0;
}
```

运行结果如下：

2.3 Alpha-Mask Adaptor

Alpha-Mask 是一个分离出来的缓存区，通常用于在底层实现任意形状的裁剪。有一个特制的适配器类，可以将所有对像素格式渲染器（pixel format renderes）的调用先丢给 alpha-mask 过滤器。alpha-mask 一般是一个灰度缓存区（每像素一个字节），大小与主渲染缓存区（main rendering buffer）是一样的。在 alpha-mask 中的每个像素就是对应着主渲染缓存区相应像素的一个额外的覆盖值(coverage value)。copy_hline()之类没有酸辣值作为参数的函数会将调用转向相应的有覆盖值的函数。比如，copy_hline()会从 alpha mask 缓存区中取出水平 span ，再用它来调用 blend_solid_hspan（） 函数。

包含文件：

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

下面是一个例子，它展示了如何声明一个带有 alpha-mask 适配器的像素格式渲染器。

```
#include "agg_pixfmt_rgb24.h"
#include "agg_pixfmt_amask_adaptor.h"
#include "agg_alpha_mask_u8.h"
//. . .
// Allocate the alpha-mask buffer, create the rendering buffer object
// and create the alpha-mask object.
//---------------------------------
agg::int8u* amask_buf = new agg::int8u[frame_width * frame_height];
```

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

注意在这里我们用的是 amask_no_clip_gray8 ，它不带区域裁剪功能。因为我们使用的 alpha-mask 缓存区和主缓存区的大小是一样的，所以只要主缓存区没有非法访问的话，alpha-mask 缓存区也不会存在非法访问。裁剪在高层中实现，如果你使用的 alpha-mask 缓存区比主缓存区小的话，你必须使用 alpha_mask_gray8 这个类来代替。

下面是一个完整的例子：

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

enum
{
    frame_width = 320,
```

```
    frame_height = 200
};


// [...write_ppm is skipped...]
int main()
{
    // Allocate the main rendering buffer and clear it, for now "manually",
    // and create the rendering_buffer object and the pixel format renderer
    //-------------------------------
    agg::int8u* buffer = new agg::int8u[frame_width * frame_height * 3];
    memset(buffer, 255, frame_width * frame_height * 3);
    agg::rendering_buffer rbuf(buffer,
        frame_width,
        frame_height,
        frame_width * 3);
    agg::pixfmt_rgb24 pixf(rbuf);
    // Allocate the alpha-mask buffer, create the rendering buffer object
    // and create the alpha-mask object.
    //-------------------------------
    agg::int8u* amask_buf = new agg::int8u[frame_width * frame_height];
    agg::rendering_buffer amask_rbuf(amask_buf,
        frame_width,
        frame_height,
        frame_width);
    agg::amask_no_clip_gray8 amask(amask_rbuf);
    // Create the alpha-mask adaptor attached to the alpha-mask object
```

```
// and the pixel format renderer
agg::pixfmt_amask_adaptor<agg::pixfmt_rgb24,
    agg::amask_no_clip_gray8> pixf_amask(pixf, amask);
// Draw something in the alpha-mask buffer.
// In this case we fill the buffer with a simple verical gradient
unsigned i;
for(i = 0; i < frame_height; ++i)
{
    unsigned val = 255 * i / frame_height;
    memset(amask_rbuf.row_ptr(i), val, frame_width);
}
// Draw the spectrum, write a .ppm and free memory
//----------------------
agg::rgba8 span[frame_width];
for(i = 0; i < frame_width; ++i)
{
    agg::rgba c(380.0 + 400.0 * i / frame_width, 0.8);
    span[i] = agg::rgba8(c);
}
for(i = 0; i < frame_height; ++i)
{
    pixf_amask.blend_color_hspan(0, i, frame_width, span, 0);
}
write_ppm(buffer, frame_width, frame_height, "agg_test.ppm");
delete [] amask_buf;
delete [] buffer;
```

```
    return 0;
}
```

这是运行结果：

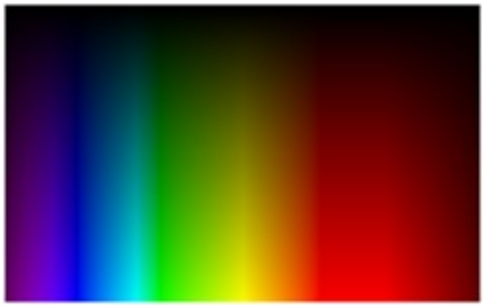
你看到了，我们是用白色来清空主缓存的，如果我们将：

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

修改成：

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

那么结果就像下面这样：



换句话说，alpha-mask 是这样工作的：它是一个分离出来的 alpha 通道，用于混合渲染基本的描画物。因为它包含的是 8-bit 的值，所以你可以将描画裁剪成任意的形状，而且这种裁剪可以有很非常好的去锯齿效果。

3 Basic Renderers

有两种基础渲染器，renderer_base 和 renderer_mclip 它们的功能几乎是一样的。主要使用的是前者(renderer_base)，它进行低层次的裁剪处理。通用的裁剪处理(clipping)是一个复杂的任务。在 AGG 中，至少可以有两层的裁剪，底层（像素级）的，和高层的（向量级）。这两个类可以进行像素级别的裁剪，用以防止对缓存区的越界访问。 renderer_mclip 类可以支持多个矩形区域的裁剪区域，但它的性能和裁剪区域的数量有关。

renderer_base 和 renderer_mclip 都是模板类，它们的模板参数就是像素格式渲染器（pixel format renderer)。

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

3.1 Creation

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

两个类的构造函数都可以接受像素格式渲染器（pixel format renderer)对象作为参数。renderer_mclip 在内部使用 renderer_base<PixelFormat>，用来进行单矩形区域的裁剪。注意，你也可以使用 pixfmt_amask_adaptor 作为参数 PixelFormat。

构造的开销是非常小的，它只是初始化各个成员变量。不过，如果你添加新的裁剪区域，那么 renderer_mclip 需要申请新的内存，在析构时也会有相应的释放动作。它会用 pod_deque 类来完成内存块的申请，并且它不会（提前）释放不需要的内存，当你重新设置裁剪区域的时候，它会重用原来申请的内存区。 AGG 中广泛使用了这种技术，这可以避免产生过多的内存碎片。

3.2 Member Functions

```
const pixfmt_type& ren() const;

pixfmt_type& ren();
```

返回指向像素渲染器（pixel format renderer)的引用。

```
unsigned width()  const;

unsigned height() const;
```

返回渲染缓存区（rendering buffer）的高度和宽度。

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

这个函数会重设裁剪区域，如果 visibility 值设置为 true ，那么裁剪区域会被设置为 (0, 0, width()-1, height()-1)，如果设置为 false，会设置一个不可见的区域，比如(1,1,0,0)。对于 renderer_mclip 的这个函数来说，它会移除之前添加的所有剪裁区域。

重要!

如果你将另一块内存区附着到 rendering_buffer 上的话（rendering_buffer 是连接着这个（basic_renderer)，你必须调用一次 reset_clipping() ，否则的话，原来的裁剪区域会变得无效。 因为这个内存改变动作做完之后并不会，rendering buffer 不会给 renderers 任何反馈。也就是说，renderer_base 和 renderer_mclip 对于 rendering buffer 的变动一无所知（译注：renderer_base 是用 pixel format render 来构造的，而 pixel format render 又是由 rendering buffer 来支撑，所以

它们之间就有这样的关系，具体可见前文）。如果有这样的机制，可以在它们之间传递消息或是使用委托，那么这种机制可能成为 overkill （译注：不知道怎么理解这个 overkill）。

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

这个函数用于设置新的裁剪区域(clipping box)。只有 renderer_base 才有这个函数。裁剪区域包含了边界位置，所以最大的裁剪区域是 (0, 0, width()-1, height()-1) 。裁剪区域在（重新）设置之前，会被设置为最大值。所以，就算你设置新区域的比 (0, 0, width()-1, height()-1) 还要大，也是安全的（译注：因为不会被采用）

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

添加一个新的裁剪区域。只有 renderer_mclip 才有这个函数。你可以添加任何个新的矩形区域，但他们之间不能有重叠的部分。如果有重叠的区域的话，有些元素就会被描画两次或两次以上。被添加的新区域会在加入之时被 (0, 0, width()-1, height()-1) 这个区域预先裁剪掉，这主要是有效率上的考量。这也意味着调用 reset_clipping(false) 没有任何意义，因为所有被添加的新区域都会用一个不可见区域内先裁剪掉，因此也就不会被添加。（译注：原句如下：t also means that calling reset_clipping(false) for the renderer_mclip doesn't make any sense because all adding regions will be clipped by an invisible area and will not be actually added.对这个我不能理解它的意思，后面看了代码再来修正吧）。可见区域包含裁剪区域的边界，也就是说，调用 add_clip_box(100,100,100,100) 会添加一个 1 像素的裁剪区域。

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

只有 renderer_base 有这个函数，它用来给 rendering buffer 大小设置一个新的裁剪区域。这个函数不安全，主要是 renderer_mclip 类在使用它，用来在不同的子区域之间快速的切换。这个函数的目的就是为了避免额外的开销。

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

检查 (x,y) 这个点是不是在裁剪区域内。只有 renderer_base 才有这个函数。

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

这两个函数用于枚举渲染器所有的裁剪区域。对于 renderer_base 类来说，它是返回的是空的，next_clip_box() 始终返回 false。

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

以一个矩形的形式可是以四个独立的整数值形式返回裁剪区域。 renderer_mclip 的这个函数始终返回 (0, 0, width()-1, height()-1)。

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

以一个矩形的形式可是以四个独立的整数值形式返回裁剪区域的边界。对于 renderer_base ，这个函数返回的值于上面那组函数是一样的。对于 renderer_mclip 来说，它们返回的边界是由所有被添加的矩形合计得出的。

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

用 c 这个颜色来清除缓存区内的所有区域（不考虑裁剪区域）。

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

设置一个像素的颜色（考虑裁剪区域 clipping)。

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

混合描画一个像素。它的行为与 pixel format renderer 的对应函数是一样的。

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

获取指定坐标（x,y)的颜色值，如果这个点在裁剪区域外，这个函数返回 color_type::no_color(). 对于 rgba8 来说，就是 (0,0,0,0).

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

描画（拷贝）或是混合渲染水平或垂直的像素线。行为与 pixel format renders 的相应函数一样。但在这里，使用的是线的起始点和终点坐标，而不是 pixfmt 中的(x, y, length)。

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

描画（拷贝）或是混合渲染一个矩形区。

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

混合渲染一个水平或是垂直的纯色（solid-color）的 span。这些在渲染实多边形时会用到。

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

混合渲染一个水平或是垂直的（? vertical -color）的 span。这个函数与不同的 span 生成器一起使用，比如 gradients, images, patterns, Gouraud interpolation 等等。函数接受一个颜色数组，颜色的类型必须与正在使用 pixel format 兼容。

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

与上面的函数是一样的，但不考虑裁剪区域。这两个函数用到 scanline renderers 中。分离出这两个函数的原因也是为了效率,scanline 由很多的 spans 组合而成,在进行区域裁剪时,拥有整个 scanline 的信息会比逐个的裁剪每个 span 来得更有效率一些，对于 renderer_mclip 这个类尤其如此。

```
void copy_from(const rendering_buffer& from,
               const rect* rc=0,
```

```
                int x_to=0,
                int y_to=0);
```

将源缓存区的内容拷入本缓存区中（考虑区域裁剪）。它假设两块缓存区的像素格式是一样的。rc 是一个可选项，它指示的是源缓存区内的一个矩形，x_to 、 y_to ———— rc->x1, rc->y1 坐标值映射到目标缓存区。

3.3 A common example

在使用 rendering buffer 和底层 renderers 时，下面的代码是非常常用的。

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


unsigned char* buffer = new unsigned char[frame_width * frame_height * bytes_per_pixel];
agg::rendering_buffer rbuf(buffer,
                           frame_width,
                           frame_height,
                           frame_width * bytes_per_pixel);
pixfmt_type pixf(rbuf);
renbase_type rbase(pixf);
rbase.clear(clear_color);
```

