Omnimap Geometry Correction Lib

Developer Requirements

WindowsXP

New Nvidia GPU

Visual Studio 2005

1. Extract hallway.rar

2. Extract omnimap.rar into the source directory

There are three directories created

A lib directory

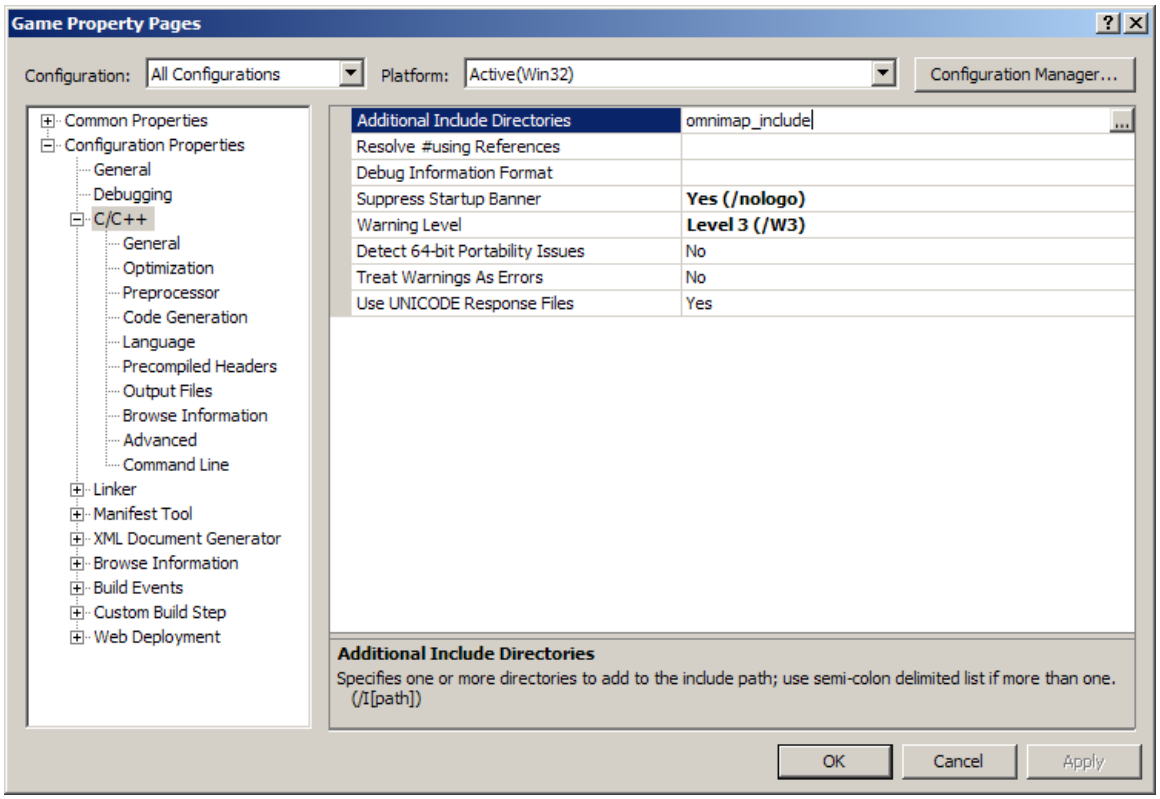
A include directory

And a config directory

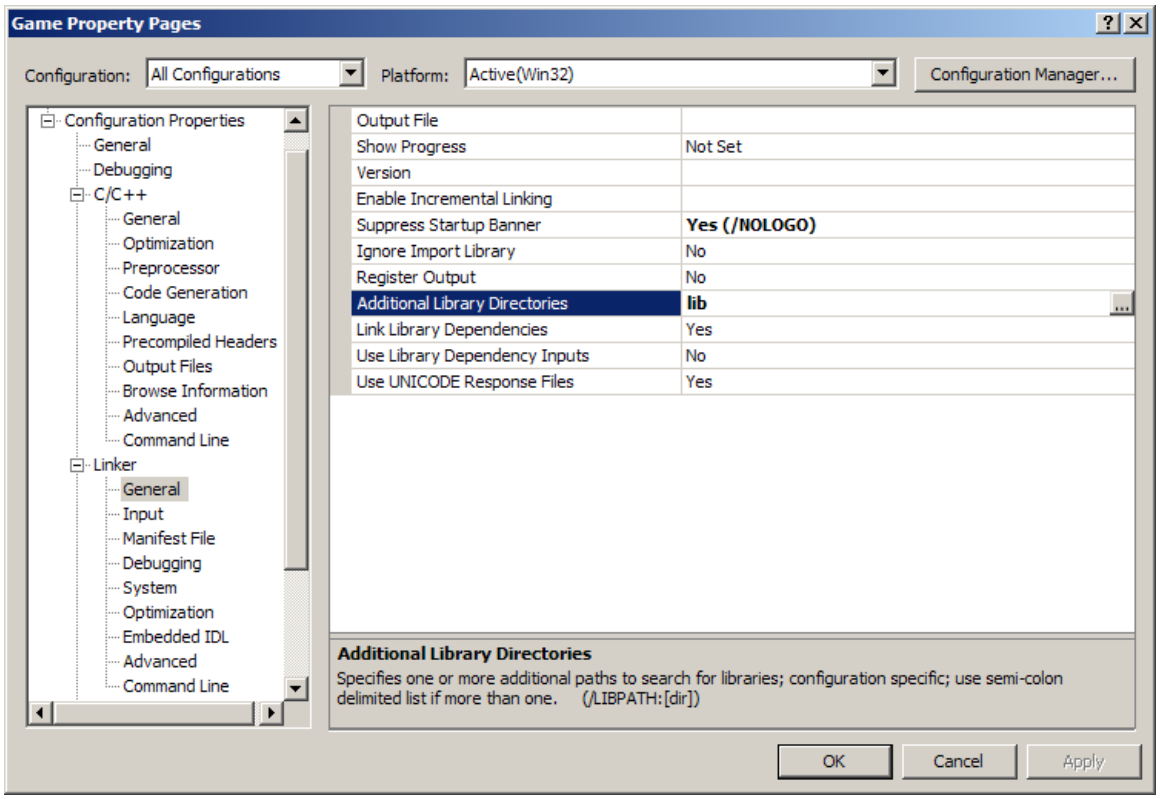
Plus some dll files are extracted into the main directory

