CHECKPOINT 1

? *Review your knowledge on C++, OpenGL and Windows. Write a small program to demonstrate basic features of OpenGL – be inspired by the tutorials found on http://nehe.gamedev.net or the sample program shown in the lecture.*

There are many resources on the web explaining how to write a Win32 based OpenGL program. Most of them use free libraries such as GLUT or SDL. They provide all you – but they also limit your freedom.

I wanted to write a program that is as small as possible and still written in proper C++. Moreover, it should be fast (as fast as the hardware can be !) and offer some limited interactions. Here is a screenshot taken on a Centrino 1.4GHz / GeForce4MX 440 computer:

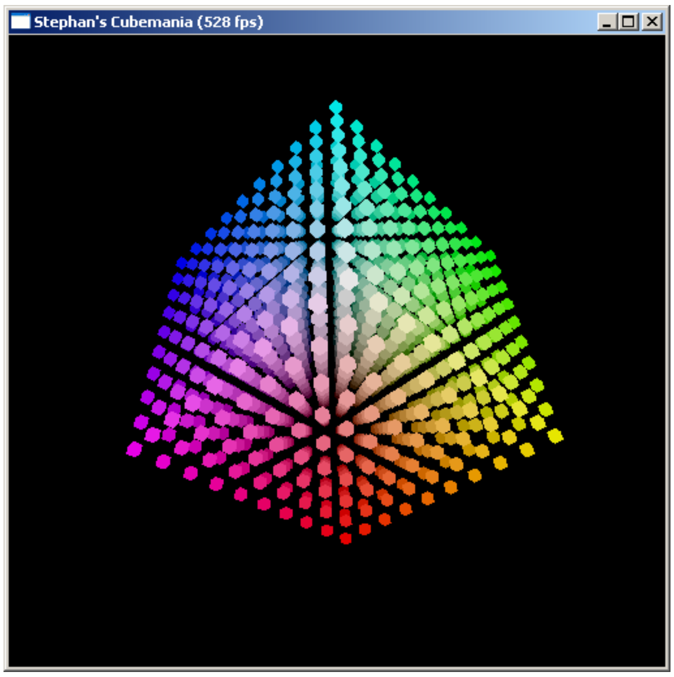

Figure 1: Cubemania in action

As you can see, my program runs at a speed of more than 500 frames per second (fps) with a window size of 512x512 pixels. More recent hardware achieves frame rates beyond 1000 fps. The program is usually limited by the pixel fillrate because in fullscreen mode (1024x768) the frame rate drops to 250 fps on my computer.

These keys are supported:

Key	Action
ESC	Quit
Space, Pause	Stop rotation
F5	Fullscreen

The most challenging task was to create a good but small Win32 framework. Using object-oriented concepts, all Win32 functionality is encapsulated in the `GLWindow` class. An OpenGL program inherits from GLWindow and has to implement the methods `init()` and `redraw()`. The former can be used to initialize some OpenGL settings like enabling the depth buffer. `redraw` is called every time the application has to repaint the window contents.

An UML like diagram shows the relationships:

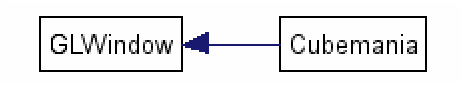

Figure 2: Class diagram

The minimal set of #includes:

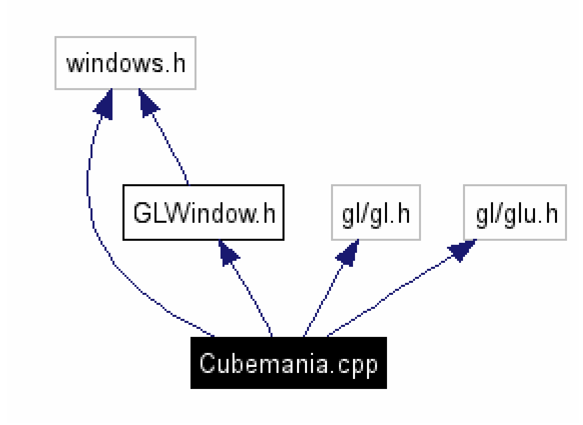

Figure 3: Required header files

I will not talk about further Win32 implementation details since I describe them in depth in the source code.