```
    //. . .
```

在最后，我们使用 clear()函数，以某种颜色清除了缓存区中，而没有“手动地”使用 memset() :-)。同时要注意，与例子所不同的是，stride 值并不是必须要和 frame_width * bytes_per_pixel 值一致。很多时候应用时会有一些对齐上的要求，比如，对于 windows 中的位图（bitmap）来说，这个值必须是 4 的倍数。

4 Primitives and Markers Renderers

AGG 中也加入了一些基础描画对象（primitives）和标记（marker）的渲染器。这个机制可以让你很快地描画一些常见的、带锯齿的对象，比如直线，矩形，椭圆等。标记（marker)可以画出一些在散点图（scatter plots）中常见的形状。如果你不打算用它们，那么你可以跳过这一节的内容。

4.1 Primitives Renderer

头文件: agg_renderer_primitives.h

4.1.1 Declaration

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

这里的 BaseRenderer 可以用 renderer_base 或是 renderer_mclip.

4.1.2 Creation

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

创建的开销非常小，只是初始化指向 base renderer 对象的指针、两个颜色信息、以及初始化坐标值，这些坐标值会在 move_to 和 line_to 函数中用到。

Member functions

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

将一个 double 型的坐标值转换成亚像素精度的 int 型。它做的事就是将 double 值乘以 256 ，然后返回它的整型部分。

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

获取或是设置 fill 或是直线的颜色。颜色可以带有 alpha 值并且这个值会生效，这样的话，基础画面对象（primitives）就可以进行 alpha 混合渲染。

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

描画一个矩形，但不填充线的颜色。使用的是正规的像素坐标，边界的宽度始终是 1 像素，不能修改。

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

描画一个不带边界的实心矩形，填充线的颜色。使用的是像素坐标。

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

描画一个带边界的实心矩形。使用了颜色线(colors line)和填充(fill?)。

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

void solid_ellipse(int x, int y, int rx, int ry);

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

描画椭圆，空心的、实心的或是实心带边界的。坐标是整数值，以像素数来衡量。rx 和 ry 是像素数衡量的椭圆半径。

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

使用亚像素精度描画一条 bresenham 光栅线。坐标的格式是 24.8，即，1/256 的像素精度。 last 决定是否描画最后一个像素，这对于使用 alpha 混合渲染来描画 consecutive 线段会非常重要。不应该有像素被多次描画。

```
void move_to(int x, int y);

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

line()版本。

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

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

返回 base renderer 对象的引用。

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

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

返回附着于 base renderer 的 rendering buffer 的引用。

4.2 Marker Renderer

头文件: agg_renderer_markers.h

marker renderer 可以描画或是 alpha 混合沉浸下面的这些图形:

```
enum marker_e

{

    marker_square,

    marker_diamond,

    marker_circle,

    marker_crossed_circle,

    marker_semiellipse_left,

    marker_semiellipse_right,

    marker_semiellipse_up,

    marker_semiellipse_down,

    marker_triangle_left,

    marker_triangle_right,
```

```
        marker_triangle_up,
        marker_triangle_down,
        marker_four_rays,
        marker_cross,
        marker_x,
        marker_dash,
        marker_dot,
        marker_pixel
    };
```

4.2.1 Declaration

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

4.2.2 Creation

```
    renderer_markers(base_ren_type& rbuf) :
```

```
                        base_type(rbuf)
    {}
```

正如你所见，构造的过程和 renderer_primitives 是一样简单的。

4.2.3 Member Functions

marker 的所有函数都接受 x 、y 和半径作为参数，这些值以像素数来衡量。在 pixel() 中，半径值没有任何作用。填充(fill) 和边线(line)颜色都来自 renderer_primitives 基类。

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

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

当然，有一些为图方便的函数存在，这些函数基本上就是使用 switch/case 来构造：

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

根据给定的类型来描画一个 marker。

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

根据给定的类型描画一系列的 marker，坐标存储在 x 和 y 两个数组中。

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

根据给定的类型描画一系列的 marker，坐标和半径值分别存储在 x、 y 和 r 三个数组中。

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

根据给定的类型描画一系列的 marker，坐标、半径值以及颜色分别存储在 x、 y 、 r 和 fill_colors 四个数组中。

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

```
                marker_e type);
```

根据给定的类型描画一系列的 marker，坐标、半径值以及颜色分别存储在 x、 y 、 r 、fc 和 lc 等数组中。

5 Scanline Containers

底层的 render 操作的是最简单的数据形式，当然它们本身也很简单。其实 pixel format renderer 并不是 AGG 必须的组成部分，它们是可以替换或是改写的。比如说，如果你有一个功能相似的 API，而且这个 API 带有硬件加速，那么使用这个 API 来代替纯软件的混合渲染（主要是 alpha 混合）会更好。当然，也可以使用 Intel SSE/SSE2 来编写优化过的 renderer。AGG 其它的渲染功能都是基于这些简单的类。

描画去锯齿的对象时我们首先要对它进行光栅化。AGG 中主要的光栅化技术是基于 scanline 的，也就是说，一个多边形会先转换成很多的水平扫描线，然后再逐个描画这些扫描线。同样的， scanline rasterizer 并不是唯一可以用于生成水平扫描线(scanline)的类，你也可以使用一些容器，甚至是你自己的 super-ultra-mega rasterizer 。

在从 rasterizer（光栅化器） 到 scanline renderer 之间的信息转换时，用到了 scanline 容器。一个 scanline 由很多的水平的、不相交的 span 组成。这些 span 按 x 轴排序。这就意味着并没有提供排序这个操作，必须在将 span 添加到 scanline 时就把顺序处理好。如果顺序不是按要求排列的，那么可能会出现未定义的行为。

AGG 中有以下三种 scanline 容器。

scanline_u - 未包装的 scanline 容器。

scanline_p - 包装过的 scanline 容器。

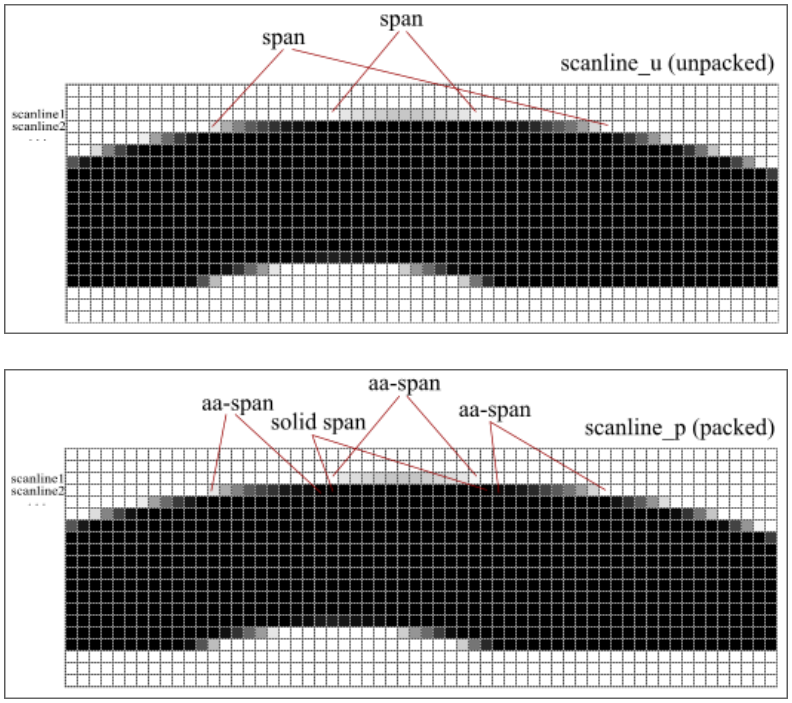
scanline_bin - 原始的，“带锯齿”的 scanline 容器。

前面两种容器可以包含去锯齿信息，第三种不行。

重要信息！

所有的 scanline 容器都为速度进行过优化，但没有考虑内存用量。事实上，它们会为最坏情况申请内存，所以有开销有点大。如果你只是使用少量的 scanline 的话，这不会成为瓶颈，但如果为存储整个形状而使用 scanline 的数组可能不是一个好主意，因为这会使用超出结果图像大小的内存量。

包装的(packed） scanline 与未包装的（unpacked） scanline 的区别在于， unpacked scanline 总是会为所有像素保存覆盖值（coverage value），包括那些全部覆盖在多边形内部的像素。而 packed scanlines 则会把具有相同 coverage value 的像素合并成 solid span。



看起来好像使用 packed 版的 scanline 总是会好一点，但在实践中并非如此。但你渲染一些大型的实心多边型时，使用 scanline_p 会更快，那是因为多边型的面积比周长要大很多（从像素的角度来衡量）。但是如果是渲染字符的时候绝对应该使用 scanline_u ，这样就能处理更少的 span 了，（scanline_u 会比 scanline_p）大约少产生三倍左右的 span，而 span 数本身就是一项重要的开销。同样的，在大多数 span 产生器中（比如 gradients, Gouraud shader, 以及 image 产生器等），span 的数量更加是（效能的）关键，所以一般也不会使用 scanline_p。

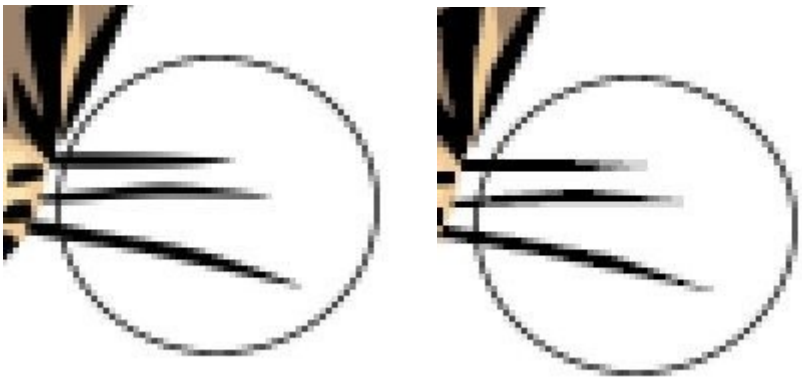
6 Gamma Correction

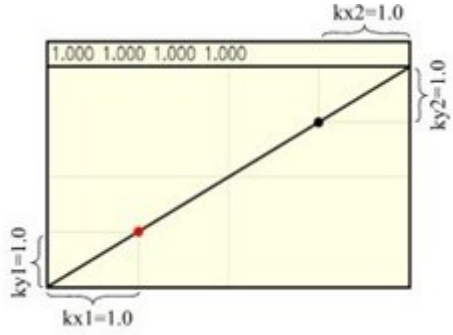
去锯齿是很复杂困难的。它的困难之处不仅仅在于算法上，也在于图像的视觉效果是与显示设备有关的。去锯齿的图像在 CRT 显示器和 LCD 显示器上的效果是不一样的。通常，这是一门叫颜色管理(Color Management)的科学（或许应该叫艺术）。Anti-Grain Geometry 使用的是一种尽量让我们达到最好效果的去锯齿方法。渲染的过程中会计算每个边界像素的覆盖值，这个值会反应成去锯齿的级别(Anti-Aliasing levels）。 Anti-Aliasing Geometry 使用 256 级，这对于任何实际用途来说都已经足够了，（这种详细的分

级产生的效果）比大部分 5 级去锯齿的软件要好，比如 Adobe 的大部分产品，以及 True-Type 的字体渲染器等。我非常确定我使用的渲染方法能给出最好的效果。但当我试着使用 Anti-Grain Geometry 和 Adobe SVG Viewer 去渲染同一个图片时，我发现 Adobe SVG Viewer 虽然只用了 5 级的去锯齿层级，但效果有时候看起来要比 Anti-Grain Geometry 却要好。

左边图是使用 Anti-Grain Geometry 渲染的，右边的是使用 Adobe SVG Viewer。

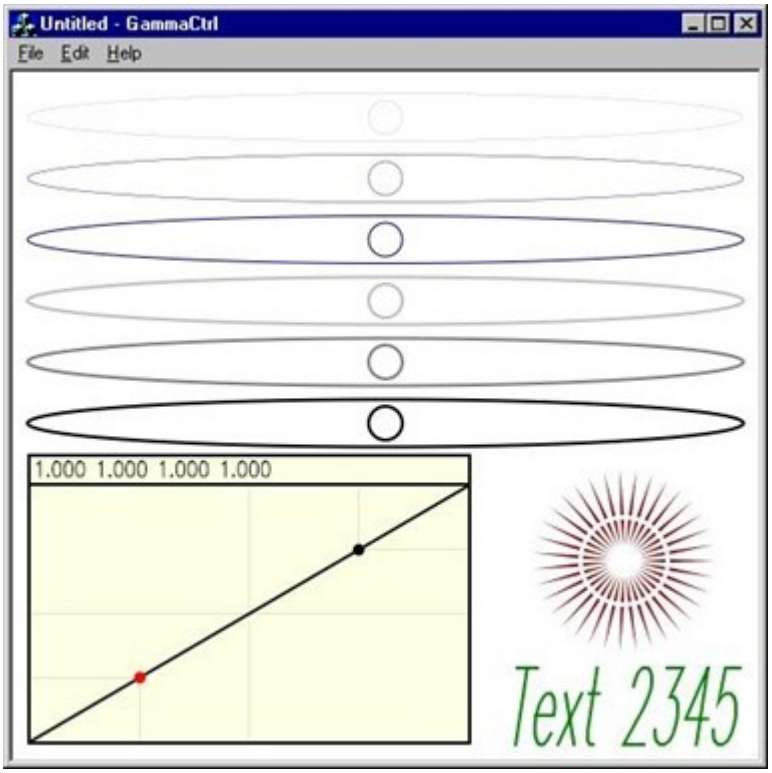
使用 5 级去锯齿的 Adobe SVG Viewer 渲染时，狮子的须会看起来很平滑。至少在 CRT 显示器上看起来是这样的。但是，大形图像的渲染仍然告诉我们，Adobe Viewer 的去锯齿分层是不够的。





