Courses 3DCx, C2: Texture, Code Comments

In `MainFrm.cpp` inside the function `PreCreateWindow(CREATESTRUCT& cs)`

`cs.cx = 700; cs.cy = 700;`//These two variables force a nearly quadratic window that suits well the default quadratic coordiante system of OpenGL (-1 to +1 in both directions).

In `texture1View.h` in `class CTexture1View : public CView`

`private: HGLRC hglrc;`//handle for an OpenGL-Rendering-Context (see FAQs of project 4 = C4_OpenGL_FAQ.htm)

`std::vector< BYTE > ScaledImage;`//Dynamic array for a scaled texture image. OpenGL textures must have x/y-dimensions as powers of 2: 16, 32, 64, 128, 256, 512, 1024. They need not to be quadratic (f.i. 64*1024 is possible.). Any image can directly serve as an OpenGL texture image, when it fulfills this restriction but we will always transform the original image into a quadratic scaled texture image (default 128*128) carrying the name `ScaledImage`.

In the head of `texture1View.cpp` in front of `class CTexture1View : public CView`

`#include < gl/gl.h >` //This line is useless if `gl/gl.h` is already included (usually in `texture1Doc.h`) otherwise we need it here. `gl.h` contains all function declarations of OpenGL. When your compiler does not contain this file (usually in the path `C:\Programs\Microsoft Visual Studio\VC98\Include\GL`), you have no chance to compile this project.

`float f[6][3]={{0.0f,0.0f,1.0f},{0.7f,0.7f,0.0f},{-0.7f,0.7f,0.0f},{-0.7f,-0.7f,0.0f},{0.7f,-0.7f,0.0f},{0.0f,0.0f,-1.0f}};` //The octahedron has 6 vertices and each vertex has three coordiantes: x, y and z . The first and the last vertex are the tips of the octahedron on the Z-axis and the 4 remaining vertices form the quadratic "equator" whose sides are parallel to the X- and Y-axes.

`int face[8][3]={{0,4,1},{0,1,2},{0,2,3},{0,3,4},{5,1,4,},{5,2,1},{5,3,2},{5,4,3}};` //The octahedron has 8 triangular faces. Each face is defined by 3 vertices. The vertices carry numbers corresponding to their order in f[6][3]. Any 3 vertices are aligned counterclockwise.

`#define NX 0.8192f #define NY 0.8192f #define NZ 0.5735f` //All 8 normal vectors perpendicular to the 8 triangles are symmetrically composed of these 3 values . Explanation: Let us consider the first triangle `{0,4,1}` and let us call its vertices
`{P0, P1, P2}` = `{{0.0f,0.0f,1.0f},{0.7f,-0.7f,0.0f},{0.7f,0.7f,0.0f}}`.

1. Take two arbitrary legs from the triangle and express them as vectors.
   Let us choose leg1 from `P0` to `P1`: `V1=P1-P0= { 0.7-0.0,-0.7-0.0, 0.0-1.0 } = { 0.7,-0.7,-1.0 }`.
   Let us choose leg2 from `P1` to `P2`: `V2=P2-P1= { 0.7-0.7, 0.7+0.7, 0.0-0.0 } = { 0.0, 1.4, 0.0 }`.
2. Compute the vector product `VP = V1 x V2` defined as:
   `VPx=V1y*V2z - V2y*V1z = (-0.7)*(+0.0) - (+1.4)*(-1.0) = +1.4`
   `VPy=V1z*V2x - V2z*V1x = (+0.0)*(+0.7) - (+0.0)*(+0.7) = +0.0`
   `VPz=V1x*V2y - V2x*V1y = (+0.7)*(+1.4) - (+0.0)*(-0.7) = +0.98`
3. Correct the orientation of VP if necessary: It must point toward the outside = outer surface of the octahedron. If VP points to the wrong (opposite) direction, OpenGL will probably show us an empty triangular hole instead of a triangular surface covered by a texture.
   The easiest test is the scalar product `SPtest` of `VP` with an arbitrary vector `Vtest` pointing into the center of the octahedron.
   Let us choose the vector from `P0` to `{ 0,0,0 }` as `Vtest = { 0,0,0 } - P0 = { 0.0, 0.0, -1.0 }`.
   `SPtest = VP * Vtest = 1.4*0.0 + 0.0*0.0 +0.98*(-1.0) = -0.98.`
   Everything is OK if `SPtest` is negative, otherwise `VPx`, `VPy` and `VPz` must be multplied by `-1.0`.
4. Normalize `VP` to the normal length of `1.0`.
   The length of `VP` is `sqrt(1.4*1.4 + 0.0*0.0 + 0.98*0.98) = 1.7089`.
   We have to divide `VPx`, `VPy`, `VPz` by this length in order to obtain the final normal vector `N = { 1.4/1.7089, 0.0, 0.98/1.7089 } = {0.8192, 0.0, 0.5735 }`

