OmniMap API OpenGL Integration Guide

Version 0.9.1

Last updated: March 8, 2007

IMMERSIVE PROJECTION DESIGN
THE ELUMENATI
www.elumenati.com

Contents

© 2007 The Elumenati, LLC. All rights reserved. The Elumenati and the Elumenati logo are registered trademarks of The Elumenati, LLC. Other trademarks and brands are the property of their respective owners. The information in this document belongs to The Elumenati, LLC.

Notice of non-liability:
The Elumenati, LLC is providing the information in this document to you AS-IS with all faults. The Elumenati, LLC makes no warranties of any kind (whether express, implied or statutory) with respect to the information contained herein. The Elumenati, LLC assumes no liability for damages (whether direct or indirect), caused by errors or omissions, or resulting from the use of this document or the information contained in this document or resulting from the application or use of the product or service described herein. The Elumenati, LLC reserves the right to make changes to any information herein without further notice.

Please send any questions or comments to support@elumenati.com.

Overview

This document is a guide for integrating the OmniMap Geometry Correction Library API into existing OpenGL applications. The document will provide an overview of the library's architecture, and how it fits into an existing OpenGL application, followed by a case study of the steps required to integrate the OmniMap API into a sample OpenGL program.

For more information, see the *OmniMap API Configuration Guide* and the *OmniMap API Class Documentation* included with the OmniMap installer.

Intended Audience

This document is written for programmers interested in implementing geometry correction within their real-time OpenGL application for use with the Elumenati OmniFocus range of products.

Developer Requirements

- Microsoft WindowsXP
- GPU supporting OpenGL 2.0
- Visual Studio 2005

Assumptions

This document assumes the developer has installed the OmniMap API from the installer program *OmniMap.msi*,and used the default location for the installation. The default installation location for the OmniMap API is:

C:\Program Files\Elumenati\OmniMap API

In this document, we will refer to this folder as <*InstallationDir*>. So if you have chosen to install the OmniMap API in a location other than the default location, <*InstallationDir*> refers to that location.

OmniMap API Components

The OmniMap API is shipped with documentation, the example program referred to in this document, C++ header files, lib files and corresponding DLLs. The documentation includes this document, the API Configuration document, and detailed class documentation in HTML format.

The class documentation can be found in:

<InstallationDir>/docs/classdocs

The header files are found in:

<InstallationDir>/include

The library files can be found in:

<InstallationDir>/lib

The DLL's can be found in:

<InstallationDir>/bin

The example program can be found in:

<InstallationDir>/examples

OmniMap API Architecture

The OmniMap API is designed to be easily integrated into existing OpenGL applications. The main class that provides the dome rendering functionality is called *OmniMap*. The *OmniMap object* is constructed with arguments for the width and height of the final output:

OmniMap omniMapObj = OmniMap::OmniMap(width, height);

To generate a scene for a dome, the OmniMap class needs to call the application's rendering function 3 or 4 times with a modified viewing frustum and viewing angle. The rotation and viewing angles of each channel are dependent upon the dome configuration specified in the Lua script files *omnimap_startup.lua*, and *omnimap_user_edit.lua*. Each of these render passes is managed by the *OmniMapChannelBase* class. In the application where the rendering of a single frame is done, a call is made to the *OmniMap* object telling the *OmniMap* object which function to call for rendering a single channel. That call looks like:

omniMapObj->ForEachChannel(renderFunction);

where *renderFunction* is a pointer to a function that takes a pointer to an *OmniMapChannelBase* object as an argument:

*void renderFunction(OmniMapChannelBase *channel);*

ForEachChannel will call the *renderFunction* once for each channel. Prior to making that call, the OpenGL projection matrix has been set to the projection required to render the channel, and the OpenGL Modelview matrix has been set to the correct rotation to render the channel. The application programmer can access these values through the *OmniMapChannelBase* class interface. As each channel is rendered, the resulting image is placed into an OpenGL texture.

When all channels have been rendered, the application calls:

omniMapObj->PostRender()

This method will composite the channels and spherically project them onto the display.

Demo Program Case Study

This section describes what was done to create the project file for the Demo project located in:

<*InstallationDir*>*examples**demo**demo_dome_enabled**Source*

1. Open Visual Studio Workspace File

1.1 Open the Visual Studio Workspace for the dome enabled Demo program by double-clicking on “Demo.sln” in this directory.

2. Add OmniMap include Directory.

2.1 Using the properties dialog for the Demo project, we first added the include path for the OmniMap API include files to the property:

Configuration Properties

C/C++

General

Additional Include Directories

That path is :

<*InstallationDir*>*include*

In the project file, it is referenced relative to the Source directory. It is refererenced as ../../../../include.

3. Add OmniMap API Directory

3.1 Using the same properties dialog, we then added the library path for the OmniMap API library files to the property:

Configuration Properties

Linker

General

Additional Library Directories

That path is :

<*InstallationDir*>*lib*

In the project file, it is referenced relative to the Source directory. It is referenced as ../../../../lib.

4. Add OmniMap library dependency.

4.1 Finally, using the same properties dialog, we added the OmniMap library to the list of libraries to link with the Demo application. That property is:

Configuration Properties

Linker

Input

Additional Dependencies

That library is *OmniMap.lib* for the Release configuration, and *OmniMapD.lib* for the Debug configuration.