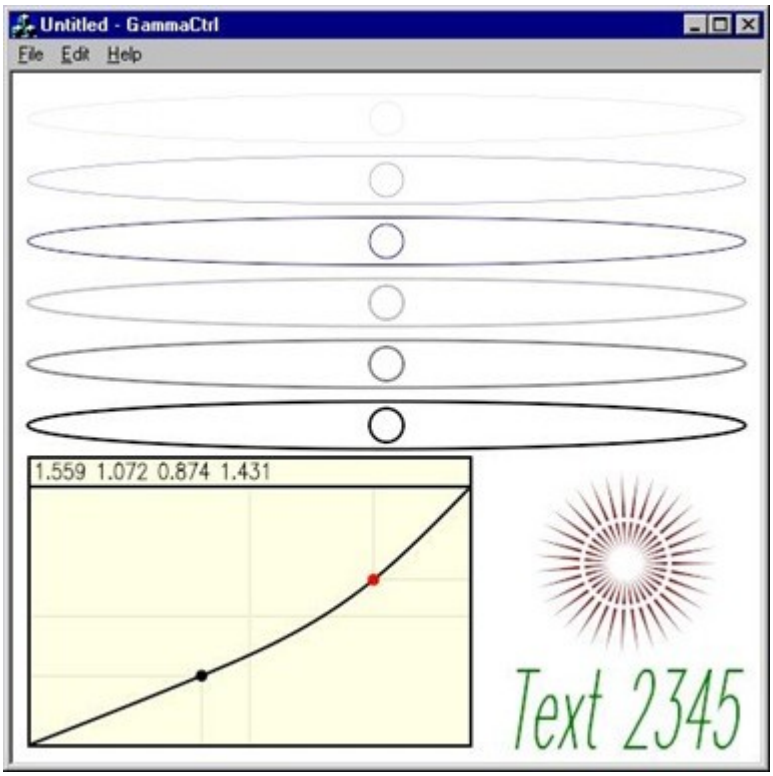
显然， Anti-Grain Geometry 可以渲染得更好，但是，对于像素覆盖值到亮度之间的对应依赖使用一个简单的线性关系并不是最好的，需要进行校正。在颜色管理中，这被称为 Gamma 校正。为了进行 gamma 校正，我使用了一个 256 个值的数组，可以给出对应覆盖值的像素的亮度。如果数组中的值都等于它们的索引值，比如，0,1,2,3,4……也就等于是没有进行 Gamma 校正。这个数组可以使用任何方法来进行计算，但最简单的方法是使用 B-Spline 曲线，通过两个相关点和四个象限（kx1, ky1,

kx2, ky2）来决定它的形状。 我创建了一个程序，使用特殊的 gamma 校正控制来计算 gamma 值数组中的各个值。这个程序中画了 6 个非常扁的椭圆，6 个小圆圈，以及一些其它的用来测试去锯齿效果的图像。



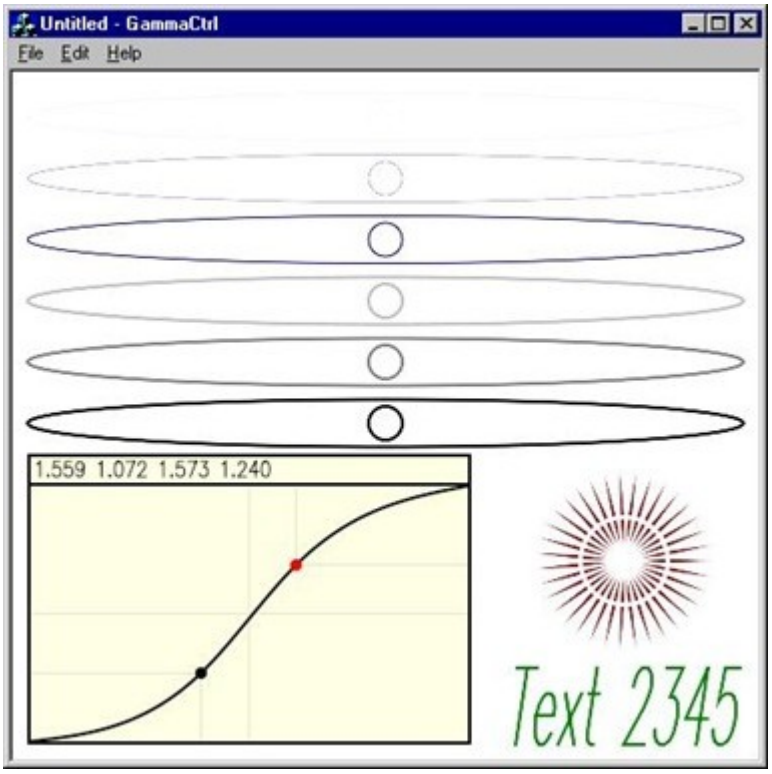
无 gamma 校正

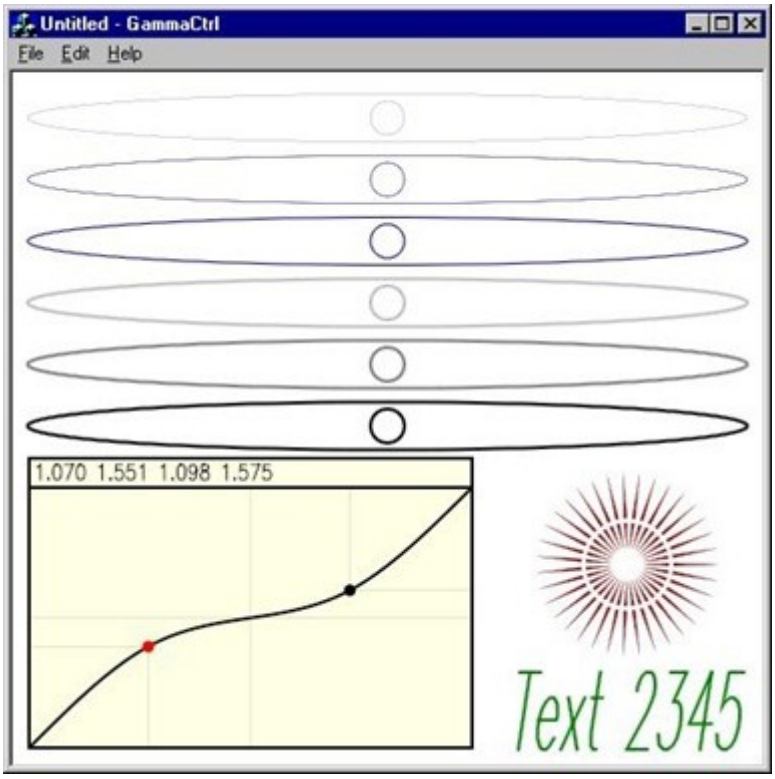
控制点可以在它们所在的象限内移动，下面的图在 CRT 显示器上看起来的效果就要好多了。

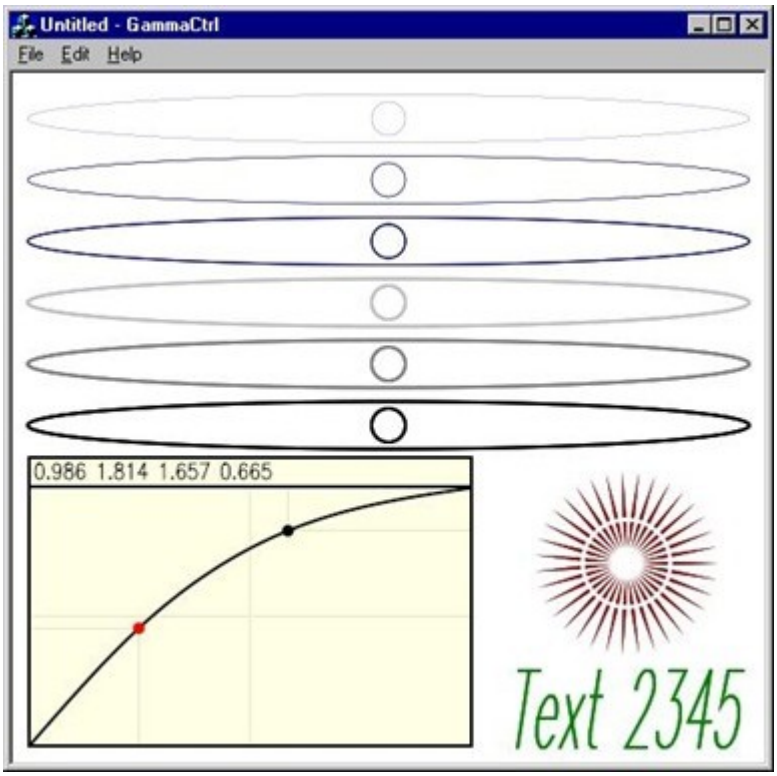


为 CRT 进行的 gamma 校正

对于某些的宽度和亮度，我们可以获得最好的显示显示效果，但这种情况并不通用。上面的例子中，就是一种虽然对某些参数下并非最好效果，但在平均情况下效果很好的（Gamma 校正方法），在 CRT 显示器上是如此，在 LCD 上也是如此。下面是一些其它形状 gamma 典型的例子。

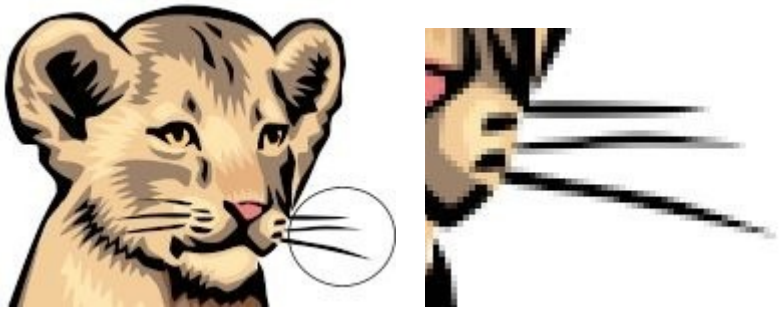

Untitled - GammaCtrl
File Edit Help
1.559 1.072 1.573 1.240
Text 2345


Untitled - GammaCtrl
File Edit Help
1.070 1.551 1.098 1.575
Text 2345



另外，gamma 校正的效果与图像的内容关系也很密切。上面例子中，那些画椭圆效果很好的 gamma 值，描画文字时效果却不太好。文字需要比较明显的边界形状，但大形的几何图像用平滑的边界看起来更好些。

下面在 CRT 上这是使用了 gamma 校正之后画的狮子，现在它看起来要比 Adobe SVG Viewer 看起来要好了。



你可以下载这个程序：Gamma Control。

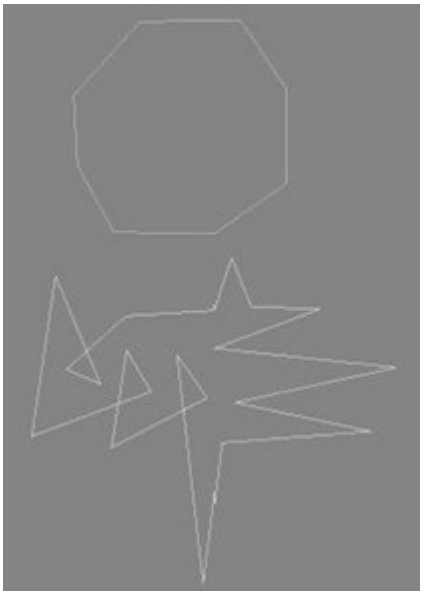
这个程序的源代码在分布的程序包的 example 目录中可以找到。

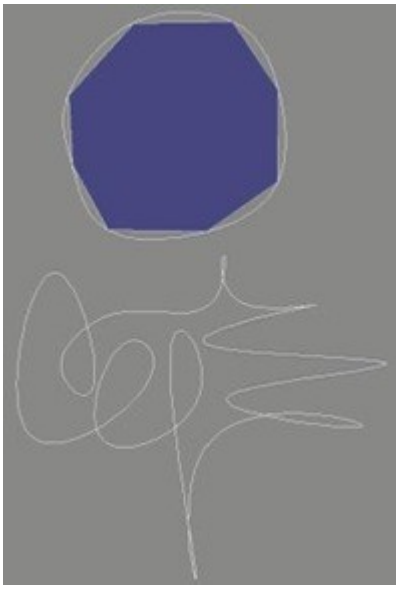
7 贝塞尔插值

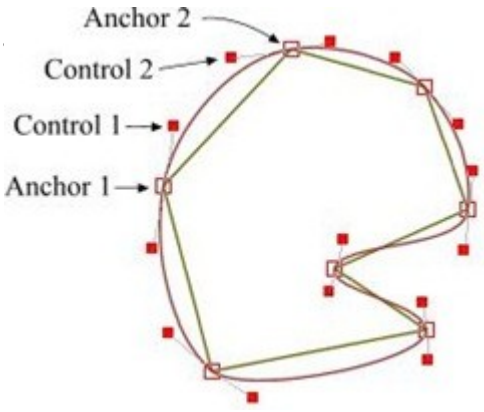
Interpolation with Bezier Curves　贝塞尔插值

A very simple method of smoothing polygons　一种非常简单的多边形平滑方法

之前 comp.graphic.algorithms 上有一个讨论，是关于怎么样使用曲线对多边形进行插值处理，使得最终产生的曲线是光滑的而且能通过所有的顶点。Gernot Hoffmann 建议说使用著名的 B-Spline 来进行插值。这里有他当时的文章。B-Spline 在这里效果很好，它看起来就像是一个固定在多边形顶点上的橡皮尺（elastic ruler）。

