Omnidirectional image transformation on graphics hardware

v1.3 02-16-2006

transformer

Motivation – why use graphics hardware

Graphics hardware is fast in the first place. I could run transformation you see on the image above (simple unwrapping with four samples per pixel) as fast as 527 frames per second on NVidia GeForce 6600 on target buffer resolution 1024x768 and 32 bits per pixel when switched off vertical synchronization.

But how really fast they are? Pentium 4, running on 3 GHZ has theoretical power of 6 GFLOPS and it‘s memory transfers are as fast as 5.96 GB/sec (peak). GeForce 6800 Ultra has observed (meassured by long shader, consisting of nothing but multiply instructions) power of **40 GFLOPS**, with it‘s memory bandwidth of 35.2 GB/sec (peak).

My first shader had 6 add (weight 1), 5 mul (weight 2) and 1 sin and 1 cos (weight 8):

*527 * 1024 * 768 * (6 * 1 + 5 * 2 + 2 * 8) = 13,26* GFLOPS!

(it wasn‘t real benchmark since there were some processes running in background and i was still measuring extra time of page flip, etc ...)

And more – Moore‘s law for CPU‘s says performance after year is 1.5 times higher. For GPU‘s it‘s 2 times.

Also – all shader operations are performed in floating point so the results are precise. Current graphics hardware can work with IEEE754 32bit floating point buffers so we don‘t have to use traditional open-gl‘s 8 bits per channel any more.

Library

The goal was to write some C++ library, working with DigiLib to perform transformations of recorded AMI meetings. The result is a dozen of classes for work with OpenGL shaders and class for geometric transformation. It takes any OpenGL texture as an input and renders rectangle with transformed image. The library contains classes for work with mechanisms for rendering to texture (namely pixel-buffers and framebuffer-objects) so the output image can be rendered into texture and sent to next OpenGL processing. (or it can be read back by standard OpenGl‘s glReadPixels() (version 1.3 adds supports for asynchronous data transfers, which (especcialy with PCI-express board) proved to be a lot faster) and written on the HDD)

Algorithms

The algorithms are based on Igor Potůček‘s *Omni-directional image transformation* paper. The simple unwrap algorithm was used without any modifications, but the algorithm, taking mirror profile into account was slightly optimised for GPU‘s.

Library files description

Library is distributed in release and debug version, it consists of files:

CallStack.h

- call stack guard (tracks function call history, usable with _ASSERTE macro and for debugging)

GLState.h

- open gl state guard (used to speed up open-gl state changes by remembering most commonly used ones)

OpenGL20.h

- open gl 2.0 extensions (class with functions for retrieval extension function adresses, header also contains their headers)

OpenGLDrv.h

- open gl driver, used by CTrUt (can be used individually as well)

RenderBuffer.h

- classes for open gl off-screen rendering buffer mechanisms. Currently implemented WGL_ARB_pixel_buffer (windows gl extension, exists in XGL form for linux as well, but it‘s not implemented here) GL_ARB_framebuffer_object (portable version, but not supported on all cards)

RenderPath.h

- mechanism for definig render paths. There are three parameter strings, first one contains required extensions, the second one contains supported vendors (or „*“ for any) and the third one contain card model list. Cards names are preceded with + prefix meaning to add this card explicitly to list (for example card of different vendor which happens to be compatible) or – prefix meaning card of this vendor is one of not compatible ones.

Shader.h

- file with implementation of three classes to handle three mechanisms of shaders first one is for GL_ARB_vertex_program or GL_ARB_fragment_program (low level asm-like language), the second one is for GL_ARB_vertex_shader or GL_ARB_fragment_shader (high level c-like language) and the third one is for OpenGL 2.0 core shading language (the same one as used for GL_ARB_*_shader)

Transform.h

- file with transformation class interface

TrUt.h

- transformation utility. Very simple interface, used to control the whole library using minimum of functions. It‘s described on the end of the document.