5. Edit "Basecode.cpp"

5.1 The changes to the code are shown in the *Demo Program Source Code* section below (on page 9), highlighted in red.

5.2 The changes include the following:

5.2.1 Add *Omnimap* header file

5.2.2 Add *glew* header file

5.2.3 Nest *render()* function - or *display()* function – inside the *fun()* function. The *fun()* funcion will be called once for each OmniMap render channel. The application programmer is required to call the following methods in the sequence shown here:

```
// Push matrices, set projection, model view matrix
// enable frame buffer object for this channel's rendering
chan->beginRenderToChannel();
// The application's rendering code
Render();
// Pop matrices, disable frame buffer object
chan->endRenderToChannel();
```

The rest of the code shown in the example is for demonstration purposes, but is not required.

5.2.4 Update *quit()* and *resize()* functions.

5.2.5 Update the render function (or display code) to insure that all rendering calls happen inside the render function AND that all scene updates happen outside the render function.

5.2.6 Set up the render channel.

5.2.7 Replace the display code in the main loop with the Omnimap display code.

5.2.8 Delete Fullscreen message box – if present

5.2.9 Create the Omnimap object

5.2.10 Compile and Run.

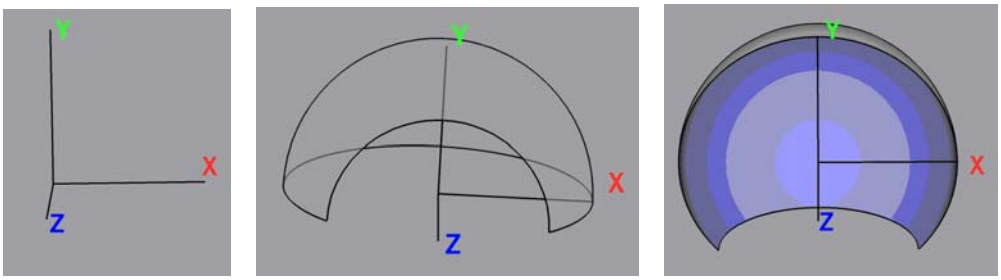
6. Set Omnimap Parameters

6.1 For details on setting OmniMap parameters, see the OmniMap Configuration Guide. OminMap parameters control where the projector is located, the direction of the projector, the orientation of the display and other configurable parameters. The parameters are set using Lua scripts. The default location of the configuration files is a directory called *OmniMapConfig*, located in the working directory of the program. The *OmniMap* library looks for a file called *OmniMapConfig/omnimap_startup.lua.* The location or name of that file can be changed using the 3rd argument to the *OmniMap* class constructor. There are two other LUA files referred to by *OmniMapConfig/omnimap_startup.lua* in the *OmniMapConfif* directory: *omnimap_user_edit.lua*, and *omnimap_dome_wiz_ai.lua*. If these files are not in the directory called *OmniMapConfig*, then the reference to them must be modified in *OmniMapConfig/omnimap_startup.lua*. The lines to be changed are:

```
Dofile("OmniMapConfig/omnimap_user_edit.lua")
Dofile("OmniMapConfig/omnimap_dome_wiz_ai.lua")
```

The parameters to the *dofile* call must be the correct paths to the two files.

6.2 Omnimap Coordinate System: Right Hand Rule where the X-vector is to the RIGHT, the Y-vector is UP and the Z-vector is AWAY from the user's gaze. The images below show the coord system in both a horizontal and vertical dome.



6.3 Parameters like dome style (horizontal, vertical), number of render channels and projector and audience position are set in the Lua (ASCII) file *omnimap_startup.lua* in the *Source/OmniMapConfig* directory.

6.4 The most common parameters however have been assembled into more the file *omnimap_user_edit.lua* in the same directory.

6.5 Run the program. Make sure that the directory containing the OmniMap DLLs is in your path. The directory is *<InstallationDir>\lib.*

Demo Program Source Code

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

// Add Omnimap header file
#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();

// Nest render() function - or display() function – inside the fun() function.
// The fun() funcion will be called once for each OmniMap render channel.
void fun(OmniMapChannelBase *chan)
{
	// Shows how to get a channel name from the channel pointer.
	std::string channelName = OmnimapLib->GetChannelName(chan);
	// Setup the modelview and projection matrices.
	chan->beginRenderToChannel();
	// Shows how to get metadata set in the lua scripts from a channel
	// In this case the metadata is called ExampleMetaData and is an integer.
	GenericDataContainer * cont=
		chan->ChannelMetaData.IndexDataMap("ExampleMetaData");
	int ExampleMetaData= 0; if(cont)  ExampleMetaData = cont->GetINT();
	// Shows how to get a number variable from the lua script
	double var2fromlua = OmnimapLib->ScriptingEngine->GetVariableNumber("nearclip");
	// Shows how to get a number variable from the lua script
	double var4fromlua = OmnimapLib->ScriptingEngine->GetVariableNumber("farclip");
	//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("Demo", hInstance))
	{
		MessageBox(NULL, "Failed to unregister class", "DoH!", MB_OK |
MB_ICONEXCLAMATION);
		hInstance = NULL;
	}
}


// ** SETUP PIXEL FORMAT **
// ------------------------
bool SetupPixelFormat()
{
	int PixelFormat;
	static PIXELFORMATDESCRIPTOR 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 = "Demo";                                // 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
                                                    "Demo",
                // 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 a separate 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
				OmniMapLib->ForEachChannel(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();
}
```