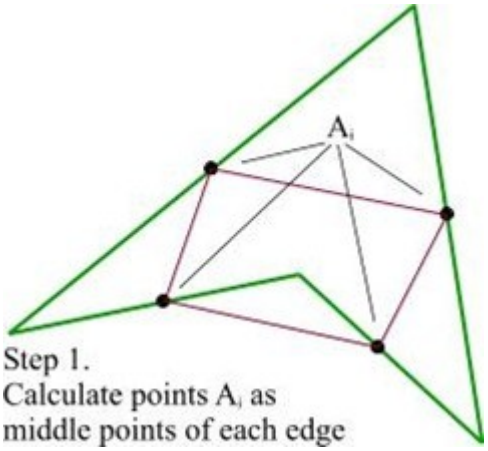




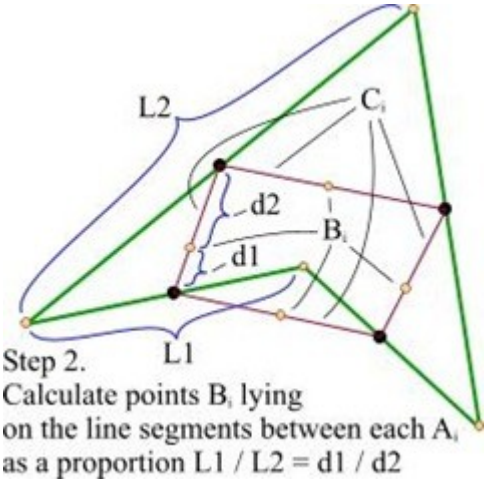


但我有个大胆的推测，我觉得肯定还存在更简单的方法。比如，使用三次贝塞曲线(cubic Bezier)进行近似。贝塞尔曲线有两个固定点（起点和终点），另加两个决定曲线形状的控制点(CP）。关于贝塞尔曲线的更多知识可以在搜索引擎中找到，比如，你可以参考 Paul Bourke 的站点。现在给贝塞尔曲线的锚点（固定点），也就是多边形的某一对顶点，那么问题是，我们怎么计算控制点的位置？我运行 Xara X 然后画出了右边这个图形，这很简单，所以我决定尝试下计算出它们的坐标。很显然，多边形两条相邻边的两个控制点与这两个控制点之间的顶点应该在一条直线上，只有这样，两条相邻的插值曲线才能平滑地连接在一起。所以，这两个控制点应该是相对于顶点是对称的，不过，也不全是……，因为真正的对称就要求它们与中心点的距离应该是相等的，但对于我们的情况中并不完全是这样的。一开始，我试着先算出多边形两条边的角平分线，然后把控制点放在这条角平分线的垂直线上。但从图上可以看到，控制点的连线并不会总是垂直于角平分线的。

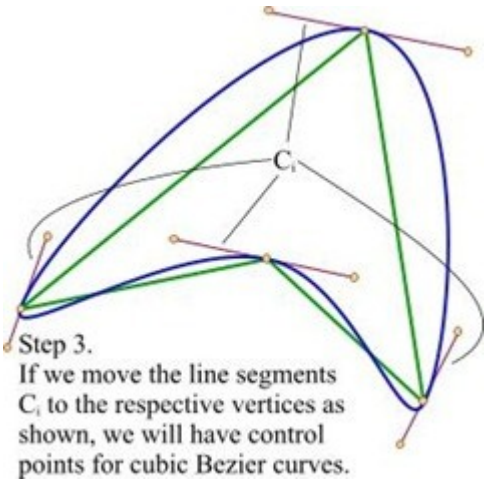
最终，我找到一个非常简单的办法，不需要任何复杂的数学计算。首先，我们计算出多边形所有边线的中点，Ai。



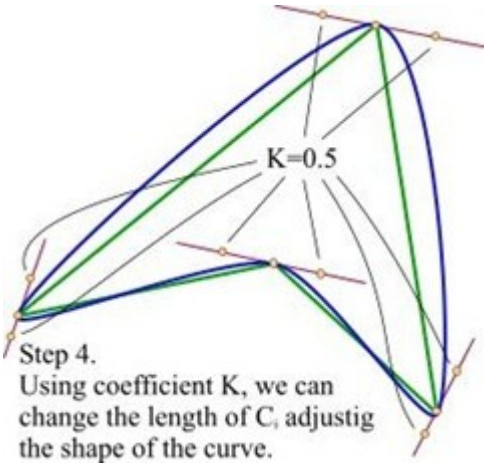
然后连接起相邻边中点，得到很多线段，记为 Ci 。并用图记的方法计算出 Bi 点。



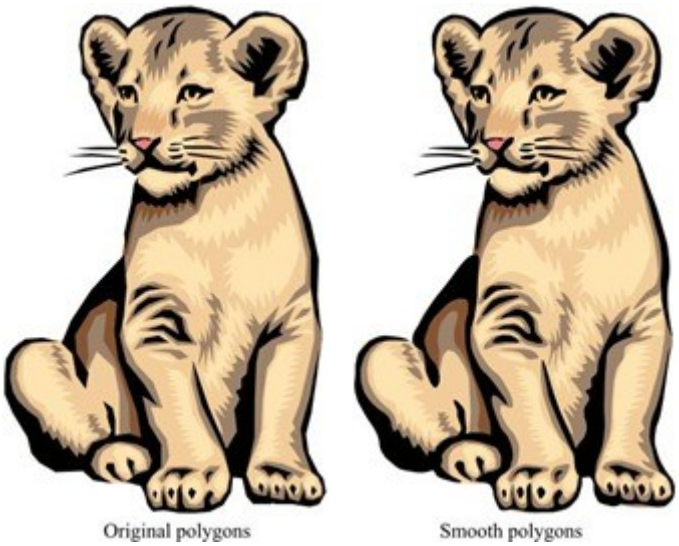
最后一步，只需要简单地将 Ci 进行平移，平移的路径就是每条线段上 Bi 到对应顶点的路径。就这样，我们计算出了贝塞尔曲线的控制点，平滑的结果看起来也很棒。



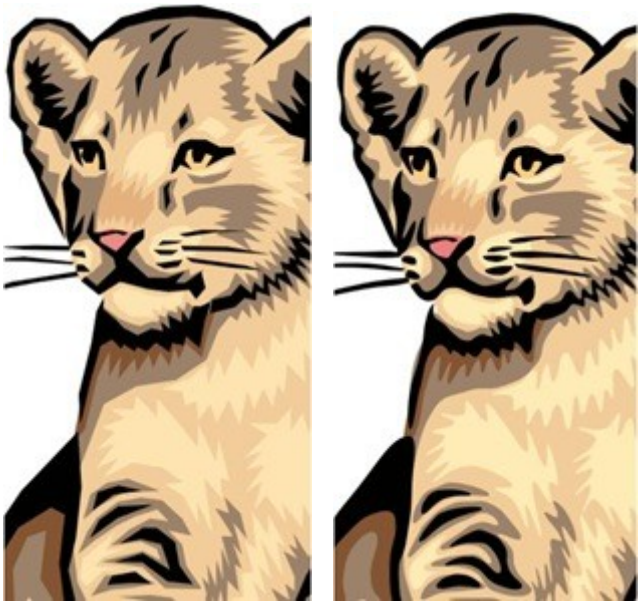
这里还可以做一点小小的改进，因为我们已经得到了一条决定控制点的直线，所以，我们可以根据需要，使控制点在这条直线上移动，这样可以改变插值曲线的状态。我使用了一个与控制点和顶点初始距离相关的系数 K ，用来沿直线移动控制点。控制点离顶点越远，图形看起来就越锐利。



下面是用原始形状和系统 K=1.0 的贝塞尔插值两种方法来描画的 SVG 的狮子。



下面是放大图



这个方法对于自相关的多边形也适用，下面的例子可以看到，结果非常有意思：

这个方法只是探索和经验式的，如果从严格的数学模型的角度看它可能是错误的。但在实际使用中的效果已经足够好了，而且这个方法只需要最小的计算量。下面的代码就是用来画出上面狮子图像的。这些代码并没有进行优化，只是用来演示的。里面有些变量计算了两次，在实际程序中，如果连续的步骤中都用到同一个变量值，我们可以先缓存变量值进行复用（以避免重复的计算）。

```
// Assume we need to calculate the control
// points between (x1,y1) and (x2,y2).
// Then x0,y0 - the previous vertex,
//      x3,y3 - the next one.
double xc1 = (x0 + x1) / 2.0;
double yc1 = (y0 + y1) / 2.0;
double xc2 = (x1 + x2) / 2.0;
double yc2 = (y1 + y2) / 2.0;
```

```
double xc3 = (x2 + x3) / 2.0;
double yc3 = (y2 + y3) / 2.0;
double len1 = sqrt((x1-x0) * (x1-x0) + (y1-y0) * (y1-y0));
double len2 = sqrt((x2-x1) * (x2-x1) + (y2-y1) * (y2-y1));
double len3 = sqrt((x3-x2) * (x3-x2) + (y3-y2) * (y3-y2));
double k1 = len1 / (len1 + len2);
double k2 = len2 / (len2 + len3);
double xm1 = xc1 + (xc2 - xc1) * k1;
double ym1 = yc1 + (yc2 - yc1) * k1;
double xm2 = xc2 + (xc3 - xc2) * k2;
double ym2 = yc2 + (yc3 - yc2) * k2;
// Resulting control points. Here smooth_value is mentioned
// above coefficient K whose value should be in range [0...1].
ctrl1_x = xm1 + (xc2 - xm1) * smooth_value + x1 - xm1;
ctrl1_y = ym1 + (yc2 - ym1) * smooth_value + y1 - ym1;
ctrl2_x = xm2 + (xc2 - xm2) * smooth_value + x2 - xm2;
ctrl2_y = ym2 + (yc2 - ym2) * smooth_value + y2 - ym2;
```

使用三次贝塞尔近似的代码：

```
// Number of intermediate points between two source ones,
// Actually, this value should be calculated in some way,
// Obviously, depending on the real length of the curve.
// But I don't know any elegant and fast solution for this
// problem.
#define NUM_STEPS 20
void curve4(Polygon* p,
            double x1, double y1,    //Anchor1
            double x2, double y2,    //Control1
            double x3, double y3,    //Control2
            double x4, double y4)    //Anchor2
```

```
{
    double dx1 = x2 - x1;
    double dy1 = y2 - y1;
    double dx2 = x3 - x2;
    double dy2 = y3 - y2;
    double dx3 = x4 - x3;
    double dy3 = y4 - y3;
    double subdiv_step  = 1.0 / (NUM_STEPS + 1);
    double subdiv_step2 = subdiv_step*subdiv_step;
    double subdiv_step3 = subdiv_step*subdiv_step*subdiv_step;
    double pre1 = 3.0 * subdiv_step;
    double pre2 = 3.0 * subdiv_step2;
    double pre4 = 6.0 * subdiv_step2;
    double pre5 = 6.0 * subdiv_step3;

    double tmp1x = x1 - x2 * 2.0 + x3;
    double tmp1y = y1 - y2 * 2.0 + y3;
    double tmp2x = (x2 - x3)*3.0 - x1 + x4;
    double tmp2y = (y2 - y3)*3.0 - y1 + y4;
    double fx = x1;
    double fy = y1;
    double dfx = (x2 - x1)*pre1 + tmp1x*pre2 + tmp2x*subdiv_step3;
    double dfy = (y2 - y1)*pre1 + tmp1y*pre2 + tmp2y*subdiv_step3;
    double ddfx = tmp1x*pre4 + tmp2x*pre5;
    double ddfy = tmp1y*pre4 + tmp2y*pre5;
    double dddfx = tmp2x*pre5;
    double dddfy = tmp2y*pre5;
    int step = NUM_STEPS;
    // Suppose, we have some abstract object Polygon which
```

```
        // has method AddVertex(x, y), similar to LineTo in
        // many graphical APIs.
        // Note, that the loop has only operation add!
        while(step--)
        {
            fx  += dfx;
            fy  += dfy;
            dfx  += ddfx;
            dfy  += ddfy;
            ddfx += dddfx;
            ddfy += dddfy;
            p->AddVertex(fx, fy);
        }
        p->AddVertex(x4, y4); // Last step must go exactly to x4, y4
    }
```