Because of the symmetry of the octahedron, all other 7 normals are composed of the same 3 values `0.8192` and `0.0` and `0.5735` (but at different positions and with changing signs).

`float n[8][3]={{NX,0.f,NZ},{0.f,NY,NZ},{-NX,0.f,NZ},{0.f,-NY,NZ},{NX,0.f,-NZ},{0.f,NY,-NZ},{-NX,0.f,-NZ},{0.f,-NY,-NZ} };` //The 8 normal vectors perpendicular to the 8 triangles are symmetrically composed of 3 (positive and negative) constants and zero. There is exactly one vector in any quadrant of the XZ plane and exactly one in any quadrant of the YZ plane.

`float diffuseLight[4]={1.5f,1.5f,1.5f,1.0f};` //A light source dissipating light into all directions like a candle does. The first 3 parameters indicate the intensity of red, green and blue (RGB components). The last parameter has no effect at all. You can change the RGB values at run time in order to enforce (`>1.0`) or to dim (`<1.0`) the light.

`float materialDiffuse[4]={1.0f,1.0f,1.0f,0.0f};` //Reflective properties of the material of the octahedron. The first 3 parameters indicate the reflectivity for red, green and blue (RGB components). The last parameter has no effect. Thus the octahedron reflects 100% of any diffuse red, green and blue light. At run time OpenGL will multiply these RGB values with the correponding RGB-values of the `diffuseLight`-vector in order to compute the local brightness.

`float LightPosition[4]={1.0f,0.0f,1.0f,1.0f};` //The first 3 parameters indicate the position of the light source at the right side of the window in front of the screen. The last parameter has no effect. At run time OpenGL will calculate for each point P in each triangle the cosine of the angle between the beam from `LightPosition` to P and the normal of the triangle. Then OpenGL will multiply the cosine with the intensity specified by `diffuseLight` and with the RGB-properties specified by `materialDiffuse` in order to get the brightness of the point P. Normally the `LightPosition` rotates together with the octahedron. It will require some special effort to keep the `LightPosition` constant.

In `texture1View.cpp` inside `CTexture1View::OnCreate(LPCREATESTRUCT lpCreateStruct)`

`if (CView::OnCreate(lpCreateStruct) == -1) return -1;` //This line of code is from Microsoft. Leave it here, it produces an error message if something goes wrong during the first construction of our client area.

`CClientDC dc( this );` //Device context of the current client area. Such a device context normally offers more than one single pixel format (see below).

`PIXELFORMATDESCRIPTOR pfd = { sizeof( PIXELFORMATDESCRIPTOR ), 1, PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER, PFD_TYPE_RGBA, 24,0,0,0,0,0,0,0,0,0,0,0,32,0,0,PFD_MAIN_PLANE,0,0,0 };` //pfd specifies what we expect from our device context (see FAQs). The most important parameter is the bit-combination of `PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER.` When we do not set these bits, we loose the opportunity for using and swapping the double output buffer which results in severe flicker effects.

`int i = ChoosePixelFormat( dc.m_hDC, &pfd );` //Checks all possible pixel formats that our device context offers. Returns the no. of the one that matches best to `pfd`.

`SetPixelFormat( dc.m_hDC, i, &pfd );` //Realizes the best matching pixel format inside our device context.

`hglrc = wglCreateContext( dc.m_hDC );` //Makes a hardware-independent OpenGL rendering context that matches as good as possible to the current hardware- and Windows-dependent device context and returns a handle to it (or NULL when it is impossible to detect one). The rendering context frees the OpenGL code from any dependency from the hardware and from the operating system. Any code lines that will use this `hglrc` are truely platform-independent.