3. Setting Up the Visual Studio Workspace

Go to the property page of the workspace



Add omnimap_include as an additional include directory



Add lib as an additional library directory

Edit Basecode.cpp

```
// ** SYSTEM LIBRARIES **
// ----------------------
#pragma comment(lib, "opengl32.lib")
#pragma comment(lib, "glu32.lib")
#pragma comment(lib, "glaux.lib")
#pragma comment(lib, "winmm.lib")     // Sound
```

```
#include "omnimap.h"
// ** SYSTEM HEADER FILES **
// ------------------------
#include <gl\glew.h>
#include <windows.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <stdio.h>


OmniMap *OmnimapLib=0;


void Render();
void SetupRenderChannelTextureContext();

void fun(pOmniMap_Channel chan)
{


            chan->beginRenderToChannel();
            gluPerspective(120.0f, (float)OmnimapLib->resWidth /
(float)OmnimapLib->resHeight, 2, 150.0f);
// Aspect ratio of window
            SetupRenderChannelTextureContext();
          Render();
            chan->endRenderToChannel();

}


// ** DECLARATIONS **
// ------------------
HDC           hDC   = NULL;           // Device context
HGLRC    hRC   = NULL;           // Rendering context
HWND     hWnd = NULL;           // Window handle
HINSTANCE hInstance;           // Instance of application

int SCREEN_WIDTH  = 1024;
int SCREEN_HEIGHT = 768;
int SCREEN_DEPTH  = 16;

bool active       = true;
bool fullScreen = true;

float frameInterval = 0.0f;


// ** FUNCTION PROTOTYPES **
// -------------------------

// Misc
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
```

```
// Map
void LoadGLTextures();
void LoadMap();
void RenderScene();

// FX
void RenderSteam(float);
void RenderBubbles(float, int, float);
void RenderSpace(float);
void RenderFade(float);
void RenderDoors();

// Movement
void OpenDoor(float);
void MoveCamera(float);


// ** RESIZE SCREEN **
// -------------------
void ReSizeScreen(int width, int height)
{
	// Prevent a divide by zero
	if (height == 0)
	{
		height = 1;
	}

	//		(x, y, width, height)
	glViewport(0, 0, width, height);	// Viewport wholescreen, could
make smaller
	glMatrixMode(GL_PROJECTION);		// Set projection matrix
	glLoadIdentity();				// Reset projection
matrix


	// NOTE: FOV below is set to 90 to give impression tunnel is longer
than it actually is,
	//	normally this would be 45 degrees.

	//		(FOV, width -> height ratio, closest clip, furthest
clip)
	gluPerspective(90.0f, (float)width / (float)height, 2, 150.0f);  //
Aspect ratio of window

	glMatrixMode(GL_MODELVIEW);			// Set modelview matrix
	glLoadIdentity();				// Reset modelview
matrix

	if(OmnimapLib)
	{
		OmnimapLib->resWidth =width;
		OmnimapLib->resHeight =height;
		OmnimapLib->ScriptingEngine->RunString("onResize()");
```

```
OmnimapLib->ScriptingEngine->RunString("ConsolePrintString(\"calling
onResize() after switching to res %d,%d\")",width,height);
	}

}


// ** INITIALISE OPENGL **
// -----------------------
int InitOpenGL()
{
	glEnable(GL_TEXTURE_2D);                                        //
Enable texture mapping

	glBlendFunc(GL_SRC_ALPHA,GL_ONE);                               // Set
blending function for translucency
	glShadeModel(GL_SMOOTH);                                        //
Enable smooth shading
	glClearColor(0.0f, 0.0f, 0.0f, 1.0f);                           //
Background
	glClearDepth(1.0f);
	// Depth buffer
	glEnable(GL_DEPTH_TEST);                                        //
Enable depth testing
	glDepthFunc(GL_LESS);
// Type of depth testing
	glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);      //
Perspective calculations

	// Load from files
	LoadGLTextures();
	LoadMap();

	// Fogging
	float fogColor[4] = {0.0f, 0.0f, 0.0f, 1.0f};

