# Using WinAPI to Render Text

## A simple class to extract True Type glyphs using WinAPI GetGlyphOutline()

There are many questions about drawing of high quality text. **AGG** provides a class that produces vector text of a fixed typeface (very primitive, ANSI 7-bit character set). I added this class just to have a simple mechanism to draw text in demo examples. It has a propriatory data format and isn't worth further developing.

The good news is that you can use any available library or API to extract glyphs and render them with **AGG**. One can say if the glyph format consists of line segments, conic and cubic bezier curves, it's possible to render it with **AGG**. All available converters and transformers are applicable, as well as all the renderers. For example, you can draw an outlined text with conv_stroke, or change the font weight (make it bolder or lighter) using conv_contour. You can also render it with gradients or fill the glyphs with images.

It's a long story how to integrate **AGG** with different font engines, like FreeType. This example demonstrates a simplest way to use Windows API to extract the glyphs and to render text with **AGG**. It calls `GetGlyphOutline()`, extracts native curved contours, applies conv_curve and renders the text with any available scanline renderers.

It works relatively slow, not only because each glyph is being rasterized every time (no cache mechanism), but also because `GetGlyphOutline()` works terribly slow. More than a half of total time is spent in `GetGlyphOutline()`. It would be a good solution to cache the glyphs or even pre-rendered bitmaps; it could speed up rendering vastly, but it's out of this topic.

Below is a simple console application that creates `agg_test.ppm` file (the simplest possible RGB bitmap file). The file isn't natively supported by **Microsoft** Windows, but there are many viewers and converters that can work with it, for example, IrfanView (www.irfanview.com).

Class `tt_glyph` is not included into **AGG** because it depends on the Windows API (`#include <windows.h>`), while the main part of **AGG** is supposed to be fully platform independent.