`SetTimer(1, 1, NULL);` //Starts the timer no. 1 with the highest possible velocity. The first parameter assigns a user no to this timer (64 parallel timers are possible). Parameter 2 defines the desired time interval in milliseconds from one timer event (timer message) to the next one. The operating system cannot react with 1000 events in a second but it will now produce such events as often as possible. Parameter 3 is not used.

`return 0;` //Exit with the normal return value of 0.

In `texture1View.cpp` inside `CTexture1View::OnTimer(UINT nIDEvent)`

`CClientDC dc( this );` //Device context of the current client area

`wglMakeCurrent( dc.m_hDC, hglrc );` //Windows is informed, that it has to map in the future the `hglrc` rendering context onto its current `dc.m_hDC` device context. All graphics drawn by OpenGl into its rendering context are accepted from now on by Windows to be automatically drawn into the device context of the currently active window (more exactly: into the device context of the client area of the currently active window, which is ours in this moment).

`glDrawBuffer( GL_BACK );` //In order to prevent the graphics from flickering draw the image into the invisible back buffer rather than into the visible one .

`glClearColor( 0.1f, 0.1f, 0.1f, 0.f);` //The color to erase the canvas is dark gray without any transparency.

`glClear( GL_COLOR_BUFFER_BIT );` //Erase everything.

`glMatrixMode( GL_MODELVIEW );` //The following statement `glRotatef` shall rotate the polygon model (not the observation point nor the texture).

`glRotatef( 1.f, 1.f, 1.f, 1.f );` //Rotate the octahedron 1 degree around the X-, the Y- and the Z-axis.

`glPushMatrix();` //Make a copy of the current matrix in the stack because we will need later the current content of the matrix. Theory: The light position is stored in eye coordinates which automatically are transformed by the `ModelView` matrix. This and the following three statements isolate the light position from being transformed by the `ModelView` matrix.

`glLoadIdentity();` //Reset the current matrix to the unit matrix , otherwise it will rotate the light source togeher with the octahedron. We want the diffuse ligth source at a constant position.

`glLightfv (GL_LIGHT0, GL_POSITION, LightPosition );` //We set now the light source at the position as specified by the global array `float LightPosition[4]={1.0f,0.0f,1.0f,1.0f};` in the head of `texture1View.cpp`.

`glPopMatrix();` //Reload its former content from the stack into the current matrix.

`glLightfv (GL_LIGHT0, GL_DIFFUSE, diffuseLight );` //Applies the `diffuseLight` parameters for the lighting model as specified by the global array `float diffuseLight[4]={1.5f,1.5f,1.5f,1.0f};`. Both `light` statements are not obligatory. When you delete them, OpenGL applies by default the `diffuseLight` properties: `{1.f,1.f,1.f,1.f}`.

`glMaterialfv( GL_FRONT, GL_DIFFUSE , materialDiffuse );` //Applies the `materialDiffuse` parameters for the lighting model as specified by the global array `float materialDiffuse[4]={1.0f,1.0f,1.0f,0.0f};`. Both `material` statements are not obligatory. When you delete them, OpenGL applies by default the `material` properties: `{0.8f,0.8f,0.8f,0.0f}`.

`glEnable (GL_LIGHTING);` //This flag prepares OpenGL to perform lighting calculations. (If disabled there will be no calculations concerning normals, light sources and materials.).

`glEnable (GL_LIGHT0);` //This flag switches on the capability of OpenGL to evaluate the lighting equation which includes the properties of the material, the orientation of the normals and the properties of the light source(s).

`glEnable (GL_CULL_FACE);` //This flag switches on the capability of OpenGL to discard back-faced (i.e. invisible) triangles. If it is switched on, OpenGL discards by default the back-faced facets. (to cull heisst in Deutsch: als minderwertig aussortieren)

`CTexture1Doc* pDoc = GetDocument();` //Get a pointer to the class `CTexture1Doc` which contains the `OriginalImage` and the flag `APictureHasBeenLoaded`.

`int cx, cy, format, i, j;` //texture size and an OpenGL flag

`if ( pDoc->APictureHasBeenLoaded )` //When a new bitmap has been opened by the user, we must scale it and make an OpenGL texture object out of it. When nothing has currently been opened, we skip this if-clause and use no texture (at the first run) or the previously loaded texture object.