	glFogi  (GL_FOG_MODE, GL_LINEAR);          // Type of fogging
	glFogfv (GL_FOG_COLOR, fogColor);          // Fog colour (black)
	glFogf  (GL_FOG_DENSITY, 0.2f);                  // Density
	glHint  (GL_FOG_HINT, GL_DONT_CARE); // Perspective calculations
	glFogf  (GL_FOG_START, 120.0f);                  // Start of
fogging from camera
	glFogf  (GL_FOG_END, 150.0f);              // End of fogging
	glEnable(GL_FOG);                                // Enable fogging

	// Env Mapping
	glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP); // Set sphere
texture generation mapping for S
	glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP); // Set sphere
texture generation mapping for T

	PlaySound("audio/music.wav", NULL, SND_FILENAME | SND_ASYNC |
SND_LOOP);   // Play music loop
```

```
	return true;
}


// ** DE-INITIALISE OPENGL & WINDOW **
// -----------------------------------
void DeInit()
{
	// Stop audio
	PlaySound(NULL, NULL, SND_PURGE);

	// Return to desktop
	if (fullScreen)
	{
		ChangeDisplaySettings(NULL, 0);
		ShowCursor(true);
	}

	// Release and delete RC
	if (hRC)
	{
		if (!wglMakeCurrent(NULL, NULL))
			MessageBox(NULL, "Failed to release DC & RC", "DoH!",
MB_OK | MB_ICONEXCLAMATION);
		if (!wglDeleteContext(hRC))
			MessageBox(NULL, "Failed to release rendering context",
"DoH!", MB_OK | MB_ICONEXCLAMATION);

		hRC = NULL;
	}

	// Release DC
	if (hDC && !ReleaseDC(hWnd, hDC))
	{
		MessageBox(NULL, "Failed to release device context", "DoH!",
MB_OK | MB_ICONEXCLAMATION);
		hDC = NULL;
	}

	// Destroy window
	if (hWnd && !DestroyWindow(hWnd))
	{
		MessageBox(NULL, "Failed to release hWnd", "DoH!", MB_OK |
MB_ICONEXCLAMATION);
		hWnd = NULL;
	}

	// Unregister class
	if (!UnregisterClass("Game", hInstance))
	{
		MessageBox(NULL, "Failed to unregister class", "DoH!", MB_OK
| MB_ICONEXCLAMATION);
		hInstance = NULL;
```

```
	}
}


// ** SETUP PIXEL FORMAT **
// ------------------------
bool SetupPixelFormat()
{
	int PixelFormat;
	staticPIXELFORMATDESCRIPTOR pfd=			// pfd says
how we want the window to be
	{
		sizeof(PIXELFORMATDESCRIPTOR),			// Size of
pixel format descriptor
		1,
	// Version number (always 1)
		PFD_DRAW_TO_WINDOW |					//
Format to support window
		PFD_SUPPORT_OPENGL |					//
Format to support opengl
		PFD_DOUBLEBUFFER,					//
Format to support double buffering
		PFD_TYPE_RGBA,
// Request RGBA format
		SCREEN_DEPTH,
// Select color depth
		0, 0, 0, 0, 0, 0,					//
Color bits ignored
		0,
	// No alpha buffer
		0,
	// Shift bit ignored
		0,
	// No accumulation buffer
		0, 0, 0, 0,
// Accumulation bits ignored
		16,
	// 16Bit z-buffer (depth buffer)
		0,
	// No stencil buffer
		0,
	// No auxiliary buffer
		PFD_MAIN_PLANE,
// Main drawing layer
		0,
	// Reserved
		0, 0, 0
	// Layer masks ignored
	};

	// Get a device context
	if (!(hDC=GetDC(hWnd)))
	{
		DeInit();
```

```
            MessageBox(NULL, "Failed to create a device context", "DoH!",
MB_OK|MB_ICONEXCLAMATION);
            return false;
      }

      // Choose a pixel format that best matches above
      if (!(PixelFormat=ChoosePixelFormat(hDC,&pfd)))
      {
            DeInit();
            MessageBox(NULL, "Failed to find a suitable pixelformat",
"DoH!", MB_OK|MB_ICONEXCLAMATION);
            return false;
      }

      // Set pixel format chosen above
      if(!SetPixelFormat(hDC,PixelFormat,&pfd))
      {
            DeInit();
            MessageBox(NULL, "Failed to set the pixelformat", "DoH!",
MB_OK|MB_ICONEXCLAMATION);
            return false;
      }

      // Get a rendering context
      if (!(hRC=wglCreateContext(hDC)))
      {
            DeInit();
            MessageBox(NULL, "Failed to create a rendering context",