你可以下载一个能运行的画狮子的例子，对它进行旋转和缩放，也可以生成一些随机的多边形。点左键并拖动它可以围绕中心点旋转和缩放图像。点右键并从左向右拖动，可以改变系统数 K。 K＝1 时大约是距窗口左边 100 像素处。每次双击会产生一个随机的多边形，对于这些多边形，也可以进行旋转、缩放以及改变 K 值的操作。

8 适应性细分法描画贝塞尔曲线

Adaptive Subdivision of Bezier Curves

-- An attempt to achieve perfect result in Bezier curve approximation

8.1 介绍

贝塞尔曲线在现代的 2D 和 3D 描画中被广泛使用。在大多数的程序中，使用二次或是三次的曲线就足够了。在互联网上有大量的关于贝塞尔曲线的解释，我相信你看过之后就知道它是怎么回事了。现在，主要问题是我们如何（在计算机上）画出这种曲线。

曲线的描画通常都是使用一系列的短小的线段来近似的，这是描画曲线的唯一高效的方法。在这篇文章里，我要解释的是这样一种方法：如何通过使用最少的点来达到最完美的近似效果。

首先我们得先看一下 Paul Bourke 在下面这个链接中说明的一个基本方法：

http://astronomy.swin.edu.au/~pbourke/curves/bezier

下面的这段代码用来计算一条三次（贝塞尔）曲线上的任意一个点（这是从 Paul 的网页上拷下来的）

```
/*
Four control point Bezier interpolation
mu ranges from 0 to 1, start to end of curve
*/
XYZ Bezier4(XYZ p1,XYZ p2,XYZ p3,XYZ p4,double mu)
{
   double mum1,mum13,mu3;
   XYZ p;
   mum1 = 1 - mu;
   mum13 = mum1 * mum1 * mum1;
   mu3 = mu * mu * mu;
   p.x = mum13*p1.x + 3*mu*mum1*mum1*p2.x + 3*mu*mu*mum1*p3.x + mu3*p4.x;
   p.y = mum13*p1.y + 3*mu*mum1*mum1*p2.y + 3*mu*mu*mum1*p3.y + mu3*p4.y;
   p.z = mum13*p1.z + 3*mu*mum1*mum1*p2.z + 3*mu*mu*mum1*p3.z + mu3*p4.z;
   return(p);
}
```

这里面，有四个控制点和一个叫 "mu" 的参数，mu 的取值范围是 [0,1] 。原则上这就足够了，我们可以使用递增的 mu 来计算出一系统的点，并用这些点来画面近似用的线段。

8.2 Problems of the Incremental method

首先，这个方法很慢，2D 的情况下，每个点的计算需要进行 24 次浮点数乘法。不过这个问题可以很容易的解决，我们可以使用递增的方法代替这种直接计算，在这篇文章的最后有相关的描述：Interpolation with Bezier Curves，你可以看到，主循环中只有 6 次加法操作。就我所知，这是目前最快的方法，特别是在现代的处理器上更是如此，因为这些处理器对浮点数的计算都已经相当的快了。

但主要的问题是如何决定中间点的数目，还有如何对 mu 值进行递增。最简单的方法是选择某个步进，比如，每次增加 0.01 ，这样任何曲线都会被分成 99 条线段。当然，缺点也显而易见，对于长曲线中间点太少，而对于短曲线中间点则太多。

很显然，我们应该基于曲线的长度来计算步进，但计算曲线长度又要求我们能计算出曲线本身，于是，在这里我们陷入了经典的“第二十二条军规”的尴尬矛盾中（译注：因为我们对曲线的长度和形状都是不确定的，所以用线段来近似它，但最佳的线段的数目和长度最要求我们知道曲线长度和形状）。（所以，为此，我们要找别的方法），对于长度，采用有一个相当不错的估算方法：

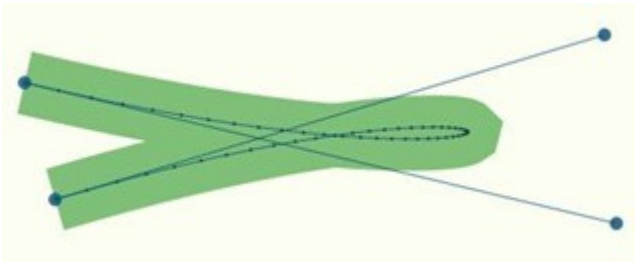
(p1,p2)+(p2,p3)+(p3,p4);

根据这个，我通过实验发现典型屏幕分辨率下做以下的估算就已经足够了：

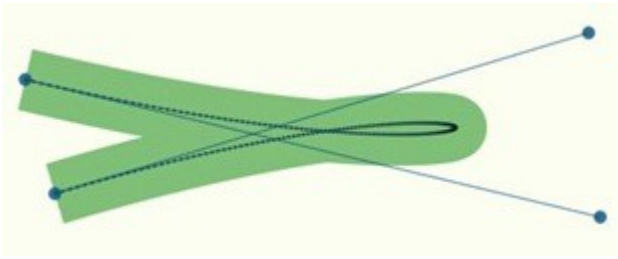
```
dx1 = x2 - x1;
dy1 = y2 - y1;
dx2 = x3 - x2;
dy2 = y3 - y2;
dx3 = x4 - x3;
dy3 = y4 - y3;
len = sqrt(dx1 * dx1 + dy1 * dy1) +
sqrt(dx2 * dx2 + dy2 * dy2) +
sqrt(dx3 * dx3 + dy3 * dy3);
num_steps = int(len * 0.25);
```

注意，我们在是去锯齿和亚像素精度的前提下来讨论这个的。对于常规的像素精度以及使用 MoveTo/LineTo 接口的情况来说，会有很大的不同。

但即使我们这样精确地估算曲线的步进，问题仍然存在。一条三次曲线可能会有非常急的转向，或是很窄的环，甚至是尖端。看下面这个图：

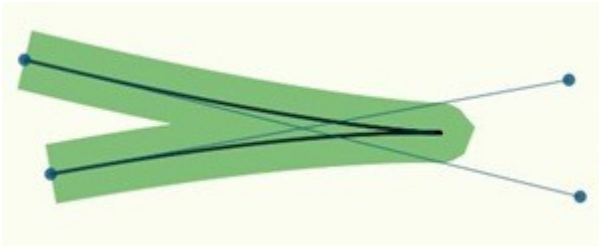


这张图是使用 52 条线段进行近似画出来的曲线。可以看到，使用等距的轮廓(stroke）画出来的环形看起来很不精确。为了使它更准确一些，我们必须要增加中间点的数量。



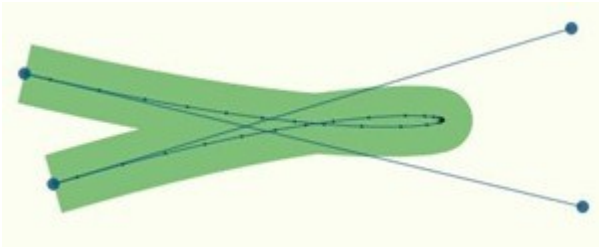
上面我们使用了 210 条线段来画，很明显，大部都是没用的。也就是在曲线“平坦”的部分产生的中间点太多了，而在曲线转弯处的中间点又太少。

但在使用很小的步进，也还会存在问题：

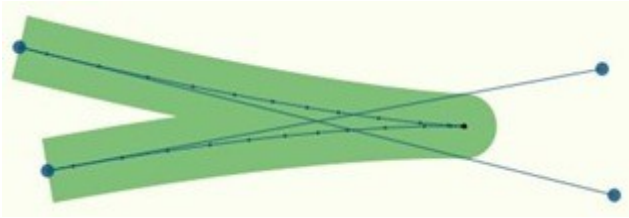


这条曲线使用了 1091 条线段，但在转弯处的效果仍然不行。

理想状态下，我们需要有一个自适应的步进，能使最终效果看起来如下图所示：



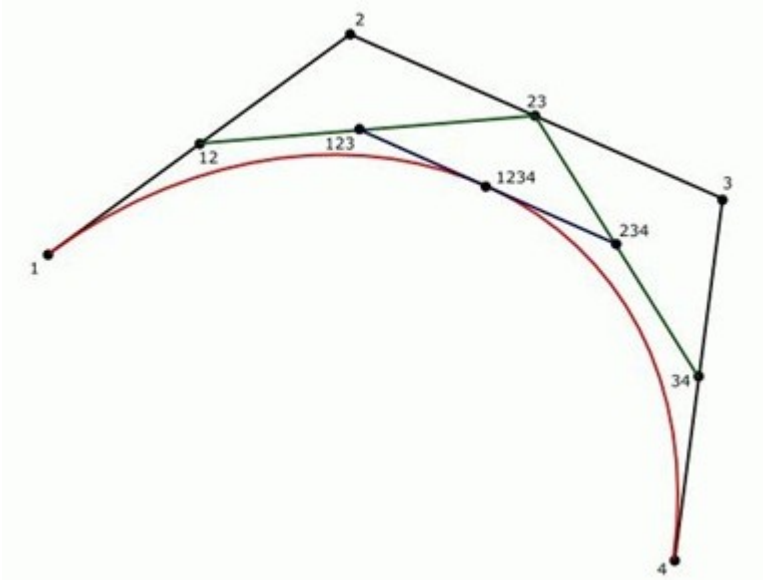
这图中只使用了 40 条线段，但却几乎完美地画出了曲线（考虑到轮廓的效果很好）。这才是我们想要的结果，而且我们的确做到了。你注意看，就算是那个很急的转弯处 stroke 的表现仍然很平滑，而且线段的数量也很合适，在这个例子中，只有 44 条（译注：怎么从 40 到 44 的我也没搞清楚，呵呵）。



8.3 Paul de Casteljau Divides and Conquers

Paul de Casteljau， 雪铁龙（Citroen）的一个非常有才华的工程师，他发现了贝塞尔曲线的一种非常有意思的属性，即，任意角度的曲线都可以被分成两条相同角度的相同曲线（译注：原文是，It's namely that

any curve of any degree can be easily divided into two curves of the same degree.我拿不准这里的 degree 是指什么）。下面的图很经典：



我们有 1,2,3,4 四个控制点。计算出它们的中间 12, 23, 34 ，以及“高一阶”的中点 123, 234 ，最后一阶的中间 1234 就落在曲线上了。这里也可以不用中点(系数 t ＝ 0.5)，而使用其它的 0 到 1 之间的系数。但在这篇文章中，我们使用中点。产生的两条新曲线与原曲线完全一致，但新曲线的控制点变成了：1,12,123,1234（左半边），和 1234,234,34,4（右半边）。

很明显，“新”的曲线比原来的曲线要平坦一点，所以，如果我们重复这个过程若干次，我们就可以把用线段来代替曲线。

细分的递归程序代码也相当的经典：

```
void recursive_bezier(double x1, double y1,
                      double x2, double y2,
                      double x3, double y3,
                      double x4, double y4)
{
    // Calculate all the mid-points of the line segments
    //----------------------
    double x12   = (x1 + x2) / 2;
    double y12   = (y1 + y2) / 2;
    double x23   = (x2 + x3) / 2;
    double y23   = (y2 + y3) / 2;
    double x34   = (x3 + x4) / 2;
    double y34   = (y3 + y4) / 2;
    double x123  = (x12 + x23) / 2;
    double y123  = (y12 + y23) / 2;
    double x234  = (x23 + x34) / 2;
    double y234  = (y23 + y34) / 2;
```

```
        double x1234 = (x123 + x234) / 2;
        double y1234 = (y123 + y234) / 2;
        if(curve_is_flat)
        {
            // Draw and stop
            //----------------------
            draw_line(x1, y1, x4, y4);
        }
        else
        {
            // Continue subdivision
            //----------------------
            recursive_bezier(x1, y1, x12, y12, x123, y123, x1234, y1234);
            recursive_bezier(x1234, y1234, x234, y234, x34, y34, x4, y4);
        }
    }
```

就这些！这就是你画一条贝塞尔曲线所需要的一切。不过，魔鬼就在“当曲线是平坦”（代码中的curve_is_flat）的这个条件上。

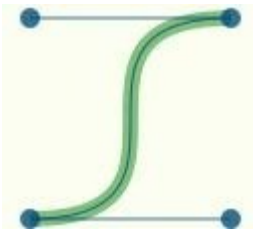
8.4 Estimation of the Distance Error

Casteljau 的细分法有很多优点。我们可以估计曲线的平坦程度，因为我们能得到这方面的信息：即起始点与其它的中间点。而在递增方法中，我们只有一个点的信息。当然，我们可以在计算之前取到最少两个点，然后再分析点的信息，但这样会变得相当的复杂。

停止细分程度有很多种策略，最简单的方法是，直接计算点 1 和点 4 之间的距离。这种方法不好，因为点 1 和点 4 一开始就可能是重合的，当然，有补救的办法，那就是第一次细分的时候我们强制进行进行。