VertexBufferObject.h

- class for VBO‘s. They are essentialy blocks of memory, allocated directly on the graphics card, making them very fast. They are used in geometrical transformation (there are two versions, one calculates reflection vectors on per-pixel basis, the other one on per-vertex basis, which proved to be faster though needs highly tesselated mesh to work properly) With VBO‘s we don‘t have to worry about hundreds of thousands polygons, it still runs smooth.

Then there are **debug** and **release** folders with .lib and .dll files.

There‘s an example, containing code for work with all the library functionality and for transfers between ImageStruct and OpenGL. It‘s all in **TransformLib_Test_glut.cpp**.

Shaders description

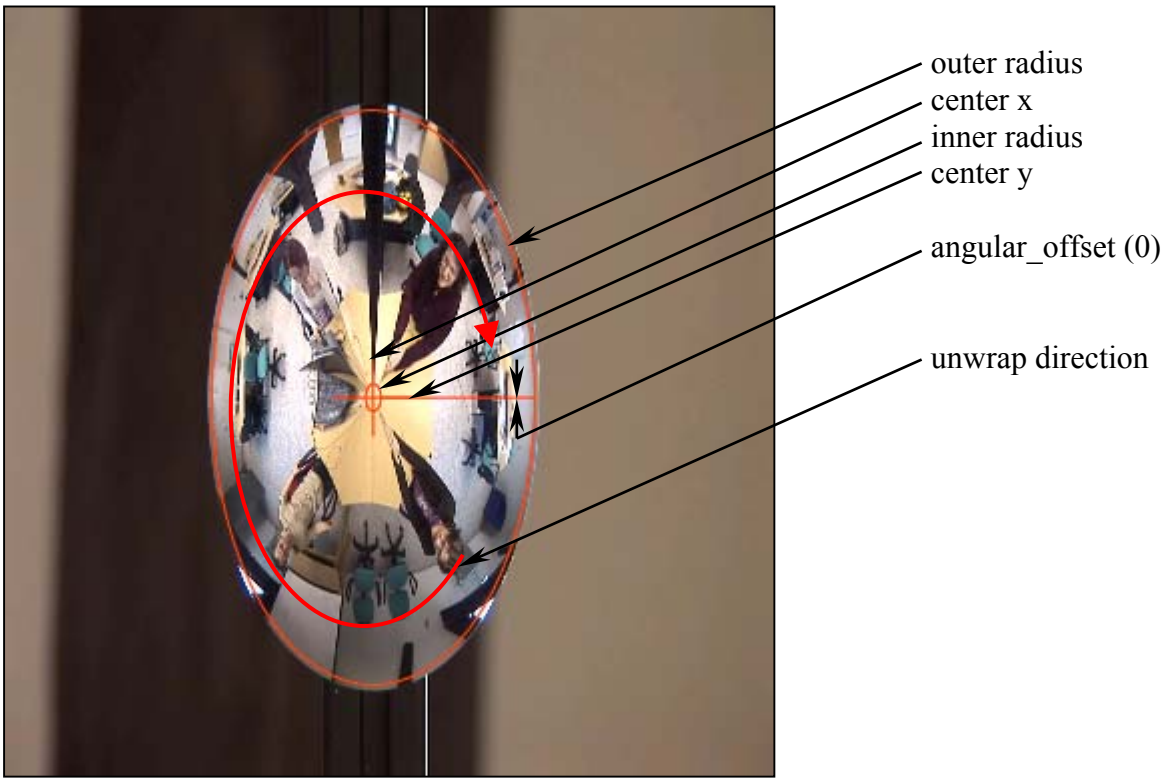
Simple transformation shaders

They must be named simple*, for example simple_4tap_glsl, which means simple transformation, 4-tap filter, OpenGL shading language (language shader source code is written in)

They has two parameters, each is vector of four floats:

$$mirror\_area = \{r\_inner, r\_outer, center\_x, center\_y\}$$
$$offset\_ratio = \{angular\_offset, image\_ratio, pixel\_w, pixel\_h\}$$