"DoH!", MB_OK|MB_ICONEXCLAMATION);
            return false;
      }

      // Activate rendering context
      if(!wglMakeCurrent(hDC,hRC))
      {
            DeInit();
            MessageBox(NULL, "Failed to activate the rendering context",
"DoH!", MB_OK|MB_ICONEXCLAMATION);
            return false;
      }

      return true;
}


// ** CREATE THE WINDOW **
// -----------------------
bool CreateGLWindow(char* title)
{
      WNDCLASS wndClass;       // Windows class structure
      DWORD    dwExStyle;            // Windows extended style
      DWORD    dwStyle;        // Windows style
      RECT     wndRect;        // Windows dimensions
```

```
	wndRect.left   = 0;
	wndRect.right  = SCREEN_WIDTH;
	wndRect.top    = 0;
	wndRect.bottom = SCREEN_HEIGHT;

	hInstance = GetModuleHandle(NULL);   // Grab an instance for the
window

	wndClass.style = CS_HREDRAW |                                    //
Redraw on horizontal size
                         CS_VREDRAW |
// Redraw on vertical size
                                CS_OWNDC;
        // Window has own DC
	wndClass.lpfnWndProc   = (WNDPROC)WndProc;               // Handles
the windows messages
	wndClass.cbClsExtra    = 0;
// No extra window data
	wndClass.cbWndExtra    = 0;
// No extra window data
	wndClass.hInstance     = hInstance;                              //
Set the instance
	wndClass.hIcon         = LoadIcon(NULL, IDI_WINLOGO); // Default
icon
	wndClass.hCursor       = LoadCursor(NULL, IDC_ARROW); // Default
cursor
	wndClass.hbrBackground = NULL;                                   //
No background
	wndClass.lpszMenuName  = NULL;                                   //
No menu
	wndClass.lpszClassName = "Game";                                 //
Class name

	if (!RegisterClass(&wndClass)) // Register window class
	{
		MessageBox(NULL, "Failed to register window class", "DoH!",
MB_OK | MB_ICONEXCLAMATION);
		return false;
	}

	if (fullScreen)
	{
		DEVMODE dmSettings;
	// Device mode
		memset(&dmSettings, 0, sizeof(dmSettings));      // Clear
memory
		dmSettings.dmSize       = sizeof(dmSettings);    // Size of
devmode structure
		dmSettings.dmPelsWidth  = SCREEN_WIDTH;                  //
Screen width
		dmSettings.dmPelsHeight = SCREEN_HEIGHT;         // Screen
height
		dmSettings.dmBitsPerPel = SCREEN_DEPTH;                  //
Bits per pixel
```

```
            dmSettings.dmFields     = DM_BITSPERPEL |
                                                  DM_PELSWIDTH  |
                                                  DM_PELSHEIGHT;

            // Try set selected mode
            if (ChangeDisplaySettings(&dmSettings, CDS_FULLSCREEN)!=
DISP_CHANGE_SUCCESSFUL)
            {
                  if (MessageBox(NULL, "Fullscreen mode not supported by
your video card\nDo you want to use windowed mode instead?",
                                          "Screen mode", MB_YESNO |
MB_ICONQUESTION)==IDYES)
                  {
                        fullScreen = false;
                  }
                  else
                  {
                        return false;
                  }
            }
      }

      if (fullScreen)          // If still in fullscreen
      {
            dwExStyle = WS_EX_APPWINDOW;                              //
Window extended style
            dwStyle   = WS_POPUP;
// Window style
            ShowCursor(false);
      }
      else
      {
            dwExStyle = WS_EX_APPWINDOW | WS_EX_WINDOWEDGE;  // Window
extended style
            dwStyle   = WS_OVERLAPPEDWINDOW;                          //
Window style
      }


      AdjustWindowRectEx(&wndRect, dwStyle, false, dwExStyle);      //
Adjust window to size

      // Create the window
      if (!(hWnd = CreateWindowEx(dwExStyle,
