Image Parallelogram Transformations

Using perspective transformations to simulate the functionality of WinAPI PlgBlt()

The declaration of PlgBlt() is:

```
BOOL PlgBlt(
  HDC hdcDest,           // handle to destination device context
  CONST POINT *lpPoint,  // vertices of destination parallelogram
  HDC hdcSrc,            // handle to source device context
  int nXSrc,             // x-coord. of upper-left corner of source rectangle.
  int nYSrc,             // y-coord. of upper-left corner of source rectangle.
  int nWidth,            // width of source rectangle
  int nHeight,           // height of source rectangle
  HBITMAP hbmMask,       // handle to bitmask
  int xMask,             // x-coord. of upper-left corner of bitmask rectangle.
  int yMask              // y-coord. of upper-left corner of bitmask rectangle.
);
```

Here the most important argument is:

lpPoint

- Pointer to an array of three points in logical space that identify three corners of the destination parallelogram. The upper-left corner of the source rectangle is mapped to the first point in this array, the upper-right corner to the second point in this array, and the lower-left corner to the third point. The lower-right corner of the source rectangle is mapped to the implicit fourth point in the parallelogram.

It means that this function can apply arbitrary affine transformations to the image. **Anti-Grain Geometry** can do that too, but there's a problem with proper calculating of the affine transformation matrix. It really is tricky.

In **AGG** there are good news, bad news, and then good news again. The good news is that you can use the perspective transformations that in general can transform an arbitrary convex qudrangle to another convex quadrangle, particularly, a rectangle to an arbitrary parallelogram.

The bad news is that in general case, the perspective transformations work much slower than the affine ones. It's because the image transformations use the “scanline” approach. You take your destination scanline (a row of pixels in the destination canvas), then apply the **reverse** transformations to each pixel and pick up the source pixel possibly considering a filter (bilinear, bicubic, etc…). In case of affine transformations we don't have to calculete every point directly. Instead, we can calculate only two points for each scanline (begin and end) and use a bresenham-like linear interpolator that works in integer coordinates, thus very fast. But the restriction is that the transformations must be linear and parellel. It means that any straight line must remain straight after applying the transformation, and any two parallel lines must remain parallel. In case of perspective transformations it is not so (they are not parallel), and we cannot use linear interpolation.

The good news again is that the parallelogram case of the perspective transformations is linear and parallel, so, the the linear interpolation is perfectly applicable and it will work as fast as the image affine transformations.

To demonstrate it we modify the **AGG** example `image_perspective.cpp` (it can be found in `agg2/examples/`). Just replace the code of `image_perspective.cpp` to the following:

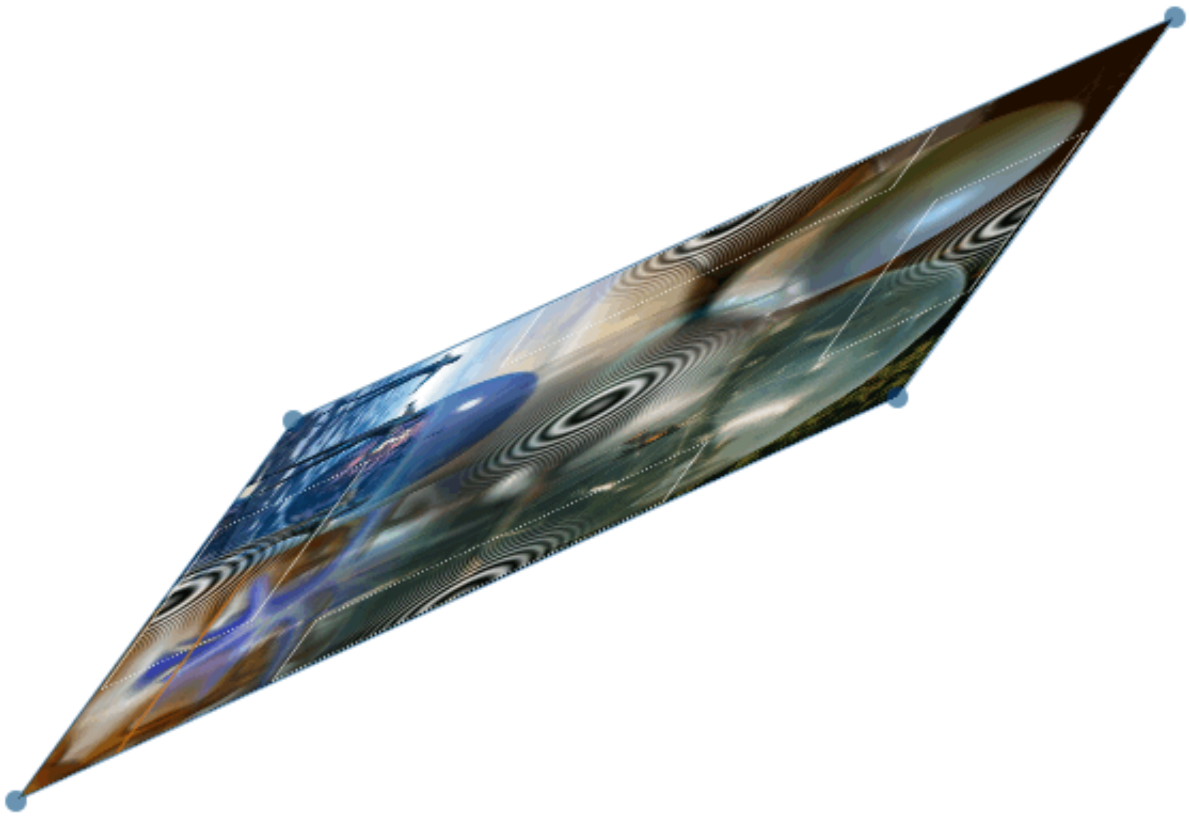
```
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>
#include "agg_basics.h"
#include "agg_rendering_buffer.h"
#include "agg_rasterizer_scanline_aa.h"
#include "agg_scanline_u.h"
#include "agg_renderer_scanline.h"
#include "agg_path_storage.h"
#include "agg_conv_transform.h"
#include "agg_trans_bilinear.h"
#include "agg_trans_perspective.h"
#include "agg_span_interpolator_trans.h"
#include "agg_span_interpolator_linear.h"
#include "agg_pixfmt_rgb24.h"
#include "agg_span_image_filter_rgb24.h"
#include "ctrl/agg_rbox_ctrl.h"
#include "platform/agg_platform_support.h"
#include "interactive_polygon.h"


enum { flip_y = true };

agg::rasterizer_scanline_aa<> g_rasterizer;
agg::scanline_u8  g_scanline;
double            g_x1 = 0;
double            g_y1 = 0;
double            g_x2 = 0;
double            g_y2 = 0;

class the_application : public agg::platform_support
{
public:
    typedef agg::pixfmt_bgr24 pixfmt;
    typedef agg::renderer_base<pixfmt> renderer_base;
    typedef agg::renderer_scanline_aa_solid<renderer_base> renderer_solid;

    agg::interactive_polygon   m_triangle;

    the_application(agg::pix_format_e format, bool flip_y) :
        agg::platform_support(format, flip_y),
        m_triangle(4, 5.0)
    {
    }


    virtual void on_init()
    {
        g_x1 = 0.0;
```

```
        g_y1 = 0.0;
        g_x2 = rbuf_img(0).width();
        g_y2 = rbuf_img(0).height();
        double dx = width()  / 2.0 - (g_x2 - g_x1) / 2.0;
        double dy = height() / 2.0 - (g_y2 - g_y1) / 2.0;
        m_triangle.xn(0) = g_x1 + dx;
        m_triangle.yn(0) = g_y1 + dy;
        m_triangle.xn(1) = g_x2 + dx;
        m_triangle.yn(1) = g_y1 + dy;
        m_triangle.xn(2) = g_x2 + dx;
        m_triangle.yn(2) = g_y2 + dy;
        m_triangle.xn(3) = g_x1 + dx;
        m_triangle.yn(3) = g_y2 + dy;
    }

    virtual void on_draw()
    {
        // Calculate the 4-th point of the parallelogram
        m_triangle.xn(3) = m_triangle.xn(0) +
                           (m_triangle.xn(2) - m_triangle.xn(1));

        m_triangle.yn(3) = m_triangle.yn(0) +
                           (m_triangle.yn(2) - m_triangle.yn(1));

        pixfmt pixf(rbuf_window());
        renderer_base rb(pixf);
        renderer_solid r(rb);
        rb.clear(agg::rgba(1, 1, 1));

        g_rasterizer.clip_box(0, 0, width(), height());

        typedef agg::span_allocator<agg::rgba8> span_alloc_type;
        span_alloc_type sa;

        agg::trans_perspective tr(m_triangle.polygon(),
                                  g_x1, g_y1, g_x2, g_y2);

        if(tr.is_valid())
        {

            //=================== The trick with interpolator.

            // ------- Slow variant
            // span_interpolator_trans is a general purpose interpolator.
            // It calls the Transformer::transform() for each point of the
            // scanline, thus, it's slow. But it can be used with any
            // kind of transformations, linear or non-linear.
            //----------------------------
            //typedef agg::span_interpolator_trans<agg::trans_perspective>
            //    interpolator_type;



            // ------- Fast variant
            // span_interpolator_linear is an accelerated version of the general
            // purpose one, span_interpolator_trans. It calculates
            // actual coordinates only for the beginning and the ending points
            // of the span. But the transformations must be linear and parallel,
            // that is, any straight line must remain straight after applying the
            // transformation, and any two parallel lines must remain parallel.
            // It's not sutable for perspective transformations in general