The first one is called "mirror_area" and it contains *inner radius* (typically some small value, like 0.01), *outer radius* (> inner radius and < 0.5), *horizontal center of mirror* in texture space (0.5) and *vertical center of mirror* in texture space (0.5). The second one is called "offset_ratio" and contains *angular offset* in radians (angle is measurred with positive-x half-axis, rising counterclockwise, whereby image is unwrapped in clockwise direction, which means image part, corresponding with angle 0 is on the left, angle -2π is on the right). Next components of "offset_ratio" are *image side ratio*, defined as *image width* / *image height* (all sizes are meassured in image width multiples), *pixel width* and *pixel height* (defined as 1 / *image width* or 1 / *image height* respectively) You can see transformation parameters marked by red lines on following image:



Simple transformation is considerably fast (transformation with source resolution 1440×1080 and destination resolution 1024×768 and four samples, each of them bilinearly filtered, of course, can be as fast as ~1.5 miliseconds on GeForce 6600), but it has poor quality with H3G mirror which doesn't have uniform radial resolution. (with the new mirror, transformation quality should be sufficient as you can see on the following image)

result of simple transformation

Geometric transformation shaders

They must be named geom*, for example geom_glsl, which means geometric transformation shader, written in OpenGL shading language.

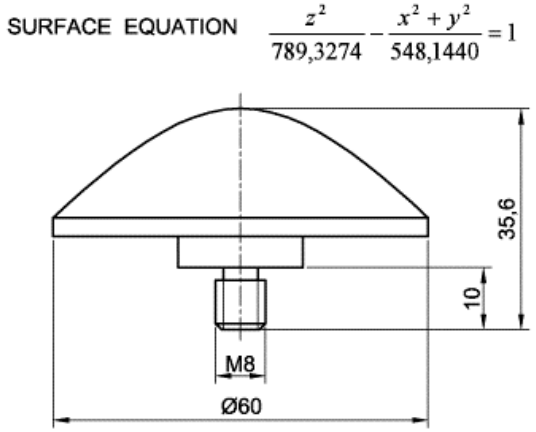
They has the same two parameters as simple transformation shaders ("mirror_area" and "offset_ratio") with the same content. Next there are three extra parameters:

$$mirror\_geom = \left\{ \frac{a^2}{b^2}, \frac{1}{\Delta^2}, \frac{2\sqrt{a^2 + b^2}}{\Delta}, -\left(a^2 + b^2\right) \right\}$$

$$mirror\_geom2 = \left\{ a^2, \sqrt{a^2 + b^2}, \frac{b^2}{a^2}, b^4/a^2 \right\}$$

$$geom\_offset = \{ y\_bottom, y\_top - y\_bottom \}$$

There isn‘t much to explain, there are just pre-computed values that remain constant over the whole transformation. *a* and *b* are mirror parameters, Δ is distance between mirror axis and projection cylinder (or in different words, projection cylinder radius). *y_bottom* and *y_top* are bottom and top bounds of projection cylinder. All those have to be meassured in texture-space, here is a snippet of c-like pseudo-code on how to calculate them for a given mirror:



H3G mirror configuration

```
float a2 = 789.3274, b2 = 548.144;
// mirror equation ...

float delta = 1000.0, y_bottom = -870.0, y_top = 4630.0;
// delta, bottom and top y; in milimeters

float f_scale = r_outer / 30.0;
// r_outer refers to outer radius in texture space
// (second component of „mirror_area“ vector)

delta *= f_scale, y_bottom *= f_scale, y_top *= f_scale;
// scale delta and bottom and top projection cylinder bounds

f_scale *= f_scale;
// a and b are their respective 2nd powers so we have to scale them
// accordingly with 2nd power of scaling coefficient

a2 *= f_scale, b2 *= f_scale;
// scale a and b

float F = sqrt(a2 + b2);
// calc F (mirror eccentricity, y-coordinate of mirror‘s focus)

mirror_geom = vec4(a2 / b2, 1.0 / (delta * delta),
                   2 * F / delta, -a2 - b2);
mirror_geom2 = vec4(a2, F, b2 / a2, b2 * b2 / a2);
geom_offset = vec2(y_bottom, y_top – y_bottom);
// fill param vectors
```