      // Extended style
                                          "Game",
                  // Class name
                                          title,
                  // Window title
                                          dwStyle |
            // Defined window style
                                          WS_CLIPSIBLINGS |
            // Required window style
```

```
                                                WS_CLIPCHILDREN,
                  // Required window style
                                                0, 0,
                  // Window position
                                                wndRect.right  -
wndRect.left,     // Window width
                                                wndRect.bottom -
wndRect.top, // Window height
                                                NULL,
                  // No parent window
                                                NULL,
                  // No menu
                                                hInstance,
                  // Instance
                                                NULL)))
                  // Don't pass anything to WM_CREATE
      {
            DeInit();
            MessageBox(NULL, "Failed to create window", "DoH!", MB_OK |
MB_ICONEXCLAMATION);
            return false;
      }

      // Setup the pixel format
      if (!SetupPixelFormat())
      {
            return false;
      }

      // Finalise window
      ShowWindow(hWnd, SW_SHOW);
      SetForegroundWindow(hWnd);
      SetFocus(hWnd);
      ReSizeScreen(SCREEN_WIDTH, SCREEN_HEIGHT);

      // Initialise opengl
      if (!InitOpenGL())
      {
            DeInit();
            MessageBox(hWnd, "Failed to initialise", "DoH!", MB_OK |
MB_ICONEXCLAMATION);
            return false;
      }

      return true;
}


// ** CALCULATE FRAMERATE **
// -------------------------
void CalculateFrameRate()
{
      static float FPS       = 0.0f; // Frames per second
      static float lastTime  = 0.0f; // Time from last frame
```

```
	static float frameTime = 0.0f; // Current frame time
	char strFrameRate[10] = {0};   // Window title

	// Get current time in seconds
	float currentTime = GetTickCount() * 0.001f;

	// Set frame individual
	frameInterval = currentTime - frameTime;
	frameTime = currentTime;

	// Increase frame counter
	FPS++;

	// If a second has passed refresh FPS
	if (currentTime - lastTime > 1.0f)
	{
		lastTime = currentTime;

		// Show FPS in title bar
		sprintf(strFrameRate, "FPS: %d", int(FPS));
		SetWindowText(hWnd, strFrameRate);

		FPS = 0.0f; // Reset counter
	}
}


void onQuit()
{
	if(OmnimapLib)
		delete OmnimapLib;
	OmnimapLib=0;

}

// ** WndProc - Handles Window Messages **
// ---------------------------------------
LRESULT CALLBACK WndProc(HWND  hWnd,			// Window handle
					UINT uMsg,			// Message
for window
					WPARAM	wParam,
// Additional message info
					LPARAM	lParam)
// Additional message info
{
	switch (uMsg)							//
Check for window messages
	{

	// Windows active state changes
	case WM_ACTIVATE:
		{
			if (!HIWORD(wParam))			// Check
minimisation state
```

```
                {
                        active = true;
                }
                else
                {
                        active = false;
                }

                return 0;
        }

    // Power saving operations
    case WM_SYSCOMMAND:
        {
                switch (wParam)
                {
                        case SC_SCREENSAVE:
                        case SC_MONITORPOWER:
                                return 0;
                }
                break;
        }

    // Window closed
    case WM_CLOSE:
        {