To build the example you need to indicate the **AGG** include directory and to add to the project files agg_curves.cpp and agg_rasterizer_scanline_aa.cpp.

```
#include <stdio.h>
#include <string.h>
#include "agg_pixfmt_rgb24.h"
#include "agg_renderer_base.h"
#include "agg_renderer_scanline.h"
#include "agg_rasterizer_scanline_aa.h"
#include "agg_scanline_p.h"
#include "agg_conv_curve.h"
```

```
#include <windows.h>

namespace agg
{

    //========================================================================
    class tt_glyph
    {
        enum { buf_size = 16384-32 };

    public:
        ~tt_glyph();
        tt_glyph();

        // Set the created font and the "flip_y" flag.
        //------------------------
        void font(HDC dc, HFONT f) { m_dc = dc; m_font = f; }
        void flip_y(bool flip) { m_flip_y = flip; }

        bool glyph(unsigned chr, bool hinted = true);

        // The following functions can be called after glyph()
        // and return the respective values of the
        // GLYPHMETRICS structure.
        //------------------------
        int      origin_x() const { return m_origin_x; }
        int      origin_y() const { return m_origin_y; }
        unsigned width()    const { return m_width;    }
        unsigned height()   const { return m_height;   }
        int      inc_x()    const { return m_inc_x;    }
        int      inc_y()    const { return m_inc_y;    }

        // Set the starting point of the Glyph
        //------------------------
        void start_point(double x, double y)
        {
            m_start_x = x;
            m_start_y = y;
        }

        // Vertex Source Interface
        //------------------------
        void rewind(unsigned)
        {
            m_cur_vertex = m_vertices;
            m_cur_flag = m_flags;
        }

        unsigned vertex(double* x, double* y)
        {
            *x = m_start_x + *m_cur_vertex++;
            *y = m_start_y + *m_cur_vertex++;
            return *m_cur_flag++;
        }

    private:
        HDC           m_dc;
        HFONT         m_font;
```

```
    char*         m_gbuf;
    int8u*        m_flags;
    double*       m_vertices;
    unsigned      m_max_vertices;
    const int8u*  m_cur_flag;
    const double* m_cur_vertex;
    double        m_start_x;
    double        m_start_y;
    MAT2          m_mat2;

    int      m_origin_x;
    int      m_origin_y;
    unsigned m_width;
    unsigned m_height;
    int      m_inc_x;
    int      m_inc_y;

    bool     m_flip_y;
};



tt_glyph::~tt_glyph()
{
    delete [] m_vertices;
    delete [] m_flags;
    delete [] m_gbuf;
}


tt_glyph::tt_glyph() :
    m_dc(0),
    m_font(0),
    m_gbuf(new char [buf_size]),
    m_flags(new int8u [256]),
    m_vertices(new double[512]),
    m_max_vertices(256),
    m_cur_flag(m_flags),
    m_cur_vertex(m_vertices),
    m_start_x(0.0),
    m_start_y(0.0),
    m_flip_y(false)
{
    m_vertices[0] = m_vertices[1] = 0.0;
    m_flags[0] = path_cmd_stop;
    memset(&m_mat2, 0, sizeof(m_mat2));
    m_mat2.eM11.value = 1;
    m_mat2.eM22.value = 1;
}



static inline double fx_to_dbl(const FIXED& p)
{
    return double(p.value) + double(p.fract) * (1.0 / 65536.0);
}

static inline FIXED dbl_to_fx(double d)
{
    int l;
```

```
        l = long(d * 65536.0);
        return *(FIXED*)&l;
    }


    bool tt_glyph::glyph(unsigned chr, bool hinted)
    {
        m_vertices[0] = m_vertices[1] = 0.0;
        m_flags[0] = path_cmd_stop;
        rewind(0);

        if (m_font == 0) return false;

#ifndef GGO_UNHINTED         // For compatibility with old SDKs.
#define GGO_UNHINTED 0x0100
#endif

        int unhinted = hinted ? 0 : GGO_UNHINTED;

        GLYPHMETRICS gm;
        int total_size = GetGlyphOutline(m_dc,
                                         chr,
                                         GGO_NATIVE | unhinted,
                                         &gm,
                                         buf_size,
                                         (void*)m_gbuf,
                                         &m_mat2);

        if (total_size < 0) return false;

        m_origin_x = gm.gmptGlyphOrigin.x;
        m_origin_y = gm.gmptGlyphOrigin.y;
        m_width    = gm.gmBlackBoxX;
        m_height   = gm.gmBlackBoxY;
        m_inc_x    = gm.gmCellIncX;
        m_inc_y    = gm.gmCellIncY;

        if (m_max_vertices <= total_size / sizeof(POINTFX))
        {
            delete [] m_vertices;
            delete [] m_flags;
            m_max_vertices = total_size / sizeof(POINTFX) + 256;
            m_flags = new int8u [m_max_vertices];
            m_vertices = new double [m_max_vertices * 2];
        }

        const char* cur_glyph = m_gbuf;
        const char* end_glyph = m_gbuf + total_size;

        double* vertex_ptr = m_vertices;
        int8u*  flag_ptr   = m_flags;

        while(cur_glyph < end_glyph)
        {
            const TTPOLYGONHEADER* th = (TTPOLYGONHEADER*)cur_glyph;

            const char* end_poly = cur_glyph + th->cb;
            const char* cur_poly = cur_glyph + sizeof(TTPOLYGONHEADER);

            *vertex_ptr++ = fx_to_dbl(th->pfxStart.x);