`cx = cy = 128;` //No. of columns and rows of the texture. Possible alternatives: `cx = cy = 256; cx = 64; cy = 512; etc.` The values must be 2 power n.

`if ( pDoc->pIH->biBitCount == 24 )` //Is this a 24-bit bitmap ? `pDoc` points to `CTexture1Doc::pIH` and `pIH` points to the BitmapInfoHeader.

`ScaledImage.resize( cx*cy*3 );` //The dynamic array has now space for cx*cy pixels with 3 bytes each.

`format = GL_BGR_EXT;` //This flag tells OpenGL that Microsoft has made a bad decision in the early days of Windows: Windows Device Independent Bitmaps (DIBs) store RGB-colors in reversed order: at first the blue, then the green and finally the red value. If you do not know that and keep the flag to `format = GL_RGB`, red Microsoft pixels will appear blue in OpenGL and vice versa.

`else if ( pDoc->pIH->biBitCount == 8 )` //Is this a 8-bit bitmap ? `ScaledImage.resize( cx*cy );` //The dynamic array has now space for cx*cy pixels with 1 byte each.

`format = GL_LUMINANCE;` //This flag tells OpenGL that this is a gray value texture. If there was a palette it is ignored, no colors will be displayed and the texture will probably look strange.

`gluScaleImage(format,pDoc->pIH->biWidth,pDoc->pIH->biHeight,GL_UNSIGNED_BYTE,&(pDoc->OriginalImage.front()),cx,cy,GL_UNSIGNED_BYTE,&ScaledImage.front() );` //This is an intelligent subroutine from the OpenGL utility library `glu32.lib` declared in `gl/glu.h`. It redistributes the columns and rows of a source raster image by repeating (in case of zoom > 1.0) or deleting them (in case of zoom < 1.0). It accepts many pixel formats, it minimizes the inevitable geometric distortions and works pretty fast. See the help texts of Visual Studio.

`glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );` //In case the texture must be zoomed up: Create the new pixels by repeating the nearest old pixel .

`glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );` //In case the texture must be zoomed down: Create the new pixels by preserving only the nearest old pixel.

`glTexImage2D ( GL_TEXTURE_2D, 0, GL_RGB, cx, cy, 0, format, GL_UNSIGNED_BYTE, &ScaledImage.front());` //Transforms an image into a texture object with specified border, size and pixel properties.

`glBindTexture ( GL_TEXTURE_2D, 1 );` //Stores the texture object for future use and give it the retrieval no. "1".

`pDoc->OriginalImage.clear();` //Kill the original image which is not needed anymore.

`pDoc->APictureHasBeenLoaded = FALSE;` //This flag prevents from scaling the `OriginalImage` and from creating any texture object at the next timer event.

`glEnable(GL_TEXTURE_2D);` //This flag switches on the capability of OpenGL to perform texturing. If no texture exists it has no effect.

`glBindTexture( GL_TEXTURE_2D, pDoc->texName );` //Tells OpenGL to use the existing texture object named no. 1. If no such texture object exists it has no effect.

`glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE );` //This function sets the texture environment parameter GL_MODULATE, telling OpenGL to multiply for each pixel the RGB-values of the texture with the corresponding RGB-values resulting from the lighting equation. If no texture exists it has no effect.

`#define VERTEX glVertex3f( f[j][0], f[j][1], f[j][2] );` //VERTEX is an abbreviation for a statement that we will need below three times .

`for ( i = 0; i < 8; i++ )` //Take all eight triangles

`glBegin( GL_POLYGON );` //Start do draw a triangle. If there is no texture yet, the three `glTexCoord2f`-statements make no harm and have no effect.

`glNormal3f( n[i][0], n[i][1], n[i][2] );` //Gives to OpenGL the current normal vector of the current triangle.

`glTexCoord2f(0.5f, 1.0f); j=face[i][0]; VERTEX` //Tells OpenGL the top mid point of the texture and the upper point of the current triangle.

`glTexCoord2f(0.0f, 0.0f); j=face[i][1]; VERTEX` //Tells OpenGL the lower left point of the texture and the lower left point of the current triangle.

`glTexCoord2f(1.0f, 0.0f); j=face[i][2]; VERTEX` //Tells OpenGL the lower right point of the texture and the lower right point of the current triangle.