Essentially for creating an OpenGL window is to cope with so-called *pixel format descriptors*. They define the size and format of the desired window. The size is obviously characterized only by the width and the height while the format can be pretty complex. Most important is the colour buffer depth. Today's PCs have almost exclusively 32 bit per pixel which are split in 8 bits for the red, green and blue channel. 8 bits are reserved (for the alpha channel useful for transparency). A widely supported bit order is RGBA; there are only a few reasons to go for a different bit order. Common values for the depth buffer are 16 or 32 bits per pixel. The latter clearly offers higher precision but needs more memory. A very nice feature is double buffering: OpenGL renders the new frame in an invisible colour buffer and displays it when all drawing operations are finished. This techniques effectively prevents flickering:

```
    // set pixel format
    PIXELFORMATDESCRIPTOR pfd;
    ZeroMemory(&pfd, sizeof(pfd));
    pfd.nSize      = sizeof(pfd);
    pfd.nVersion   = 1;
    pfd.dwFlags    = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | (double-
buffer ? PFD_DOUBLEBUFFER : 0);
    pfd.iPixelType = PFD_TYPE_RGBA;
    pfd.cColorBits = colorbuffer;
    pfd.cDepthBits = depthbuffer;
    pfd.cAccumBits = accumbuffer;
    pfd.iLayerType = PFD_MAIN_PLANE;

    // set buffers' depths
    int format = ChoosePixelFormat(hDC, &pfd);
    assert(format > 0);
    bool setPixelFormat = SetPixelFormat(hDC, format, &pfd) != 0;
    assert(setPixelFormat);
```

While writing and testing my frame rate counter I noticed that the frame rate was never higher than 60 fps. Incidentally, that's exactly the refresh rate of TFT displays. So I searched on the internet and found that the graphic card settings usually limits the frame rate to the monitor's refresh rate. Recent NVidia and ATI drivers allow to change that behaviour (know as VSync) but the default setting is still that OpenGL applications have to wait for the VSync. Further investigations on the topic revealed that an OpenGL extension allows to temporarily override these settings. It may not be available on all PCs because it is an extension and does not belong to the compulsory core functions of OpenGL.

```
// switch off vsync if possible
typedef GLboolean (__stdcall * SwapProc) (GLint interval);
SwapProc wglSwapIntervalEXT =
    (SwapProc) wglGetProcAddress("wglSwapIntervalEXT");
if (wglSwapIntervalEXT)
    wglSwapIntervalEXT(0);
```

To fully exploit the power of hardware graphics accelerators, I decided to take advantage of vertex arrays introduced in OpenGL 1.1. The idea is to define an array of vertices and tell OpenGL the memory address. Then, only one call is needed to submit all vertices to OpenGL. For example: a cube consists of 6 sides each determined by 4 vertices. A simple `glVertex` based approach requires 24 time-expensive OpenGL calls. A vertex array requires just four: two to enable/disable vertex arrays, one to tell OpenGL the address of the array and one to actually submit all vertices. For large objects like the Stanford bunny, vertex arrays can save many thousands of OpenGL calls.

Even better are indexed vertex arrays: we know that a cube consists of 8 corners. So why should we transfer 24 vertices ? It is more efficient to define an array of these 8 corners and a second array to describe which corners to use to draw every single side of the cube. These are my arrays for a cube from (0,0,0) to (1,1,1):

```
// all 8 vertices of a cube organized as [x1,y1,z1], [x2,y2,yz], ...
static GLfloat vertices[] = { 0,0,0,
                              0,0,1,
                              0,1,0,
                              0,1,1,
                              1,0,0,
                              1,0,1,
                              1,1,0,
                              1,1,1 };

// 6 sides referring to the vertices defined above
static GLubyte indices[] = { 0,2,3,1,
                             4,5,7,6,
                             0,1,5,4,
                             2,6,7,3,
                             0,4,6,2,
                             1,3,7,5 };
```

To draw a single cube:

```
// initiate vertex array
glEnableClientState(GL_VERTEX_ARRAY);
// 3 dimensions (x,y,z), data type float, densely packed
glVertexPointer(3, GL_FLOAT, 0, vertices);
// and draw it
glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, indices);
// finished drawing
glDisableClientState(GL_VERTEX_ARRAY);
```