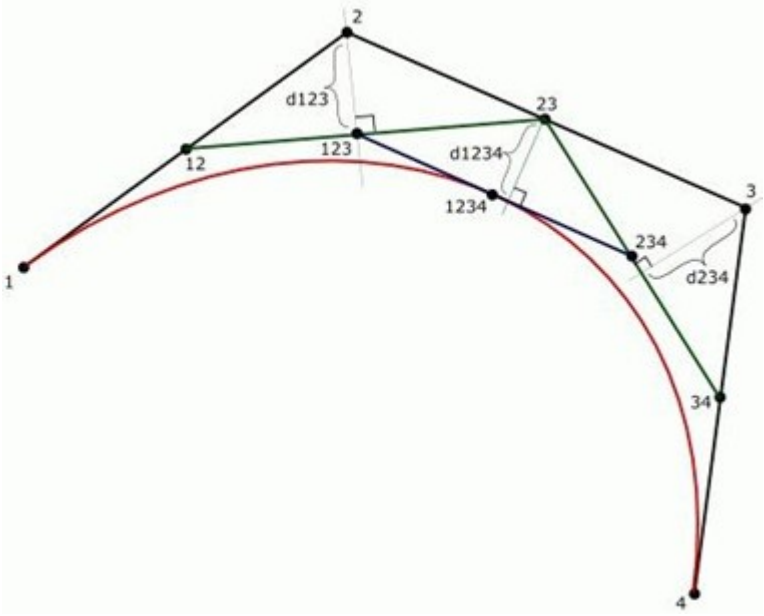
更好的办法是计算一个点到一条直线的距离，它与估算的误差是成比例的。但问题是，我们应该计算哪段距离？

一开始我觉得计算点 1234 到 直接（1-4）的距离就可以了，但，在曲线是“Z”型的时候，比如(100, 100, 200, 100, 100, 200, 200, 200)，这个值为零。



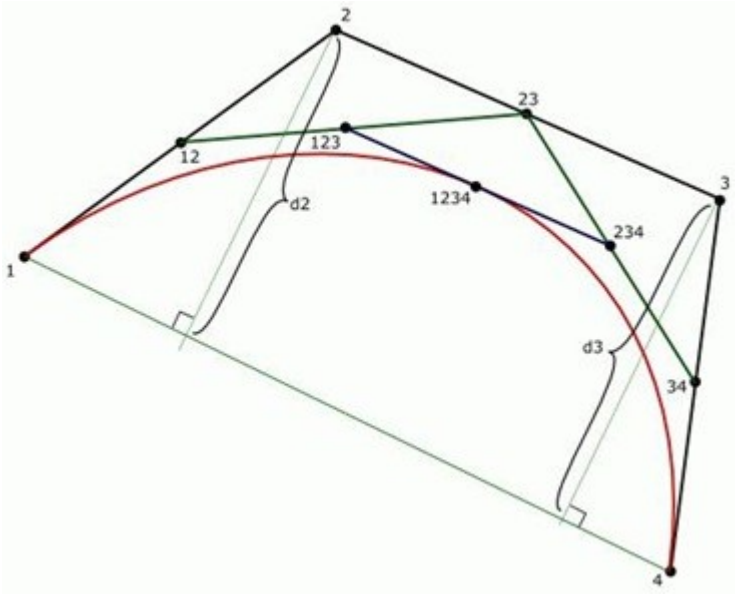
但这种情况可以通过强制进行第一次细分操作来解决。

通过进行多次的实验，我发现计算 3 个长度的和会更好：



看上图，我们计算了这三个长度的和： d123+d1234+d234 。这个方法不需要特别处理第一次细分的情况。

然后，在进行了更多次的实验之后，我发现计算另两段长度的和会甚至更好，而且这种方法也不需要对第一次细分进行特殊处理。这个和是 d2+d3 ：



就是它了，我们已经得到一个相当好的误差的估算了。计算一个点到一条直线的距离看起来开销不小，但实际上不是这样的，我们不需要计算平方根，要知道，我们要做的只是进行估计，然后对比误差（是或不是）。

所以，代码可以像下面这样写：

```
void recursive_bezier(double x1, double y1,
```

```
                                double x2, double y2,
                                double x3, double y3,
                                double x4, double y4)
{
    // Calculate all the mid-points of the line segments
    //----------------------
    double x12   = (x1 + x2) / 2;
    double y12   = (y1 + y2) / 2;
    double x23   = (x2 + x3) / 2;
    double y23   = (y2 + y3) / 2;
    double x34   = (x3 + x4) / 2;
    double y34   = (y3 + y4) / 2;
    double x123  = (x12 + x23) / 2;
    double y123  = (y12 + y23) / 2;
    double x234  = (x23 + x34) / 2;
    double y234  = (y23 + y34) / 2;
    double x1234 = (x123 + x234) / 2;
    double y1234 = (y123 + y234) / 2;
    // Try to approximate the full cubic curve by a single straight line
    //------------------
    double dx = x4-x1;
    double dy = y4-y1;
    double d2 = fabs(((x2 - x4) * dy - (y2 - y4) * dx));
    double d3 = fabs(((x3 - x4) * dy - (y3 - y4) * dx));
    if((d2 + d3)*(d2 + d3) < m_distance_tolerance * (dx*dx + dy*dy))
    {
        add_point(x1234, y1234);
        return;
    }
    // Continue subdivision
    //----------------------
    recursive_bezier(x1, y1, x12, y12, x123, y123, x1234, y1234);
    recursive_bezier(x1234, y1234, x234, y234, x34, y34, x4, y4);
}

void bezier(double x1, double y1,
            double x2, double y2,
            double x3, double y3,
            double x4, double y4)
{
    add_point(x1, y1);
    recursive_bezier(x1, y1, x2, y2, x3, y3, x4, y4);
    add_point(x4, y4);
}
```

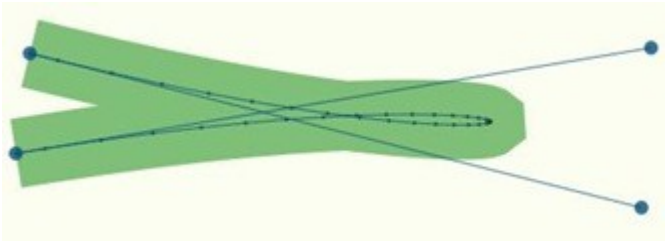
m_distance_tolerance 是能接受的最大距离的平方。对于一般的屏幕分辨率，可以取 ： 0.5*0.5＝0.25 。

现在，我们已经解决了通过最少的点来近似整条曲线的问题，误差的最大值是固定的。记得吧，之前的递增方法会产生过多的点，而且在曲线平坦的部分近似误差过少（因为分出来的点过多），而在曲线弯曲处的误则大太。而细分方法可以最小化需要的点的数目，并使得最大误差保持固定。

但这个方法还有很严重的问题。如果点 1 和点 4 重合，那么(d2 + d3)*(d2 + d3) 和 (dx*dx + dy*dy)的值都是 0 。在这里，浮点数的“小于”和“小于或等于”的比较会产生不同的结果，这是一个很少见的情况。在这种情况下细分会继续下去，而如果结果条件是“小于或等于”的话，则细分会停止。但这样的代码在 3 个点或 4 个点重合时会产生栈益出。看起来可以在条件中使用“小于或等于”然后强制进行第一次细分这样的办法来解决。但仔细的分析会发现这办法不行。因为第一次细分后，子曲线然后可能会产生一个环，并有重合的点。这些问题后面都会解决，不过我们先看看如果用角度来估计的话，会是什么样的。

8.5 Estimation of Tangent Error

上面的代码，在近似曲线时，用最优的点数保持了固定的最大误差。但它与递增方法一样，在曲线的弯曲处有同样的问题：



这个近似中用了 36 段线段，最大的误差是 0.08 个像素。

很显示，仅仅使用长度来估计是不够的。为了让宽画笔（wide strokes）在任何角度都看起来平滑，我们应该对曲率进行估计，而不考虑实际的长度。

细分后，新产生的两条曲线会比原来的曲线平坦。同时点 1 和点 4 的角度也一样。如果它们很不一样，曲线在这个点有一个很急的弯曲，那么就要继续进行细分。

注意：

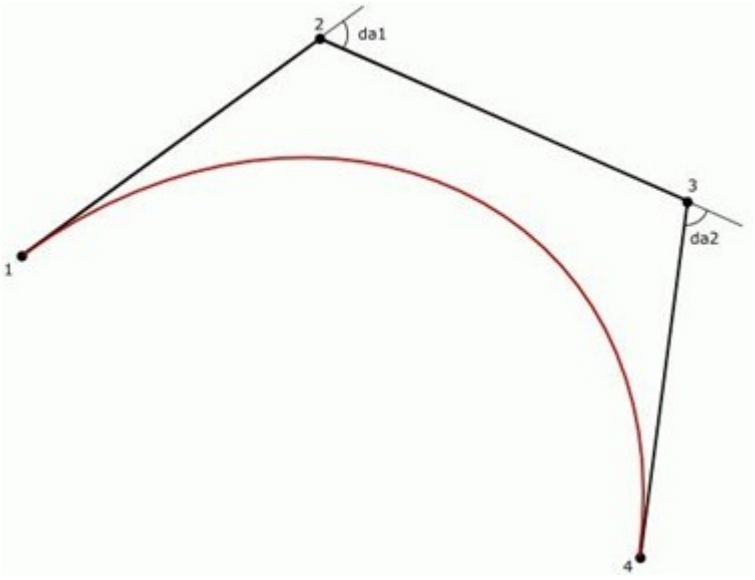
计算角度时我直接使用了 atan2 函数，这个函数开销很大，会明显地降低整个算法的速度。但是值得注意的是它并不总是很重要的。它只在我们需要描画一条等距的曲线时才比较重要，也就是使用非常宽的画笔（stroke）时。如果不是画 stroke，或是 stroke 的宽度小于等于一个像素，那么使用距离来估算就已经可以够好了。

好了，我们现在引入另一个判定标准，m_angle_tolerance :

```
// If the curvature doesn't exceed the distance_tolerance value
// we tend to finish subdivisions.
//----------------------
if(m_angle_tolerance < curve_angle_tolerance_epsilon)
{
    m_points.add(point_type(x1234, y1234));
    return;
}

// Angle & Cusp Condition
//----------------------
double a23 = atan2(y3 - y2, x3 - x2);
double da1 = fabs(a23 - atan2(y2 - y1, x2 - x1));
double da2 = fabs(atan2(y4 - y3, x4 - x3) - a23);
if(da1 >= pi) da1 = 2*pi - da1;
if(da2 >= pi) da2 = 2*pi - da2;
if(da1 + da2 < m_angle_tolerance)
{
    // Finally we can stop the recursion
    //----------------------
    m_points.add(point_type(x1234, y1234));
    return;
}
```

我们用 m_angle_tolerance 作为标志位，如果这个值小于一个特定值（epsion），那么我们就不再处理角度了。

下面的图展示了计算的方法：



嗯，在技术上计算一个角度就够了，比如第一条和第三条线段的夹角（(1-2 and 3-4)，但把两个角都计算出来会另有用处，比如后面提到的尖端的处理方法。

这段代码会认为，两条连续的线段对于计算看起来平滑的 stroke 已经足够了。但它也有问题，首先就是在处理尖端（Cusp）的时候。

8.6 Processing of the Cusps

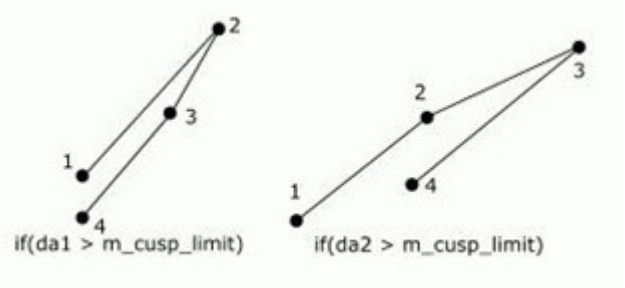
一个三次的（贝塞尔）曲线可以产生一个尖端（cusp），尖端是曲线上切线不连续的点。这种点会使曲线产生一个锐利的转弯，这种转弯无论你怎么放大曲线，看起来都一样的急（Sharp turn）。换句话说，在这个点附近，mitter-join stroke 不可能看起来平滑，因为切线不连续（译注：导数不连续），所以 stoke 也会产生非常锐利的转弯。

为了产生这样的 cusp ，贝塞点曲线的控制点应该是一个 X 字母的样子：

(100, 100, 300, 200, 200, 200, 200, 100)

在这种情况下，理论上 da1 + da2 < m_angle_tolerance 是不可能产生的，实践中，只要 4 个点重合在一起，那么条件就成立了，所有 atan2 调用的参数都会是 0 。这时，递归层次会非常非常深，很可能会产生一个栈益出。那很不好，不过好在有一个很简单的解决办法：

```
if(da1 > m_cusp_limit)
{
    m_points.add(point_type(x2, y2));
    return;
}

if(da2 > m_cusp_limit)
{
    m_points.add(point_type(x3, y3));
    return;
}
```