Pseudo-code, describing coefficient calculation

(constants *a2* and *b2* were taken from mirror surface equation, constants *delta*, *y_bottom* and *y_top* were determined experimentaly, number *30.0* in f_scale calculation is mirror radius in milimeters)

Geometrical transformations are noticeably slower, with source resolution 1440×1080 and destination resolution 1024×768 it takes ~2.6 miliseconds to transform the image on the same GeForce 6600. You can see result below. (the quality is noticeably worse, because previous image was created from HDTV 1440×1080 data and here, source image is 720×576 only)

geometric transformation in action

TrUt reference

TrUt stands for transformation utility, it's a very simple class, designed with regards to easy-to-understand interface. You can see class header below:

```
enum {
    trut_Ok,               // no error
    trut_Error,            // generic error (trut kaput)
    trut_NoPaths,          // no supported paths found (init error)
    trut_NoShaders,        // no shaders found (init error)
    trut_NoRenderBuffers,  // no render buffers supported (init error)
    trut_NoMeomory,        // not enough memory
    trut_GLError,          // OpenGL error
    trut_NoGL13,           // need at lest OpenGL 1.3 (init error)
    trut_NoMultitextureExt,// need multitexture extension (init error)
    trut_NoShaderExt,      // need shaders (init error)
    trut_BadPixelType,     // unsupported ImageStruct pixel type
    trut_ShCompileError    // shader compile error
};

enum {
    tt_Simple, tt_Geometrical, tt_Geom_Polynet // transform type
};

enum {
    ipp_Nothing, ipp_SkinDetect, ipp_CalcGrayscale
};
// “immediate“ post processing – result is stored into alpha channel
// of the unwrapped image (from the left alpha is read from source
// image, alpha is likehood of apropriate pixel color represents skin,
// alpha is intensity, calculated from pixel RGB)

class CTrUt {
public:
    CTrUt();
    ~CTrUt();

    int n_GetLastError();
    int b_Ready();

    const char *p_s_ShaderInfoLog();
    // here you can receive info when trut_ShCompileError

    int Init(int n_transform_type, int n_immediate_postprocessing_type
        int n_source_width, n_source_height,
        int n_transform_width, n_transform_height,
        int (*Load_RenderPaths)(std::vector<TPath*>&) =
        std_Load_RenderPaths,
        int (*load_sh_function)(std::vector<CShaderInfo*>&,
        const char*) = std_Load_Shaders);
    // init transformation, when succesful, b_Ready() returns true

    int n_Transform_Width(); // size of transformed image (result)
    int n_Transform_Height();// (useful when downloading images)
```

```
int Upload_ImageStruct(ImageStruct *p_image);
// p_image should contain image of size n_source_width
// per n_source_height pixels otherwise OpenGL errors may
// occur (depends on graphics card, opengl driver, etc)

int Set_SkindetectParams(float f_r_avg, float f_g_avg,
    float f_k11 = 0.62545f, float f_k21 = 0.00357f,
    float f_k12 = 0.00357f, float f_k22 = 0.23543f);
// set skin-detection params (works if post-processing
// is set to ipp_SkinDetect only)

int Set_SimpleParams(float f_inner_r, float f_outer_r,
    float f_center_x, float f_center_y,
    float f_angular_offset, float f_image_ratio);
// call for simple transform only, otherwise returns false

int Set_GeomParams(float f_inner_r, float f_outer_r,
    float f_center_x, float f_center_y,
    float f_angular_offset, float f_image_ratio, float f_a2_mm,
    float f_b2_mm, float f_mirror_radius_mm, float f_delta_mm,
    float f_bottom_y_mm, float f_top_y_mm);
// call for geom or geom_polynet transform only, otherwise
// returns false, parameters followed by "_mm" may be
// passed in milimeters

int Transform(int n_quad_tesselation, int b_upside_down = false);
// begin transforming image, process is CPU-independent, after
// the call, you can do some usefull stuff and let graphics
// card work. function switches drawing context to offscreen buffer