`glEnd();` //Stop drawing.

`glFlush();` //Wait until everything is transferred from main memory into the graphics board. You can delete this statement if your bus and your graphics board are fast enough.

`SwapBuffers( dc.m_hDC );` //Show the background buffer on the screen, hide the foreground buffer and exchange their names.

`dc.TextOut( 0, 0, "Please open a 8/24-Bit *.BMP file!");` //Urges the user to load a bitmap image.

In the head of `texture1Doc.h` in front of `class CTexture1Doc : public CDocument`

`#include < vector >` //Declares the dynamic arrays of the Standard Template Library STL. The blanks inside the < > clauses may be suppressed. They are just necessary in a HTML-Document otherwise HTML treats the words `vector`, `gl/gl.h` and `gl/glu.h` as HTML-Tags.

`#include < gl/gl.h >` //`gl.h` contains all function declarations of OpenGL. When your compiler does not contain this file (usually in the path `C:\Programs\Microsoft Visual Studio\VC98\Include\GL`), you have no chance to compile this project.

`#include < gl/glu.h >` //`glu.h` contains a lot of utility function declarations related to OpenGL. `gluScaleImage(...)` is such a utility function and we use it later to scale any BMP to the fixed format of 128*128. When your compiler does not contain `glu.h` (usually in the path `C:\Programs\Microsoft Visual Studio\VC98\Include\GL`), you cannot use `gluScaleImage(...)` in the code below.

In `texture1Doc.cpp` inside the constructor function `CTexture1Doc::CTexture1Doc()`

`pIH = (BITMAPINFOHEADER*) IBytes;` //The typed pointer `pIH` points now to the the head of `IBytes[1200]`.

`pI = (BITMAPINFO*) IBytes;` //The typed pointer `pI` points now to the the head of `IBytes[1200]` as `pIH` does. We do not use `pI` below and therefore nothing goes wrong when you delete this statement.

`APictureHasBeenLoaded = FALSE;` //This flag will be used inside the `OnTimer` function of `CTexture1View`. It switches the construction of a texture object on/off.

In `filter1Doc.cpp` inside `Serialize(CArchive& ar)` inside the `else` clause: Code for reading an image and for computing the noisy and the filtered images

`ar.Read( & FH, sizeof(BITMAPFILEHEADER) );` //Read 14 bytes from the hard disk.

`if ( FH.bfType != 'MB') { forget_it(); return;` //The first 2 bytes form the reversed string of BM.

`if ( FH.bfSize <= 54 ) { forget_it(); return;` //A bitmap file contains at least 55 bytes.

`if ( FH.bfOffBits < 54 ) { forget_it(); return;` //The shortest possible header needs 54 bytes.

`int nBytesInfo = FH.bfOffBits - sizeof(BITMAPFILEHEADER);` //That is the space left for the BitmapInfoHeader and palette.

`int nBytesPixel = FH.bfSize - FH.bfOffBits;` //That is the space for all pixels.

`ar.Read( IBytes, nBytesInfo );` //Read BitmapInfoHeader+palette from the hard disk

`OriginalImage .resize( nBytesPixel );` //Reserve memory space for the original image.

`ar.Read( &OriginalImage.front(), nBytesPixel );` //Read all pixels from the hard disk into the main memory starting at the first adress of the dynamic byte array named `OriginalImage`.

`APictureHasBeenLoaded = TRUE;` //This flag will be used inside the `OnTimer` function of `CTexture1View`. It allows the construction of a new texture object during the next call of the `OnTimer` function of `CTexture1View`.

`if ( pIH->biBitCount == 24 ) return;` //Everything is ok.

`if ( pIH->biBitCount == 8 ) return;` //Everything is ok.

`forget_it();` //This bitmap has neither 24 nor 8 bit. Throw it away.

In `filter1Doc.cpp` function `void CTexture1Doc::forget_it()`

`OriginalImage.clear();` //If space has been occupied, free it and give it back.

`APictureHasBeenLoaded = FALSE;` //This flag will be used inside the `OnTimer` function of `CTexture1View`. It switches off the construction of a texture object .

`for ( int i=0; i < 10; i++ ) MessageBeep(-1);` //Beep 10 times to inform the user that this bitmap loading has gone wrong.