Since I want to draw 1,000 cubes, I use three nested loops and translate/scale all cubes based on the original one:

```
/// draw a frame
void Cubemania::redraw()
{
    // clear color and depth buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // safe current model-view matrix
    glPushMatrix();

    // slow rotation depending on timestamp
    glRotatef(seconds() * 30, 0, 1, 1);

    // all 8 vertices of a cube organised as [x1,y1,z1], [x2,y2,yz], ...
    static GLfloat vertices[] = { 0,0,0,
                                  0,0,1,
                                  0,1,0,
                                  0,1,1,
                                  1,0,0,
                                  1,0,1,
                                  1,1,0,
                                  1,1,1 };

    // 6 sides refering to the vertices defined above
    static GLubyte indices[] = { 0,2,3,1,
                                 4,5,7,6,
                                 0,1,5,4,
                                 2,6,7,3,
                                 0,4,6,2,
                                 1,3,7,5 };

    // initiate vertex array
    glEnableClientState(GL_VERTEX_ARRAY);
    // 3 dimensions (x,y,z), data type float, densely packed
    glVertexPointer(3, GL_FLOAT, 0, vertices);

    // translate cubes from [0,1]^3 to [-.5,+.5]^3
    glTranslatef(-0.5f, -0.5f, -0.5f);

    // number and size of cubes ... just modify to get even more cubes !
    const int   cubes    = 10;
    const float gridsize = 1.0f / cubes;
    const float length   = 0.3f * gridsize;

    // generate all these funky cubes ...
    for (float x = 0; x < 1; x += gridsize)
        for (float y = 0; y < 1; y += gridsize)
            for (float z = 0; z < 1; z += gridsize)
            {
                // save current model-view matrix
                glPushMatrix();

                // color is derived from position
                glColor3f(x,y,z);
                // place cube
                glTranslatef(x,y,z);
                glScalef(length,length,length);
                // and draw it
                glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, indices);

                // restore model-view matrix
                glPopMatrix();
            }

    // finished drawing
    glDisableClientState(GL_VERTEX_ARRAY);
    glPopMatrix();
}
```