// n_quad_tesselation should be 1 for simple and geometrical
// transform. however for Geom_Polynet, value > 100 is reccomended

int Download_ImageStruct(ImageStruct *p_image,
    float f_angle, int n_y);
// download image from current bound framebuffer. p_image must be
// already allocated, f_angle is angle in radians (fmod-ed
// to interval 0 .. 2Pi), specifying angle where mid of the image
// rectangle should lie, n_y is vertical position (of top border)
// in raster
// when angle is near 0 or 2Pi so image rectangle crosses left
// or right border of unfolded image, it is wrapped over (similar
// to GL_REPEAT texture clamp mode), when image rectangle crosses
// bottom or tom border of unfolded image, it's part, lying outside
// isn't modified
// note first call of this function forces OpenGL to finish
// transformetion so if it‘s not ready, CPU have to wait here

int ReleaseFramebuffer();
// if you want to draw to your opengl window, it's necessary to
// un-bind transformation's off-screen framebuffer. however, it's
// not necessary to do this between consequent calls of Transform()
// so if you're not going to use open-gl between transformations,
// you don't have to call this

unsigned int n_TransformedTexture_OpenGL_Id();
// return OpenGL handle of texture, containing transformed image
// can be freely used for displaying results on screen, but
// must not be deleted
```

```
int ShutDown();
// free all open-gl resources, b_Ready() returns false

static int std_Load_RenderPaths(std::vector<TPath*> &r_path_list);
static int std_Load_Shaders(std::vector<CShaderInfo*>
    &r_shader_list, const char *p_s_path_name);
// here you can get standard shaders (you can also dig them out
// this way and rewrite them to satisfy your needs)

int GL_Init(HWND h_wnd, int n_width, int n_height, int n_bpp = 32,
    int n_depth_bpp = 0, int n_stencil_bpp = 0,
    int b_fullscreen = false);
int GL_SetContext();
int b_GL_Running();
int GL_PageFlip();
int GL_ShutDown();
// new functions for work with Open-GL

int GL_Wait_UntilTransformed();
// wait until transformation is done

void GL_Line(float x1, float y1, float x2, float y2,
    unsigned char r, unsigned char g, unsigned char b);
// draw 2D line; can control thickness using glLineWidth(int)

void GL_TexturedRect(float x1, float y1, float s1, float t1,
                     float x2, float y2, float s2, float t2,
                     unsigned int n_texture_id);
// draw 2D textured rectangle (x,y are screen coords,
// s,t are texture coords)
// there's currently an issue with BGR, more in changes section
// at the end of this document

int b_AsyncSupported();
// return true if asynchronous transfers are supported
// (call after init, never before!)

int b_QueryStateSupported();
// return true if transfer / transformation completeness query
// is supported (call after init!)

int Upload_ImageStruct_Async(ImageStruct *p_image);
int Download_ImageStruct_Async(ImageStruct *p_image,
    float f_angle, int n_y);
// !! effective image contents are from previous call of
// Download_ImageStruct_Async() !! (ie. image is delayed by 1 freme)
// should work asynchronously (when supported; otherwise returns
// false), parameters are the same as for their's blocking versions

int b_UploadFinished();
int b_TransformFinished();
int b_DownloadFinished();
// completeness query functions. in case completeness query is
// not supported, return always true -> use rather synchronous
// when called before call to Upload_ImageStruct_Async(),
// Download_ImageStruct_Async() or Transform() respectively (which
// is asynchronous by its nature), return value is true, but open-gl
// error GL_INVALID_OPERATION is generated
```

```
    void WaitUntil_UploadFinished();
    void WaitUntil_TransformFinished();
    void WaitUntil_DownloadFinished();
    // you can also decide to wait until the task is finished.
    // in case state queries aren't supported, returns after ALL OpenGL
    // commands in queue finished
    // (contains open-gl call, not "while(!b_*Finished());" - like loop)
};
```