            *vertex_ptr++ = m_flip_y ?
```

```
                        -fx_to_dbl(th->pfxStart.y):
                         fx_to_dbl(th->pfxStart.y);
        *flag_ptr++   = path_cmd_move_to;

        while(cur_poly < end_poly)
        {
            const TTPOLYCURVE* pc = (const TTPOLYCURVE*)cur_poly;

            if (pc->wType == TT_PRIM_LINE)
            {
                int i;
                for (i = 0; i < pc->cpfx; i++)
                {
                    *vertex_ptr++ = fx_to_dbl(pc->apfx[i].x);
                    *vertex_ptr++ = m_flip_y ?
                                        -fx_to_dbl(pc->apfx[i].y):
                                         fx_to_dbl(pc->apfx[i].y);
                    *flag_ptr++   = path_cmd_line_to;
                }
            }

            if (pc->wType == TT_PRIM_QSPLINE)
            {
                int u;
                for (u = 0; u < pc->cpfx - 1; u++)  // Walk through points in spline
                {
                    POINTFX pnt_b = pc->apfx[u];    // B is always the current point
                    POINTFX pnt_c = pc->apfx[u+1];

                    if (u < pc->cpfx - 2)           // If not on last spline, compute C
                    {
                        // midpoint (x,y)
                        *(int*)&pnt_c.x = (*(int*)&pnt_b.x + *(int*)&pnt_c.x) / 2;
                        *(int*)&pnt_c.y = (*(int*)&pnt_b.y + *(int*)&pnt_c.y) / 2;
                    }

                    *vertex_ptr++ = fx_to_dbl(pnt_b.x);
                    *vertex_ptr++ = m_flip_y ?
                                        -fx_to_dbl(pnt_b.y):
                                         fx_to_dbl(pnt_b.y);
                    *flag_ptr++   = path_cmd_curve3;

                    *vertex_ptr++ = fx_to_dbl(pnt_c.x);
                    *vertex_ptr++ = m_flip_y ?
                                        -fx_to_dbl(pnt_c.y):
                                         fx_to_dbl(pnt_c.y);
                    *flag_ptr++   = path_cmd_curve3;
                }
            }
            cur_poly += sizeof(WORD) * 2 + sizeof(POINTFX) * pc->cpfx;
        }
        cur_glyph += th->cb;
        *vertex_ptr++ = 0.0;
        *vertex_ptr++ = 0.0;
        *flag_ptr++   = path_cmd_end_poly | path_flags_close | path_flags_ccw;
    }

    *vertex_ptr++ = 0.0;
    *vertex_ptr++ = 0.0;
    *flag_ptr++   = path_cmd_stop;
```

```
        return true;
    }

}




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;
}




template<class Rasterizer, class Renderer, class Scanline, class CharT>
void render_text(Rasterizer& ras, Renderer& ren, Scanline& sl,
                 agg::tt_glyph& gl, double x, double y, const CharT* str,
                 bool hinted = true)
{
    // The minimal pipeline is the curve converter. Of course, there
    // any other transformations are applicapble, conv_stroke<>, for example.
    // If there are other thransformations, it probably makes sense to
    // turn off the hints (hinted=false), i.e., to use unhinted glyphs.
    //--------------------------
    agg::conv_curve<agg::tt_glyph> curve(gl);
    while(*str)
    {
        gl.start_point(x, y);
        gl.glyph(*str++, hinted);
        ras.add_path(curve);
        agg::render_scanlines(ras, sl, ren);
        x += gl.inc_x();
        y += gl.inc_y();
    }
}
```

```
int main()
{
    // Create the rendering buffer
    //------------------------
    unsigned char* buffer = new unsigned char[frame_width * frame_height * 3];
    agg::rendering_buffer rbuf(buffer,
                               frame_width,
                               frame_height,
                               -frame_width * 3);

    // Create the renderers, the rasterizer, and the scanline container
    //------------------------
    agg::pixfmt_rgb24 pixf(rbuf);
    agg::renderer_base<agg::pixfmt_rgb24> rbase(pixf);
    agg::renderer_scanline_aa_solid<agg::renderer_base<agg::pixfmt_rgb24> > ren(rbase);
    agg::rasterizer_scanline_aa<> ras;
    agg::scanline_p8 sl;

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


    // Font parameters
    //------------------------
    int  fontHeight      = 60;  // in Pixels in this case
    int  fontWidth       = 0;
    int  iAngle          = 0;
    bool bold            = true;
    bool italic          = true;
    const char* typeFace = "Times New Roman";


    // I'm not sure how to deal with those sneaky WinGDI functions correctly,
    // so, please correct me if there's something wrong.
    // I'm not sure if I need to call ReleaseDC() for the screen.
    //----------------------------------

    HFONT font = ::CreateFont(fontHeight,           // height of font
                              fontWidth,            // average character width
                              iAngle,               // angle of escapement
                              iAngle,               // base-line orientation angle
                              bold ? 700 : 400,     // font weight
                              italic,               // italic attribute option
                              FALSE,                // underline attribute option