Source Code

Cubemania.cpp

```
// ///////////////////////////////////////////////////////////
// Cubemania.cpp
// Copyright (c) 2004 Stephan Brumme. All rights reserved.
//

// just can't live without Windows ...
#include <windows.h>
// ... and OpenGL !
#include <gl/gl.h>
#include <gl/glu.h>

// my OpenGL window wrapper
#include "GLWindow.h"


/* of course the following 4 settings can be done in the project settings
   but I like to use #pragmas since I always keep on forgetting to set them
   in the Visual Studio project settings. I am getting older. */
// target Windows
#pragma comment (linker, "/SUBSYSTEM:WINDOWS")
#pragma comment (linker, "/ENTRY:mainCRTStartup")
// link required libraries
#pragma comment(lib, "opengl32.lib")
#pragma comment(lib, "glu32.lib")



/// Draw 10x10x10 = 1,000 spinning cubes
/**
    \class   Cubemania
    \author  Stephan Brumme, games@stephan-brumme.com
    \version 1.1: August 11, 2002
                  - Code cleanup
                  - added lots of comments
    \version 1.0: August 9, 2002
                  - Initial release
**/
class Cubemania : public GLWindow
{
public:
    /// Set up a new window, default depth of colorbuffer and depthbuffer
    Cubemania(const char* title, bool fullscreen, int width, int height)
        : GLWindow(fullscreen, title, width, height) {}

    /// override window initialization
    virtual void init();
    /// draw a frame
    virtual void redraw();
    /// recompute viewport
    virtual void resize(int newWidth, int newHeight);
};
```

```
/// override window initialization
/** Called by the baseclass GLWindow during window construction
**/
void Cubemania::init()
{
    // Gouraud shading
    glShadeModel(GL_SMOOTH);
    // black background
    glClearColor(0,0,0, 1);

    // z-buffer
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);

    // recommended by NeHe but I can't tell a difference ...
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);

    // backface culling, it's a bit faster
    glEnable(GL_CULL_FACE);

    // initial window size
    resize(width, height);
}


/// draw a frame
/** Called by the baseclass GLWindow during window construction
**/
void Cubemania::resize(int newWidth, int newHeight)
{
    // prepare resize
    GLWindow::resize(newWidth, newHeight);

    // perspective view
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60,  // field-of-view
                   newWidth/(float)newHeight, // aspect-ratio
                   1,   // front-z-plane
                   -1); // back-z-plane

    // set camera
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0, 0, 2,  // from (0,0,2)
              0, 0, 0,  // to (0,0,0)
              0, 1, 0); // up
}


/// draw a frame
void Cubemania::redraw()
{
    // clear color and depth buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // safe current model-view matrix
```

```
glPushMatrix();

// slow rotation depending on timestamp
glRotatef(seconds() * 30, 0, 1, 1);

// all 8 vertices of a cube organised as [x1,y1,z1], [x2,y2,yz], ...
static GLfloat vertices[] = { 0,0,0,
                              0,0,1,
                              0,1,0,
                              0,1,1,
                              1,0,0,
                              1,0,1,
                              1,1,0,
                              1,1,1 };

// 6 sides refering to the vertices defined above
static GLubyte indices[] = { 0,2,3,1,
                             4,5,7,6,
                             0,1,5,4,
                             2,6,7,3,
                             0,4,6,2,
                             1,3,7,5 };

// initiate vertex array
glEnableClientState(GL_VERTEX_ARRAY);
// 3 dimensions (x,y,z), data type float, densely packed
glVertexPointer(3, GL_FLOAT, 0, vertices);

// translate cubes from [0,1]^3 to [-.5,+.5]^3
glTranslatef(-0.5f, -0.5f, -0.5f);

// number and size of cubes ... just modify to get even more cubes !
const int   cubes    = 10;
const float gridsize = 1.0f / cubes;
const float length   = 0.3f * gridsize;

// generate all these funky cubes ...
for (float x = 0; x < 1; x += gridsize)
    for (float y = 0; y < 1; y += gridsize)
        for (float z = 0; z < 1; z += gridsize)
        {
            // save current model-view matrix
            glPushMatrix();

            // color is derived from position
            glColor3f(x,y,z);
            // place cube
            glTranslatef(x,y,z);
            glScalef(length,length,length);
            // and draw it
            glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, indices);

            // restore model-view matrix
            glPopMatrix();
        }

// finished drawing
glDisableClientState(GL_VERTEX_ARRAY);
glPopMatrix();
```

```
}



// here we go ;-)
int __cdecl main(int argc, char *argv[])
{
    // windowed mode, 512x512 pixels
    Cubemania opengl("Stephan's Cubemania", false, 512, 512);
    opengl.init();
    return opengl.run();
}


#ifdef _VERY_SMALL_EXE_
/* JUST DELETE THE FIRST 2 #PRAGMAS SINCE THEY ARE ONLY USEFUL TO PRODUCE
   A VERY SMALL EXECUTABLE (6K WITH VS.NET 2003) AND ARE CONSIDERED
   VERRRRRY DIRRRRRTY CODING STYLE !!!
   Linking against a VC6 lib in VS.NET 2003 (first #pragma) may lead to
   INSECURE, INSTABLE and CRASHING programs. But they are smaller ;-)

   Btw, if you compile on VS.NET 2003 and use the default system libraries
   (=> you removed the #pragmas) make sure to find MSVCR71.DLL on the
   target system. For VS7 this DLL is called somehow similar.
   These DLLs come usually with the Windows Service Packs but are not guar-
anteed
   to be installed on every system since they were not part of the original
   plain-vanilla Windows 98/XP+. On the other hand, VC6's MSVCRT.DLL always
exists
   on any Windows 98/XP+ based computer.
   */
#pragma comment(lib, "msvcrt_vc6.lib")
#pragma comment(linker, "/NODEFAULTLIB:libc.lib")
#pragma comment(exestr, "(C) http://www.stephan-brumme.com, " __DATE__ " @"
__TIME__)
  #endif
```