The intended use of this class would look like this:

```
CTrUt *p_tr_ut = new CTrUt;

if(!p_tr_ut->Init(tt_Simple, ipp_Nothing, 1440, 1080, 1024, 1024 / Pi)||
   !p_tr_ut->Set_SimpleParams(.01f, .4f, .5f, .5f, .0f, 1.333f)) {
    switch(p_tr_ut->n_GetLastError()) {
        ...
    }
    delete(p_tr_ut);
    exit(-1);
}
// init simple transformation, sourcing some HDTV video, outputing
// 1024 x 1024 / Pi ~ 326 pixels and set transform parameters

ImageStruct *p_people[4];

CreateFourImageStructs(&p_people, 1024 / Pi, 1024 / Pi);
// alloc four imagestructs to store images of people,
// sitting arround table (squares of size 1024 / Pi pixels)

while(!kbhit()) {
    ImageStruct *p_image = GrabFrame();
    // get an image to process

    if(!p_tr_ut->Upload_ImageStruct(p_image) ||
       !p_tr_ut->Transform()) {
        switch(p_tr_ut->n_GetLastError()) {
            ...
        }
        delete(p_tr_ut);
        exit(-1);
    }
    // upload image to graphics card (theoretical transfer rates ~4GB/s,
    // meassured ~800Mb/s (AGP, PCI Express should be much faster), that
    // is 5 miliseconds for our 4 MB HDTV frame)
    // you can adjust parameters here by calling Set_SimpleParams()
    // transform an image (binds offscreen rendering buffer!), next 0.64
    // miliseconds (or 1.10 for geometric transformation)

    HandleEvents(); // can do something useful at here

    for(int i = 0; i < 4; i ++) {
        if(!p_tr_ut->Download_ImageStruct(p_people[i], Pi / 2 * i, 0)) {
            switch(p_tr_ut->n_GetLastError()) {
                ...
            }
```

```
            delete(p_tr_ut);
            exit(-1);
        }
    }
    // download four images with four people, sitting arround the table
    // (next 5 miliseconds, assuming we're downloading the whole
    // transformed image, that is 10.64 - 11.10 miliseconds per frame
    // which corresponds to 93.9 – 90.1 frames per second)

    DoSomethingWithFourImages(&p_people); // use results

    if(want_draw_something_to_window_with_opengl) {
        if(!p_tr_ut->ReleaseFramebuffer()) {
            switch(p_tr_ut->n_GetLastError()) {
                ...
            }
            delete(p_tr_ut);
            exit(-1);
        }
    }
    // if you need to draw something by opengl, release offscreen
    // rendering buffer, bound by CTrUt::Transform
}

p_tr_ut->Shutdown();
delete p_tr_ut;
// cleanup
```

I hope this was self explanatory enough, but just for the sake of clarity: as first step after creation of CTrUt object, you should call Init() and say what kind of transformation and over which resolutions you want, you can also set transformation parameters here.

Then you upload an image you want to transform (*Upload_ImageStruct*()) and call *Transform*(). You may check for errors here (at this point OpenGL errors can occur only).

Now, when the transform is ready, you can call *Download_ImageStruct*() to obtain your images. If you want to do something in your OpenGL window, you have to *ReleaseFramebuffer*(), otherwise repeat the whole process (*from Upload_ImageStruct*())

OpenGL must be running when using CTrUt, you can either use GLUT, or your own OpenGL implementation, or built-in functions. Those are:

- int *GL_Init*(HWND h_wnd, int n_width, int n_height, int n_bpp, int n_depth_bpp = 0, int n_stencil_bpp = 0, int b_fullscreen = false)
- int *GL_SetContext*()
- int *b_GL_Running*()
- int *GL_PageFlip*()
- int *GL_ShutDown*()

You can use *GL_Init*() and *GL_ShutDown*() to init / shutdown OpenGL context, *b_GL_Running*() tells you if you already did initialize OpenGL (via those functions only, not ANY OpenGL initialization), *GL_SetContext*() is called to set OpenGL context, created in CTrUt as active. You should use this function only when multiple OpenGL contexts are running, otherwise you‘re just wasting time. *GL_PageFlip*() displays results.