                              FALSE,                // strikeout attribute option
                              ANSI_CHARSET,         // character set identifier
                              OUT_DEFAULT_PRECIS,   // output precision
                              CLIP_DEFAULT_PRECIS,  // clipping precision
                              ANTIALIASED_QUALITY,  // output quality
                              FF_DONTCARE,          // pitch and family
                              typeFace);            // typeface name
    if(font)
    {
        HDC dc = ::GetDC(0);
        if(dc)
        {
            HGDIOBJ old_font = ::SelectObject(dc, font);
            agg::tt_glyph gl;
            gl.font(dc, font);

            ren.color(agg::rgba8(0,0,128));
```

```
            render_text(ras, ren, sl, gl, 10, 100, "Hello, World!");

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

            ::SelectObject(dc, old_font);
            ::ReleaseDC(0, dc);
        }
        ::DeleteObject(font);
    }


    delete [] buffer;
    return 0;
}







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;
}




template<class Rasterizer, class Renderer, class Scanline, class CharT>
void render_text(Rasterizer& ras, Renderer& ren, Scanline& sl,
                 agg::tt_glyph& gl, double x, double y, const CharT* str,
                 bool hinted = true)
{
    // The minimal pipeline is the curve converter. Of course, there
    // any other transformations are applicapble, conv_stroke<>, for example.
    // If there are other thransformations, it probably makes sense to
    // turn off the hints (hinted=false), i.e., to use unhinted glyphs.
    //--------------------------
    agg::conv_curve<agg::tt_glyph> curve(gl);
```

```
    while(*str)
    {
        gl.start_point(x, y);
        gl.glyph(*str++, hinted);
        ras.add_path(curve);
        agg::render_scanlines(ras, sl, ren);
        x += gl.inc_x();
        y += gl.inc_y();
    }
}



int main()
{
    // Create the rendering buffer
    //------------------------
    unsigned char* buffer = new unsigned char[frame_width * frame_height * 3];
    agg::rendering_buffer rbuf(buffer,
                               frame_width,
                               frame_height,
                               -frame_width * 3);

    // Create the renderers, the rasterizer, and the scanline container
    //------------------------
    agg::pixfmt_rgb24 pixf(rbuf);
    agg::renderer_base<agg::pixfmt_rgb24> rbase(pixf);
    agg::renderer_scanline_aa_solid<agg::renderer_base<agg::pixfmt_rgb24> > ren(rbase);
    agg::rasterizer_scanline_aa<> ras;
    agg::scanline_p8 sl;

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


    // Font parameters
    //------------------------
    int  fontHeight      = 60;  // in Pixels in this case
    int  fontWidth       = 0;
    int  iAngle          = 0;
    bool bold            = true;
    bool italic          = true;
    const char* typeFace = "Times New Roman";


    // I'm not sure how to deal with those sneaky WinGDI functions correctly,
    // so, please correct me if there's something wrong.
    // I'm not sure if I need to call ReleaseDC() for the screen.
    //----------------------------------

    HFONT font = ::CreateFont(fontHeight,           // height of font
                              fontWidth,            // average character width
                              iAngle,               // angle of escapement
                              iAngle,               // base-line orientation angle
                              bold ? 700 : 400,     // font weight
                              italic,               // italic attribute option
                              FALSE,                // underline attribute option
                              FALSE,                // strikeout attribute option
                              ANSI_CHARSET,         // character set identifier
                              OUT_DEFAULT_PRECIS,   // output precision
                              CLIP_DEFAULT_PRECIS,  // clipping precision
                              ANTIALIASED_QUALITY,  // output quality
```

```
                                FF_DONTCARE,           // pitch and family
                                typeFace);             // typeface name
    if(font)
    {
        HDC dc = ::GetDC(0);
        if(dc)
        {
            HGDIOBJ old_font = ::SelectObject(dc, font);
            agg::tt_glyph gl;
            gl.font(dc, font);

            ren.color(agg::rgba8(0,0,128));
            render_text(ras, ren, sl, gl, 10, 100, "Hello, World!");

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

            ::SelectObject(dc, old_font);
            ::ReleaseDC(0, dc);
        }
        ::DeleteObject(font);
    }


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

And the result:


Beautiful, isn't it? In fact, it looks much better than the native text rendered in Windows. To compare the quality, run WordPad, type “Hello, World!” and change font to “Times New Roman”, set Bold, Italic and size of 39 points.

Here's an enlarged fragment of a glyph to compare the quality.