GLWindow.h

```
// ///////////////////////////////////////////////////////////
// GLWindow.h
// Copyright (c) 2004 Stephan Brumme. All rights reserved.
//

#ifndef _GLWINDOW_H_INCLUDED_
#define _GLWINDOW_H_INCLUDED_

// Windows-specific stuff
#include <windows.h>


/// Base class for an OpenGL window
/**
    \class   GLWindow
    \author  Stephan Brumme, games@stephan-brumme.com
    \version 1.1: August 11, 2002
                  - Code cleanup
                  - added lots of comments
    \version 1.0: August 9, 2002
                  - Initial release
**/
class GLWindow
{
public:
    /// Set up a new window
    GLWindow(bool fullscreen_, const char* title_,
             unsigned int width_, unsigned int height_,
             bool doublebuffer_ = true,
             unsigned char colorbuffer_ = 32, unsigned char depthbuffer_ =
32,
             unsigned char accumbuffer_ = 0);
    /// Destroy window
    ~GLWindow();


    /// OVERRIDE: Initialize OpenGL
    virtual void init() = 0;
    /// Main loop
    int run();
    /// Resize window
    virtual void resize(int width, int height);
    /// OVERRIDE: Redraw contents
    virtual void redraw() = 0;
    /// Free used stuff
    virtual void destroy() { }

    /// Toggle fullscreen/windowed
    void toggleFullscreen();
    /// Toggle active/stopped
    void toggleActive();
    /// Set active/stopped
    void setActive(bool newState);
    /// Paint window
    void paint();
    /// Returns number of seconds the program was active
    float seconds() const;


protected:
    /// Window's handle
    HWND  hWnd;
    /// Device context
    HDC   hDC;
    /// Resource context
    HGLRC hRC;
    /// Program ID
    HINSTANCE hInstance;

    /// Width of the viewport
    int width;
    /// Height of the viewport
    int height;
    /// True, if using double buffer technique
    bool doublebuffer;
    /// True, if fullscreen
    bool fullscreen;
    /// Bits of the color buffer (default: 32, RGBA)
    unsigned char colorbuffer;
    /// Bits of the depth buffer (default: 32)
    unsigned char depthbuffer;
    /// Bits of the accumulation buffer (default: 0)
    unsigned char accumbuffer;

    /// True, if window is active
    bool active;

    /// Window's caption
    const char* title;
    /// Drawn frames
    int frames;
    /// Time of initial window creation
    unsigned int timeStarted;
    /// Time spent in inactive mode
    unsigned int timeInactive;
    /// Time of last activity
    unsigned int inactiveSince;


private:
    /// Ensure singleton pattern
    static GLWindow* singleton;

    /// Process message queue
    static LRESULT CALLBACK message(HWND hWnd, UINT message, WPARAM wParam,
LPARAM lParam);

    enum
    {
        /// Hopefully unique ID
        FPS_TIMER_ID       = 0x12345679,
        /// Update interval
        FPS_TIMER_INTERVAL = 1000
    };

    static unsigned int framesAtLastFPSUpdate;
};


#endif // _GLWINDOW_H_INCLUDED_
```