                PostQuitMessage(0);
                onQuit();// delete OmnimapLib on exit
                return 0;
        }

    // Escape key pressed
    case WM_KEYDOWN:
        {
                if (wParam == VK_ESCAPE)
                {
                        onQuit();// delete OmnimapLib on exit
                        PostQuitMessage(0);
                }

                return 0;
        }

    // Screen resized
    case WM_SIZE:
        {
                ReSizeScreen(LOWORD(lParam),HIWORD(lParam));  // LOWORD
= width, HIWORD = height
                return 0;
        }
    }

    // Pass uhandled messages to DefWindowProc
    return DefWindowProc(hWnd,uMsg,wParam,lParam);
```

```
}


// all rendering calls are put into function so it can be called in the
render channel loop.

void Render()
{

                        glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);    // Clear screen and depth buffer
                        glLoadIdentity();
                  // Reset view

                        RenderSpace(frameInterval);
                        RenderDoors();
                        RenderScene();

                        // Render particles
                        glDepthMask(GL_FALSE);         // Disable
z-buffer writes

                              RenderSteam(frameInterval);

                              RenderBubbles(-2.5, 1, frameInterval);

                              RenderBubbles(13.5, 2, frameInterval);


                        glDepthMask(GL_TRUE);          // Re-enable
z-buffer writes
                        glFlush();

                        RenderFade(frameInterval);

}

void SetupRenderChannelTextureContext()
{
glEnable(GL_TEXTURE_2D);
      glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
      glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);

}

void SetupProjectiveTexturingTextureContext()
{

      glEnable(GL_TEXTURE_2D);
      glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
      glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
}
// ** MAIN LOOP **
// ---------------
```

```
WPARAM MainLoop()
{
	MSG msg;

	while(1)
	{
		if(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))	// Check for
message
		{
			if (msg.message == WM_QUIT)
			{
				break;
			}
			else
			{
				TranslateMessage(&msg);		// Find out what
message does
				DispatchMessage(&msg);		// Run message
			}
		}
		else
		{
			if (active)
			{
// any update must happen outside of the channel render loop
				OpenDoor(frameInterval);
				MoveCamera(frameInterval);
				// channel render loop
				ForEachChannel( OmnimapLib, fun);
				SetupProjectiveTexturingTextureContext();
				OmnimapLib->PostRender();


				SwapBuffers(hDC);

				CalculateFrameRate();
			}
		}
	}

	DeInit();
	return (msg.wParam);	// Return from program
}


// ** MAIN **
// ----------
int WINAPI WinMain(HINSTANCE hInstance,		// Instance
			HINSTANCE hPrevInstance,	// Previous
instance
			LPSTR	lpCmdLine,		// Command line
parameters
			int		nCmdShow)		// Window show
state
```

```
{

	// skip screen mode
if(false)
	/*if (MessageBox(NULL, "Fullscreen? (Recommended)", "Screen mode",
MB_YESNO | MB_ICONQUESTION) == IDYES)*/
	{
		if (MessageBox(NULL, "1024 * 768 resolution? (Recommended)",
"Screen mode", MB_YESNO | MB_ICONQUESTION) == IDYES)
		{
			SCREEN_WIDTH  = 1024;
			SCREEN_HEIGHT = 768;
		}
		else
		{
			SCREEN_WIDTH  = 800;
			SCREEN_HEIGHT = 600;
		}
	}
	else
	{
		fullScreen = false;
		SCREEN_WIDTH  = 640;
		SCREEN_HEIGHT = 480;
	}



	// Create window
	if (!CreateGLWindow("3D Engine"))
		return 0;

	// create object
	OmnimapLib = new OmniMap(SCREEN_WIDTH,SCREEN_HEIGHT);

	// Run main loop

	return MainLoop();
}
```

Compile and run!