```

```
            // (they are not parallel), but quite OK for this particular case,
            // i.e., parallelogram transformations.
            //----------------------------
            typedef agg::span_interpolator_linear<agg::trans_perspective>
                interpolator_type;

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


            interpolator_type interpolator(tr);

            // "hardcoded" bilinear filter
            //------------------------------------------
            typedef agg::span_image_filter_rgb24_bilinear<agg::order_bgr24,
                                                          interpolator_type>
                span_gen_type;

            typedef agg::renderer_scanline_aa<renderer_base, span_gen_type>
                renderer_type;

            span_gen_type sg(sa,
                             rbuf_img(0),
                             agg::rgba(1, 1, 1, 0),
                             interpolator);

            renderer_type ri(rb, sg);

            g_rasterizer.reset();
            g_rasterizer.move_to_d(m_triangle.xn(0), m_triangle.yn(0));
            g_rasterizer.line_to_d(m_triangle.xn(1), m_triangle.yn(1));
            g_rasterizer.line_to_d(m_triangle.xn(2), m_triangle.yn(2));
            g_rasterizer.line_to_d(m_triangle.xn(3), m_triangle.yn(3));

            agg::render_scanlines(g_rasterizer, g_scanline, ri);
        }


        //--------------------------
        // Render the "quad" tool and controls
        g_rasterizer.add_path(m_triangle);
        r.color(agg::rgba(0, 0.3, 0.5, 0.6));
        agg::render_scanlines(g_rasterizer, g_scanline, r);
        //--------------------------
    }



    virtual void on_mouse_button_down(int x, int y, unsigned flags)
    {
        if(flags & agg::mouse_left)
        {
            if(m_triangle.on_mouse_button_down(x, y))
            {
                force_redraw();
            }
        }
    }


    virtual void on_mouse_move(int x, int y, unsigned flags)
    {
```

```
        if(flags & agg::mouse_left)
        {
            if(m_triangle.on_mouse_move(x, y))
            {
                force_redraw();
            }
        }
        if((flags & agg::mouse_left) == 0)
        {
            on_mouse_button_up(x, y, flags);
        }
    }


    virtual void on_mouse_button_up(int x, int y, unsigned flags)
    {
        if(m_triangle.on_mouse_button_up(x, y))
        {
            force_redraw();
        }
    }

};


int agg_main(int argc, char* argv[])
{
    the_application app(agg::pix_format_bgr24, flip_y);
    app.caption("AGG Example. Image Perspective Transformations");

    const char* img_name = "spheres";
    if(argc >= 2) img_name = argv[1];
    if(!app.load_img(0, img_name))
    {
        char buf[256];
        if(strcmp(img_name, "spheres") == 0)
        {
            sprintf(buf, "File not found: %s%s. Download http://www.antigrain.com/%s%s\n"
                         "or copy it from another directory if available.",
                    img_name, app.img_ext(), img_name, app.img_ext());
        }
        else
        {
            sprintf(buf, "File not found: %s%s", img_name, app.img_ext());
        }
        app.message(buf);
        return 1;
    }

    if(app.init(600, 600, agg::window_resize))
    {
        return app.run();
    }
    return 1;
}
```

There is a screenshot:



NOTE
The arcticle is actually outdated. Now class trans_affine has methods to calculate an affine matrix that transforms a parellelogram to another one, a rectangle to a parellelogram, and a parellelogram to a rectangle. See `agg2/examples/image_perspective.cpp`. However, the above material is useful because it helps understand better the **AGG** concepts.