GLWindow.cpp

```cpp
// //////////////////////////////////////////////////////////////
// GLWindow.cpp
// Copyright (c) 2004 Stephan Brumme. All rights reserved.
//

#include "GLWindow.h"

// ... and OpenGL itself !
#include <gl/gl.h>
// basic string processing
#include <cstdio>
// debugging
#include <cassert>


//  Static variables
/// Initialize singleton
GLWindow* GLWindow::singleton = NULL;
/// Number of frames drawn at last update
/** Used to compute the number of frames drawn in the last interval:
    it's just (frames-framesAtLastFPSUpdate) **/
unsigned int GLWindow::framesAtLastFPSUpdate = 0;



/// Set up a new window
GLWindow::GLWindow(bool fullscreen_, const char* title_,
                   unsigned int width_, unsigned int height_,
                   bool doublebuffer_,
                   unsigned char colorbuffer_, unsigned char depthbuffer_,
                   unsigned char accumbuffer_)
    : fullscreen(fullscreen_),
      title(title_),
      width(width_), height(height_),

      doublebuffer(doublebuffer_),
      accumbuffer(accumbuffer_),
      colorbuffer(colorbuffer_),
      depthbuffer(depthbuffer_),

      active(true), timeInactive(0),
      frames(0)
{
    // singleton pattern
    assert(singleton == NULL);
    singleton = this;

    // create window
    hInstance = GetModuleHandle(NULL);

    // register window class
    WNDCLASSEX window;
    window.cbSize        = sizeof(WNDCLASSEX);
    window.style         = CS_HREDRAW | CS_VREDRAW | CS_OWNDC | CS_DBLCLKS;
    window.lpfnWndProc   = message;
    window.cbClsExtra    = 0;
    window.cbWndExtra    = 0;
    window.hInstance     = hInstance;
    window.hIcon         =
    window.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);
    window.hCursor       = LoadCursor(NULL, IDC_ARROW);
    window.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
    window.lpszMenuName  = NULL;
    window.lpszClassName = title;

    // register my window
    bool registerClass = RegisterClassEx(&window) != 0;
    assert(registerClass);

    // window style depends whether window is in fullscreen of windowed
    DWORD dwStyle   = WS_OVERLAPPEDWINDOW;
    DWORD dwExStyle = WS_EX_APPWINDOW;
    if (fullscreen)
    {
        dwStyle   = WS_POPUP | WS_MAXIMIZE;
        dwExStyle = WS_EX_APPWINDOW | WS_EX_WINDOWEDGE;
    }

    // ensure window size
    RECT size;
    size.left   = 0;
    size.top    = 0;
    size.right  = width;
    size.bottom = height;
    AdjustWindowRectEx(&size, dwStyle, FALSE, dwExStyle);

    // create main window
    hWnd = CreateWindowEx(dwExStyle,
                          title, title,
                          dwStyle | WS_CLIPSIBLINGS | WS_CLIPCHILDREN,
                          256,128, width,height,
                          NULL, NULL, hInstance, NULL);
    assert(hWnd);


    // get the device context
    hDC = GetDC(hWnd);
    assert(hDC);

    // set pixel format
    PIXELFORMATDESCRIPTOR pfd;
    ZeroMemory(&pfd, sizeof(pfd));
    pfd.nSize      = sizeof(pfd);
    pfd.nVersion   = 1;
    pfd.dwFlags    = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | (double-
buffer ? PFD_DOUBLEBUFFER : 0);
    pfd.iPixelType = PFD_TYPE_RGBA;
    pfd.cColorBits = colorbuffer;
    pfd.cDepthBits = depthbuffer;
    pfd.cAccumBits = accumbuffer;
    pfd.iLayerType = PFD_MAIN_PLANE;

    // set buffers' depths
    int format = ChoosePixelFormat(hDC, &pfd);
    assert(format > 0);
    bool setPixelFormat = SetPixelFormat(hDC, format, &pfd) != 0;
    assert(setPixelFormat);

    // create and enable the render context
    hRC = wglCreateContext(hDC);
    assert(hRC);
    wglMakeCurrent(hDC, hRC);

    // start timer
    SetTimer(hWnd, FPS_TIMER_ID, FPS_TIMER_INTERVAL, 0);

    // disable mouse cursor in fullscreen mode
    if (fullscreen)
        ShowCursor(FALSE);

    // show window
    AnimateWindow(hWnd, 200, AW_BLEND | AW_ACTIVATE);
    SetForegroundWindow(hWnd);
    SetFocus(hWnd);

    // switch off vsync if possible
    typedef GLboolean (__stdcall * SwapProc) (GLint interval);
    SwapProc wglSwapIntervalEXT =
        (SwapProc) wglGetProcAddress("wglSwapIntervalEXT");
    if (wglSwapIntervalEXT)
        wglSwapIntervalEXT(0);


    // ensure proper timing
    timeStarted = GetTickCount();
}


/// Destroy window
GLWindow::~GLWindow()
{
    // yeah, it's time to say good-bye ...

    // delete timer
    KillTimer(hWnd, FPS_TIMER_ID);

    // free device context
    wglMakeCurrent(NULL, NULL);
    wglDeleteContext(hRC);
    ReleaseDC(hWnd, hDC);

    // show cursor
    if (fullscreen)
        ShowCursor(TRUE);

    // properly remove window from screen
    AnimateWindow(hWnd, 500, AW_BLEND | AW_HIDE);
    DestroyWindow(hWnd);
    UnregisterClass(title, hInstance);
}


/// Main loop
```

```
int GLWindow::run()
{
    // run forever :-)
    while (true)
    {
        MSG msg;
        // quit ?
        while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
            // window closed, the only way of escaping the "run" method
            if (msg.message == WM_QUIT)
                return (int)msg.wParam;

            // keep message pipeline alive
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }

        // redraw window