注意，我们通常会使用的是曲线上的点 1234 ，因为它使得我们可以对称地处理端点。但在这里，我们用的点是尖端点，我是经过多次实验后产生这个想法的，这就保证了 stroke 的截面是垂直于尖端点的。

附记：

一开始我实验时只考虑角度这一个条件，这让曲线在转弯时始终保持平滑，而且轮廓看起来也很漂亮，但在平坦的部分看起来不够准确。最后我得到这样的结论：只有结合着距离和角度两个条件，才能使用尽量少的点数产生平滑的 stroke 。

8.7 The Devil is in the Details

但那还不够好！有很多病态的例子可以使得这个算法失败。我花了很长的时候去分析这些情况，并希望通过加些一些什么东西以一劳永逸地解决所有问题。如果考虑到所有情况的话，那么整个代码会成变成一陀乱麻，我讨厌这样的代码。比如说，有点重合的时候会导致角度计算失败，或是，在某种情况下，atan2(0,0)会返回 0,比如水平直线。现在，我跳过所有我遇到的痛苦，给你一个我的结论：

我的实验显示，所有病态情况都可以对付“同在一条直线”(collinear case)的方法来应对。

为了估计距离，我们有下面的计算式：

```
double dx = x4-x1;
double dy = y4-y1;
double d2 = fabs(((x2 - x4) * dy - (y2 - y4) * dx));
double d3 = fabs(((x3 - x4) * dy - (y3 - y4) * dx));
```

如果用 d2 除以 line1-4，我们就可以得到点 2 到直接 1-4 的距离，不过，就像早先提到的一样，我们并不需要真实的距离长度，只是用来进行比较就可以了。在实践中也就意味着如果我们引入一个 curve_collinearity_epsilon ，我们可以过滤掉重合的点，和其它各点同线的情况。

```
double dx = x4-x1;
double dy = y4-y1;
double d2 = fabs(((x2 - x4) * dy - (y2 - y4) * dx));
double d3 = fabs(((x3 - x4) * dy - (y3 - y4) * dx));
```

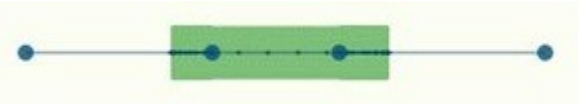
```
if(d2 > curve_collinearity_epsilon && d3 > curve_collinearity_epsilon)
{
    // Regular care
    . . .
}
else
{
    if(d2 > curve_collinearity_epsilon)
    {
        // p1,p3,p4 are collinear (or coincide), p2 is considerable
        . . .
    }
    else
        if(d3 > curve_collinearity_epsilon)
        {
            // p1,p2,p4 are collinear (or coincide), p3 is considerable
            . . .
        }
        else
        {
            // Collinear case
            . . .
        }
}
```

在各点同线的情况下（或是有重合点的情况下）我们要用不同的方法。其中有一种情况是四个点都在一条直线上。惊奇的是，是否同线的检查可以让我们在曲线有尖端的情况下远离深度的递归。我们可以不对尖端进行限制，对掉相关的代码。但我仍然保留了这些代码，因为在某些情况下，这些代码会有用。

8.8 Collinear Case

感谢 Timothee Groleau, http://www.timotheegroleau.com 里有他的方法，可以很简单地估计出曲率。就是点 1234 和线 1-4 的中点之间的距离。这与估算点和线之间的距离完全不同。他的方法清寒给出一个近似的值，虽然这个值仍然不够。但他的方法有一个很重要的优点，即，可以处理下面这种同线的情况。当四个点的顺序是下面这样的时候：

2--------1---------4----------3



所有使用点到线距离的判定方法在这种情况下都会失效。这里只有一条线段 1-4 ,而这明显是不对的。Timothee 的判定方法在这种情况下仍然有效，这样我们就有了一个检测共线情况的机制。因此，处理共线情况的代码就相当简单了：

```
. . .
else
{
    // Collinear case
    //-----------------
    dx = x1234 - (x1 + x4) / 2;
    dy = y1234 - (y1 + y4) / 2;
    if(dx*dx + dy*dy <= m_distance_tolerance)
    {
        m_points.add(point_type(x1234, y1234));
        return;
    }
}
```

唯一要做的事就是要强制进行一次细分，因为可能会有上面提到过的“Z”型的情况。

嗯，伪代码如下：

```
void recursive_bezier(double x1, double y1,
                      double x2, double y2,
                      double x3, double y3,
                      double x4, double y4,
                      unsigned level)
{
    double x12   = (x1 + x2) / 2;
    double y12   = (y1 + y2) / 2;
    double x23   = (x2 + x3) / 2;
    double y23   = (y2 + y3) / 2;
    double x34   = (x3 + x4) / 2;
    double y34   = (y3 + y4) / 2;
    double x123  = (x12 + x23) / 2;
    double y123  = (y12 + y23) / 2;
    double x234  = (x23 + x34) / 2;
    double y234  = (y23 + y34) / 2;
    double x1234 = (x123 + x234) / 2;
    double y1234 = (y123 + y234) / 2;
```

```
        if(level)
        {
            if(curve_is_flat)
            {
                draw_line(x1, y1, x4, y4);
                return;
            }
        }
        recursive_bezier(x1, y1, x12, y12, x123, y123, x1234, y1234, level + 1);
        recursive_bezier(x1234, y1234, x234, y234, x34, y34, x4, y4, level + 1);
    }
```

也可以对递归进行限制，当然，这里看起来已经不必要去限制它了（因为已经有其它的判定方法来停止递归），但是严格的数学证明对我来说非常困难，所以，我直接把递归限定在 32 次以内。在实践中，即使是最恶劣的情况下（非常长的曲线，几千个点，很小的错误，有尖端点），我也没见过超过 17 次的递归，

Timothee 的判定方法在另一种共线情况下会产生很多的点： 1---2---3-------4 。 如果先检测出这种情况可能会比较，这样就只需要产生两个点(1-4)，但我觉得这无关紧要，因为严格共线的情况非常少。所以，实际上这并不会影响性能。

还有另外两种共线的情况：

```
    if(d2 > curve_collinearity_epsilon)
    {
        // p1,p3,p4 are collinear (or coincide), p2 is considerable
        . . .
    }
    else if(d3 > curve_collinearity_epsilon)
    {
        // p1,p2,p4 are collinear (or coincide), p3 is considerable
        . . .
    }
```

那很简单，我们只是把点 1 和点 3 看起重合（点 2 和点 4 是第二种情况）。嗯，这里还有一个细节我想告诉你。

8.9 The Full Code

最后，我们看起完整的代码：

```
//------------------------------------------------------------------------
void curve4_div::init(double x1, double y1,
                      double x2, double y2,
                      double x3, double y3,
                      double x4, double y4)
{
    m_points.remove_all();
    m_distance_tolerance = 0.5 / m_approximation_scale;
    m_distance_tolerance *= m_distance_tolerance;
    bezier(x1, y1, x2, y2, x3, y3, x4, y4);
    m_count = 0;
}

//------------------------------------------------------------------------
void curve4_div::recursive_bezier(double x1, double y1,
                                  double x2, double y2,
                                  double x3, double y3,
                                  double x4, double y4,
                                  unsigned level)
{
    if(level > curve_recursion_limit)
    {
        return;
    }
    // Calculate all the mid-points of the line segments
    //----------------------
    double x12   = (x1 + x2) / 2;
    double y12   = (y1 + y2) / 2;
    double x23   = (x2 + x3) / 2;
    double y23   = (y2 + y3) / 2;
    double x34   = (x3 + x4) / 2;
    double y34   = (y3 + y4) / 2;
    double x123  = (x12 + x23) / 2;
    double y123  = (y12 + y23) / 2;
    double x234  = (x23 + x34) / 2;
    double y234  = (y23 + y34) / 2;
    double x1234 = (x123 + x234) / 2;
    double y1234 = (y123 + y234) / 2;
    if(level > 0) // Enforce subdivision first time
    {
        // Try to approximate the full cubic curve by a single straight line
        //------------------
```

```
double dx = x4-x1;
double dy = y4-y1;
double d2 = fabs(((x2 - x4) * dy - (y2 - y4) * dx));
double d3 = fabs(((x3 - x4) * dy - (y3 - y4) * dx));
double da1, da2;
if(d2 > curve_collinearity_epsilon && d3 > curve_collinearity_epsilon)
{
    // Regular care
    //-----------------
    if((d2 + d3)*(d2 + d3) <= m_distance_tolerance * (dx*dx + dy*dy))
    {
        // If the curvature doesn't exceed the distance_tolerance value
        // we tend to finish subdivisions.
        //----------------------
        if(m_angle_tolerance < curve_angle_tolerance_epsilon)
        {
            m_points.add(point_type(x1234, y1234));
            return;
        }
        // Angle & Cusp Condition
        //----------------------
        double a23 = atan2(y3 - y2, x3 - x2);
        da1 = fabs(a23 - atan2(y2 - y1, x2 - x1));
        da2 = fabs(atan2(y4 - y3, x4 - x3) - a23);
        if(da1 >= pi) da1 = 2*pi - da1;
        if(da2 >= pi) da2 = 2*pi - da2;
        if(da1 + da2 < m_angle_tolerance)
        {
            // Finally we can stop the recursion
            //----------------------
            m_points.add(point_type(x1234, y1234));
            return;
        }
        if(m_cusp_limit != 0.0)
        {
            if(da1 > m_cusp_limit)
            {
                m_points.add(point_type(x2, y2));
                return;
            }
            if(da2 > m_cusp_limit)
            {
                m_points.add(point_type(x3, y3));
                return;
```

```
                    }
                }
            }
        }
        else
        {
            if(d2 > curve_collinearity_epsilon)
            {
                // p1,p3,p4 are collinear, p2 is considerable
                //----------------------
                if(d2 * d2 <= m_distance_tolerance * (dx*dx + dy*dy))
                {
                    if(m_angle_tolerance < curve_angle_tolerance_epsilon)
                    {
                        m_points.add(point_type(x1234, y1234));
                        return;
                    }
                    // Angle Condition
                    //----------------------
                    da1 = fabs(atan2(y3 - y2, x3 - x2) - atan2(y2 - y1, x2 - x1));
                    if(da1 >= pi) da1 = 2*pi - da1;
                    if(da1 < m_angle_tolerance)
                    {
                        m_points.add(point_type(x2, y2));
                        m_points.add(point_type(x3, y3));
                        return;
                    }
                    if(m_cusp_limit != 0.0)
                    {
                        if(da1 > m_cusp_limit)
                        {
                            m_points.add(point_type(x2, y2));
                            return;
                        }
                    }
                }
            }
            else
                if(d3 > curve_collinearity_epsilon)
                {
                    // p1,p2,p4 are collinear, p3 is considerable
                    //----------------------
                    if(d3 * d3 <= m_distance_tolerance * (dx*dx + dy*dy))
                    {
```

```
                                if(m_angle_tolerance < curve_angle_tolerance_epsilon)
                                {
                                    m_points.add(point_type(x1234, y1234));
                                    return;
                                }
                                // Angle Condition
                                //----------------------
                                da1 = fabs(atan2(y4 - y3, x4 - x3) - atan2(y3 - y2, x3 - x2));
                                if(da1 >= pi) da1 = 2*pi - da1;
                                if(da1 < m_angle_tolerance)
                                {
                                    m_points.add(point_type(x2, y2));
                                    m_points.add(point_type(x3, y3));
                                    return;
                                }
                                if(m_cusp_limit != 0.0)
                                {
                                    if(da1 > m_cusp_limit)
                                    {
                                        m_points.add(point_type(x3, y3));
                                        return;
                                    }
                                }
                            }
                        }
                        else
                        {
                            // Collinear case
                            //------------------
                            dx = x1234 - (x1 + x4) / 2;
                            dy = y1234 - (y1 + y4) / 2;
                            if(dx*dx + dy*dy <= m_distance_tolerance)
                            {
                                m_points.add(point_type(x1234, y1234));
                                return;
                            }
                        }
                    }
                }
                // Continue subdivision
                //----------------------
                recursive_bezier(x1, y1, x12, y12, x123, y123, x1234, y1234, level + 1);
                recursive_bezier(x1234, y1234, x234, y234, x34, y34, x4, y4, level + 1);
            }
```

```
//------------------------------------------------------------------------
void curve4_div::bezier(double x1, double y1,
                        double x2, double y2,
                        double x3, double y3,
                        double x4, double y4)
{
    m_points.add(point_type(x1, y1));
    recursive_bezier(x1, y1, x2, y2, x3, y3, x4, y4, 0);
    m_points.add(point_type(x4, y4));
}
```

老实说其实这还不是完整的代码，不过这里已经展示了整个算法。余下的部分（类的定义之类的）可以在下面几个文件中找到：

Class curve4_div, files agg_curves.h, agg_curves.cpp.