Both *Upload_ImageStruct*() and *Download_ImageStruct*() stalls the CPU until transfer is complete, which is the reason for asynchronous functions were requested. *Upload_ImageStruct_Async*() and *Download_ImageStruct_Async*() should return (almost) immediately, but they requires GL_NV_fence extension (currently supported by both NV and ATI) and GL_ARB_pixel_buffer_object (ARB approved as 222th extension, supported by NV, hope ATI is going to support it as well) I‘ve done some meassurements which can be seen in the following table:

		PCI-Express	**AGP 8×**
synchronous transfer	upload	576.27	92.65
	download	753.42	170.78
asynchronous transfer	upload	1026.34	103.44
	download	431.43	121.43
	best upload	1816.54	657.49
	best download	898.82	273.39

all readings are in MB per second

Note „best“ upload and download are values, calcualted as *sizeof(transferred data) / (overall time – time spent waiting for transfer end)*, in other words it‘s transfer rate that can be achieved when fully exploiting time we wait for operation to finish (which is possible, but not perfectly).

There‘s a very simple example for use of CTrUt with GLUT, i hope it‘s going to be enough for everyone to understand it. If you run into any trouble, you can contact me at:

Lukas Polok
student of Faculty of Information Technology
xpolok00@stud.fit.vutbr.cz
ICQ: 200339454

To-Do & notes

Library counts on GL_ARB_texture_non_power_of_two, i‘m going to add support for GL_ARB_texture_rectangle (supported on older cards as well, but handling texture coordinates in a bit different way)

It‘s not certified to be compatible (read it‘s not compatible at all, it almost definitely will NOT run on other card than GeForce 6600) - to be fixed.

Current shader implementation is only using GL-SLang language, i‘m going to rewrite final versions of shaders to assmebly code (*GL_ARB_fragment_program*) as well.

Need to solve drawing trough OpenGL (by now everything (concerning mirror images) drawn trough open-gl has swapped red and blue channel)

Re-work this documentation. It‘s a real mess now ... add contents, skincolor, gray equations, more precise function reference.

Bug fixes / modifications

02-02-2006 - Fixed some issues with setting last error value (some functions could fail, but didn‘t set last error so we couldn‘t see what and why happened)

02-02-2006 - Unfolding direction is actually clockwise, NOT counter-clockwise as mentioned in this document. Fixed to be clockwise in both library and document.

02-07-2006 - Added functionality for initializing OpenGL, requested by mr. Sumec. It is exhibited trough class *COpenGL_Driver* in *OpenGLDrv.h* file or trough *CTrUt* itself via calls to *Init_OpenGL*(), *PageFlip*() and *ShutDown_OpenGL*().

02-07-2006 - Removed calls to *fmodf*() and *sqrtf*() functions. Those are fast *float* functions, taking advantage from new instruction sets, available in MSVC only. However, the main application is built in BCB which we found unable to use MSVC generated dll so we have to be able to recompile it under BCB as well.

02-07-2006 - Fixed bug arround DigiLib ~ RGB / OpenGL ~ RGBA mismatch, as well as upside-down image transfers, arround with removing ±1 bug, causing black strip originating in downloaded image. Added upside-down transformation option. Lots of code added into glut example (mainly debug purposes)

02-07-2006 - Added some OpenGL wrapper functions for waiting until transformed image is ready (called internaly in *ReleaseFramebuffer*() and *Download_ImageStruct*()) and for drawing coloured lines and textured rectangles. (OpenGL has no native support for ellipses, etc ... have to code it yourself and use lines)

02-16-2006 - Added support for asynchronous data transfers. This involved changing data storrage to BGRA (instead of RGBA) to avoid data swizzling during the transfers (because the graphics card is storing data as BGRA). The only drawback is you get strange coloured image when drawing trough open-gl. This can be fixed by a very simple shader.

02-16-2006 - Added immediate post processing operations and geometric quadrliliteral-mesh transformation