        if (active)
            paint();
        else
            // idle, so don't waste CPU time
            WaitMessage();
    }
}


/// Resize window
void GLWindow::resize(int newWidth, int newHeight)
{
    // store new window size
    width  = newWidth;
    height = newHeight;

    // adjust OpenGL view
    glViewport(0, 0, width-1, height-1);
}


/// Toggle fullscreen/windowed
void GLWindow::toggleFullscreen()
{
    // default windowed size if programs starts in fullscreen mode
    static int windowedWidth  = 512;
    static int windowedHeight = 512;


    if (fullscreen)
    {
        // switch from fullscreen to windowed mode

        // set new window styles
        SetWindowLong(hWnd, GWL_STYLE,   WS_OVERLAPPEDWINDOW |
WS_CLIPSIBLINGS | WS_CLIPCHILDREN);
        SetWindowLong(hWnd, GWL_EXSTYLE, WS_EX_APPWINDOW);

        // from maximized to normal mode
        ShowWindow(hWnd, SW_RESTORE);
        UpdateWindow(hWnd);

        // old window size
        width  = windowedWidth;
        height = windowedHeight;
        SetWindowPos(hWnd, HWND_TOP, 256,128, width,height,
SWP_SHOWWINDOW);

        // show mouse cursor
        ShowCursor(TRUE);
    }
    else
    {
        // switch from windowed mode to fullscreen
        windowedWidth  = width;
        windowedHeight = height;

        // set new window styles
        SetWindowLong(hWnd, GWL_STYLE,   WS_POPUP | WS_CLIPSIBLINGS |
WS_CLIPCHILDREN);
        SetWindowLong(hWnd, GWL_EXSTYLE, WS_EX_APPWINDOW |
WS_EX_WINDOWEDGE);

        // show window maximized
        ShowWindow(hWnd, SW_SHOWMAXIMIZED);
        UpdateWindow(hWnd);

        // hide mouse cursor
        ShowCursor(FALSE);
    }

    fullscreen = !fullscreen;
}


/// Toggle active/stopped
void GLWindow::toggleActive()
{
    setActive(!active);
}


/// Set active/stopped
void GLWindow::setActive(bool newState)
{
    // same state ?
    if (active == newState)
        // do nothing ...
        return;


    // well, the state changed indeed
    active = newState;

    if (!active)
        // entering inactive mode
        inactiveSince = GetTickCount();
    else
        // going back to active mode, compute time spent being inactive
        timeInactive += GetTickCount() - inactiveSince;
}


/// Paint window
void GLWindow::paint()
{
    // redraw window contents
    redraw();
    // frame counter
    frames++;
    // double-buffering: switch front and back buffer
    if (doublebuffer)
        SwapBuffers(hDC);
}


/// Returns seconds since was was opened
float GLWindow::seconds() const
{
    if (active)
        return (GetTickCount() - (timeStarted + timeInactive)) / 1000.0f;
    else
        return (inactiveSince - (timeStarted + timeInactive)) / 1000.0f;
}


/// Process message queue
LRESULT CALLBACK GLWindow::message(HWND hWnd, UINT uMsg, WPARAM wParam,
LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_SYSCOMMAND:
        {
            switch (wParam)
            {
                // disable screensavers or monitor standby
                // idea taken from http://nehe.gamedev.net
                case SC_SCREENSAVE:
                case SC_MONITORPOWER:
                return 0;
            }
            break;
        }

        // restart/pause execution
        case WM_ACTIVATE:
        {
            singleton->setActive(!HIWORD(wParam));
            return 0;
        }

        case WM_KEYDOWN:
        {
            switch (wParam)
            {
            // quit
            case VK_ESCAPE:
                PostMessage(hWnd, WM_CLOSE, 0, 0);
                return 0;

            // toggle fullscreen/windowed
            case VK_F5:
                singleton->toggleFullscreen();
                return 0;

            // pause execution
            case VK_PAUSE:
            case VK_SPACE:
                singleton->toggleActive();
                return 0;
            }

            break;
        }

        // quit
        case WM_CLOSE:
        {
            PostQuitMessage(0);
            return 0;
        }

        // redraw window
        case WM_PAINT:
        {
            singleton->paint();

            // required to avoid artefacts
            static PAINTSTRUCT ps;
            BeginPaint(hWnd, &ps);
            EndPaint(hWnd, &ps);
            break;
        }

        // resize
        case WM_SIZE:
        {
            singleton->resize(LOWORD(lParam), HIWORD(lParam));
            PostMessage(hWnd, WM_PAINT, 0, 0);
            return 0;
        }

        // timer
        case WM_TIMER:
            // framerate timer
            if(wParam == FPS_TIMER_ID)
            {
                // compute current framerate
                float fps = (singleton->frames - framesAtLastFPSUpdate)
                            * (1000.0f / FPS_TIMER_INTERVAL);
                framesAtLastFPSUpdate = singleton->frames;

                // update caption bar
                static char newCaption[257];
                sprintf(newCaption, "%s (%.0f fps)", singleton->title,
fps);
```

```
                SetWindowText(hWnd, newCaption);

                return 0;
            }
            break;
    }

    // default handler
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
```