上面我说要告诉你的那个细节是关于处理共线情况的一段代码：

```
if(da1 < m_angle_tolerance)
{
    m_points.add(point_type(x2, y2));
    m_points.add(point_type(x3, y3));
    return;
}
```

你可以看到，我使用了两个点，而不是一个点（1234）。我也是通过实验得来的。只加一个点会在尖端点产生错误的角度。我不知道数学上这是怎么回事，不过，反正它是 OK 的。

这个类有下面的这些参数：

- -approximation_scale ：最终决定估算的精度。在实际应用中，我们需要从点的世界坐标转换到屏幕坐标，因此总会存在一定的缩放因子。曲线通常是在世界坐标系中处理的，而进行估算时又要转换为像素值。一般看起来会是这样的：
  m_curved.approximation_scale(m_transform.scale());
  这里，m_transform 是一个仿型映射的矩阵，里面包含了所有的转换，包括视点和缩放。
- -angle_tolerance：以弧度来设置。这个值越少，在曲线的转弯处的估算就越精确。不过，如果设置为 0，那么表示完全 不考虑角度条件。
- -cusp_limit：一个以弧度来设置的角度。如果是 0，那么只有真正的尖端(cusp)才会有 bevel cut。如果大于 0，那么它会限制曲线的弯曲度，值越大，曲线锐利的转弯处被切得就越少。一般，这个值不应该大于 10-15 度。

我上面提到过，通过角度来进行估算开销比较大，而且我们也不是总是需要它。我们只是在有曲线有明显的 stroke 或是 contours 的时候才用它。所以，为了优化它，我们可以这样做：

```
double scl = m_transform.scale();
m_curved.approximation_scale(scl);
// Turn off processing of curve cusps
//-----------------
m_curved.angle_tolerance(0);
if(attr.fill_flag)
{
    // Process the fill
    . . .
}
if(attr.stroke_flag)
{
    // Process the stroke
    //----------------------
    // If the *visual* line width is considerable we
    // turn on processing of sharp turns and cusps.
    //----------------------
    if(attr.stroke_width * scl > 1.0)
    {
        m_curved.angle_tolerance(0.2);
    }
    . . .
}
```

这会提升整体的速度。

8.10 Quadric Curves

二次曲线处理起来要简单的得多。我们连 cusp_limit 的判定都不需要，因为共线的情况已经可以处理所有退化的情况：

```
//------------------------------------------------------------------------
void curve3_div::init(double x1, double y1,
                      double x2, double y2,
                      double x3, double y3)
{
    m_points.remove_all();
    m_distance_tolerance = 0.5 / m_approximation_scale;
    m_distance_tolerance *= m_distance_tolerance;
    bezier(x1, y1, x2, y2, x3, y3);
    m_count = 0;
}
//------------------------------------------------------------------------
```

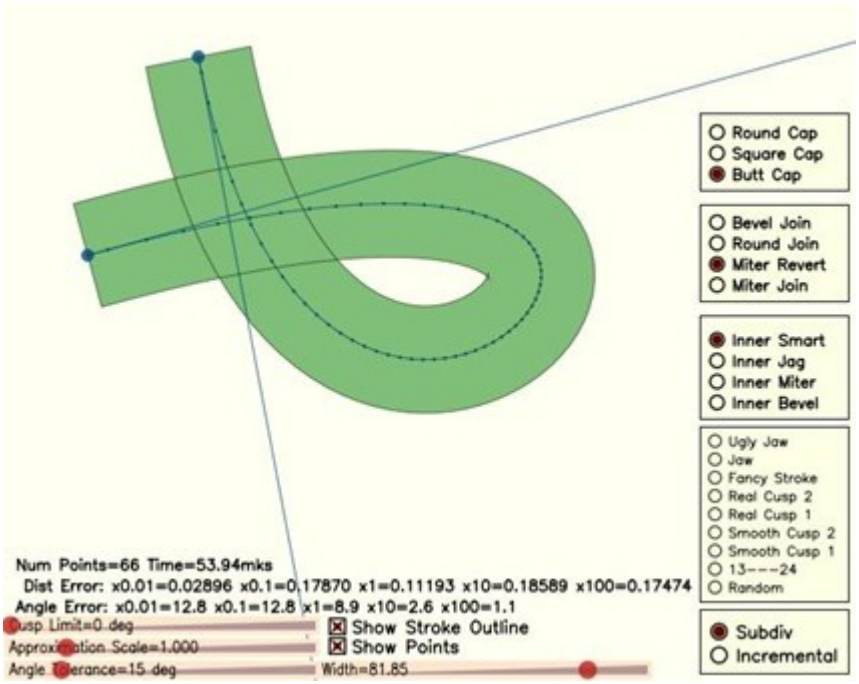
```
void curve3_div::recursive_bezier(double x1, double y1,
                                  double x2, double y2,
                                  double x3, double y3,
                                  unsigned level)
{
    if(level > curve_recursion_limit)
    {
        return;
    }
    // Calculate all the mid-points of the line segments
    //----------------------
    double x12   = (x1 + x2) / 2;
    double y12   = (y1 + y2) / 2;
    double x23   = (x2 + x3) / 2;
    double y23   = (y2 + y3) / 2;
    double x123  = (x12 + x23) / 2;
    double y123  = (y12 + y23) / 2;
    double dx = x3-x1;
    double dy = y3-y1;
    double d = fabs(((x2 - x3) * dy - (y2 - y3) * dx));
    if(d > curve_collinearity_epsilon)
    {
        // Regular care
        //-----------------
        if(d * d <= m_distance_tolerance * (dx*dx + dy*dy))
        {
            // If the curvature doesn't exceed the distance_tolerance value
            // we tend to finish subdivisions.
            //----------------------
            if(m_angle_tolerance < curve_angle_tolerance_epsilon)
            {
                m_points.add(point_type(x123, y123));
                return;
            }
            // Angle & Cusp Condition
            //----------------------
            double da = fabs(atan2(y3 - y2, x3 - x2) - atan2(y2 - y1, x2 - x1));
            if(da >= pi) da = 2*pi - da;
            if(da < m_angle_tolerance)
            {
                // Finally we can stop the recursion
                //----------------------
                m_points.add(point_type(x123, y123));
                return;
```

```
                }
            }
        }
        else
        {
            // Collinear case
            //-----------------
            dx = x123 - (x1 + x3) / 2;
            dy = y123 - (y1 + y3) / 2;
            if(dx*dx + dy*dy <= m_distance_tolerance)
            {
                m_points.add(point_type(x123, y123));
                return;
            }
        }
        // Continue subdivision
        //----------------------
        recursive_bezier(x1, y1, x12, y12, x123, y123, level + 1);
        recursive_bezier(x123, y123, x23, y23, x3, y3, level + 1);
    }
    //------------------------------------------------------------------------
    void curve3_div::bezier(double x1, double y1,
                            double x2, double y2,
                            double x3, double y3)
    {
        m_points.add(point_type(x1, y1));
        recursive_bezier(x1, y1, x2, y2, x3, y3, 0);
        m_points.add(point_type(x3, y3));
    }
```

8.11 Demo Application

你可以从这里下载一个 Win32 下的 Demo 程序

载图如下：



这个程度运行起来很慢，不过这不重要。我使用五个缩放尺度计算了最大的距离和角度误差 :0.01, 0.1, 1, 10, 和 100（这非常耗时）。距离误差是指设置 approximation_scale() 为以上五个值中的一个，计算出最大误差，然后乘以 approximation_scale() 的值。所以，它是屏幕分辨率（像素值）归一化后的最大误差。（原句： So that, it will be the maximal error normalized to the screen resolution (pixels).）

参考曲线是用 Paul Bourke 方法画的，使用的是很小的步进（4096 个点）。

你可以清楚地看到递增方法（incremental method）和细分方法（subdivision method）最大误差的差别。不管 scale 是多少，细分方法的误差基本保持了一致。但递增方法在 scale 为 10 和 100 时误差变小了（But in the incremental method it decreases on the 10x and 100x scales.）。这种行为明显是不对的，因为我们不需要 0.001 像素的错误。这意味着曲线的点太多了。我们需要在误差和点数之间取得平衡。

你可以比较下两个渲染的结果：

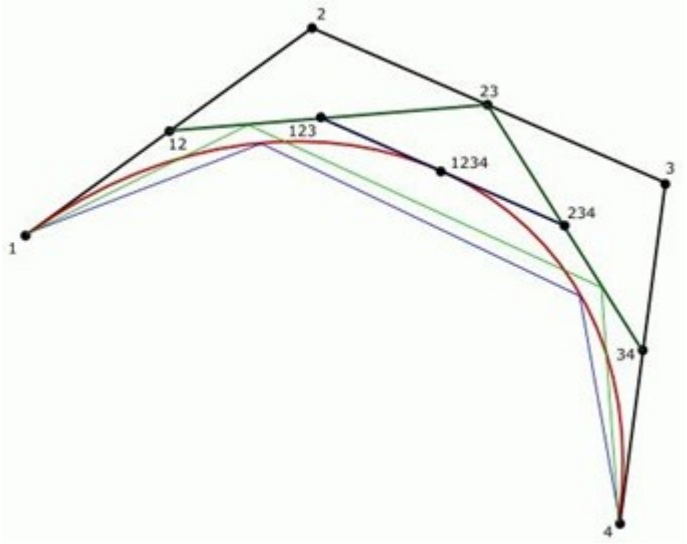
- Incremental Tiger
- Subdivided Tiger

这个结果看起来差别很小，但是如果你把两张图下下来，使用滑动窗口来转换两张图。这两个例子中，我设置精度的目标是为了让两者产生的点的数目一相近。如果你仔细比较这两个图，你会发生细分方法渲染的结果要精确得多。

当然，递增方法也很好，因为它很简单，很快，可以用在那些性能要求很高的地方。一个典型的情形是，用于字体的渲染。字体通常没有多少复杂的曲线，所有的曲线都像弧线（TrueType 字体大部分情况下只用了二次曲线）。

8.12 Update 1

在写这篇文章的时候我已经意识到，添加点 1234 并不够好。In this case the curve is circumscribed and the polyline is escribed. 如果我们添加点 23，我们可以得到更好的结果。假设我们只进行一次细分，比较一下两者的结果：



这里，蓝色的线是我们用点 1234 产生的，而绿色的则是使用点 23 产生的。很明显，绿色线的误差要小一些，在实际中，大约是 1.5 或 2 倍的差别。

8.13 Update 2

在 comp.graphics.algorithms 新闻组，我因为对专著的不熟悉而受到了相当严厉的批评。完整的讨论在这里可以找到，你也可以使用“Adaptive Subdivision of Bezier Curves McSeem”在 http://groups.google.com 里进行搜索。 在网上可以很容易地找到一篇叫“Adaptive forward differencing for rendering curves and surfaces”的文章。里面介绍的方法很好，不过仍然不能解决平滑 stroke 中遇到的所有问题。

但最有意思是信息来自一个昵称是“Just d'FAQs”的人。下面是他的话：

一个很有名的曲线平坦度检测方法比你使用的方法要简单而且可靠。这种方法是基于这样的观察：如果曲线是一个点在端点到端点之间以匀速运动形成的，那么它的控制点会均匀地分布在两个端点之间。因此，我们计算偏移时应该使用中间控制的距离，而不是用线段的距离（原句：Therefore, our measure of how far we deviate from that ideal uses distance of the middle controls, not from the line itself, but from their ideal *arrangement*.)。点 2 应该是点 1 和点 3 的中点 ,点 3 应该是点 2 和点 4 的中点。（译注：这段把我弄晕了……）

这个方法仍然可以改进，因为我们可以消除距离检测中的平方根计算，可以保留 Euclidean metric，但是 taxicab metric 会更快 ,而且一样安全。在这种度量方法中(taxicab metric),(x,y)的长度是 :|x|+|y| 。

反应成代码就像下面这样：

```
if(fabs(x1 + x3 - x2 - x2) +
   fabs(y1 + y3 - y2 - y2) +
   fabs(x2 + x4 - x3 - x3) +
   fabs(y2 + y4 - y3 - y3) <= m_distance_tolerance_manhattan)
{
    m_points.add(point_type(x1234, y1234));
    return;
}
```

这种方法不需要强制进行第一次细分，而且在任何情况下都很健壮。作者坚持认为这种方法已经够好了，不过我经过一些额外的实验后发现，结果和递增方法有些类似。这种估算没有在点数和差误之间平衡好。但在共线的情况下很棒。我把它融合到我的代码中了，现在看起来，我可以停止这方面的研究了。这个方法需要几种不同的度量，“Manhattan”（或者叫“taxicab”），我们需要将 tolerance 像下面这样规范化：

```
m_distance_tolerance_square = 0.5 / m_approximation_scale;
m_distance_tolerance_square *= m_distance_tolerance_square;
m_distance_tolerance_manhattan = 4.0 / m_approximation_scale;
```

在共线的情况下，这种方法比 Timothee Groleau 的方法好。

完整的代码在这里: curve4_div, 文件：agg_curves.h, agg_curves.cpp.