DXUT Tutorial Win32 Sample

<https://github.com/walbourn/directx-sdk-samples>

This is the DirectX SDK's Direct3D 11 tutorial updated to use Visual Studio 2012 and the Windows SDK 8.0 without any dependencies on legacy DirectX SDK content. This sample is a Win32 desktop DirectX 11.0 application for Windows 10, Windows 8.1, Windows 8, Windows 7, and Windows Vista Service Pack 2 with the DirectX 11.0 runtime.

**This is based on the legacy DirectX SDK (June 2010) Win32 desktop tutorial. This is not intended for use with Windows Store apps, Windows RT, or universal Windows apps.**

# Description

The original Direct3D 10 tutorial series covered Win32 basics in Tutorial01 - Tutorial07, and then DXUT in Tutorial08 - Tutorial10. This is Tutorial08 - Tutorial10 updated to use the [DXUT for Direct3D 11](http://go.microsoft.com/fwlink/?LinkId=320437) framework for Win32 desktop applications. These tutorials do not make use of Effects 11.

## Tutorial08



This tutorial introduces DXUT. DXUT is a layer that is built on top of Direct3D to help make samples, prototypes, tools, and professional games more robust and easier to build. It simplifies Windows and Direct3D APIs, based on typical usage.

The result of this tutorial is a sample that looks just like Tutorial07. However, it is implemented differently.

### DXUT

DXUT provides a simplified process for creating a window, creating (or selecting) a Direct3D device, and handling Windows messages. This allows you to spend less time worrying about how to perform these standard tasks.

DXUT for Direct3D 11 is also highly componentized. The core component of DXUT contains the standard window creation, device creation, and management functions. Optional components of DXUT include functions such as camera manipulation, a GUI system, and mesh handling. This tutorial explains the DXUT core component. Optional components are explained in later tutorials. Features introduced in this tutorial include device creation, the main loop, and simple keyboard input.

DXUT exposes a wide range of callback functions that the user can hook into. The callbacks are called by DXUT at logical points during program execution. You can insert custom code into the callbacks to build the application, while DXUT manages the window and device management implementation requirements.

#### Setting Callback Functions

Modifying the WinMain function of an application is the first step. In previous tutorials, WinMain invoked the initialization function, and then entered the message loop. In the DXUT framework, WinMain behaves similarly.

First, the callback functions are set by the application. These are the functions that DXUT calls during specific events when the application is run. Notable events include device creation, swap chain creation, keyboard entry, frame move, and frame rendering.

// Direct3D11 callbacks

DXUTSetCallbackD3D11DeviceAcceptable( IsD3D11DeviceAcceptable );

DXUTSetCallbackD3D11DeviceCreated( OnD3D11CreateDevice );

DXUTSetCallbackD3D11SwapChainResized( OnD3D11ResizedSwapChain );

DXUTSetCallbackD3D11SwapChainReleasing( OnD3D11ReleasingSwapChain );

DXUTSetCallbackD3D11DeviceDestroyed( OnD3D11DestroyDevice );

DXUTSetCallbackD3D11FrameRender( OnD3D11FrameRender );

DXUTSetCallbackMsgProc( MsgProc );

DXUTSetCallbackKeyboard( KeyboardProc );

DXUTSetCallbackFrameMove( OnFrameMove );

DXUTSetCallbackDeviceChanging( ModifyDeviceSettings );

DXUTSetCallbackDeviceRemoved( OnDeviceRemoved );

Then the application sets any additional DXUT settings, as in the following example:

// Show the cursor and clip it when in full screen

DXUTSetCursorSettings( true, true );

Finally, the initialization functions are called. The difference between this tutorial and the basic tutorials is that you have to deal only with application specific code during initialization. This is because the device and window creation is handled by DXUT. Application specific code can be inserted during each of the associated callback functions. DXUT invokes these functions as it progresses through the initialization process.

// Initialize DXUT and create the desired Win32 window and

// Direct3D device for the application. Calling each of these

// functions is optional, but they allow you to set options that

// control the behavior of the framework.

DXUTInit( true, true, nullptr );

// Parse the command line, handle the default hotkeys,

// and show msgboxes

DXUTCreateWindow( L"Tutorial8" );

DXUTCreateDevice( D3D\_FEATURE\_LEVEL\_10\_0, true, 800, 600 );

#### Debugging with DXUTTrace

DXUTTrace is a macro that can be called to create debug output for the application. It functions much like the standard debug output functions, but it allows a variable number of arguments. For example:

DXUTTRACE( L"Hit points left: %d\n", nHitPoints );

In this tutorial, it is placed next to function entry, to report the state of the application. For instance, at the beginning of OnD3D11ResizedSwapChain(), the debugger reports "SwapChain Reset called".

DXUTTRACE( L"SwapChain Reset called\n" );

### Device Management and Initialization

DXUT provides various methods for creating and configuring the window and the Direct3D device. The methods that are used in this tutorial are listed below. They are sufficient for a moderately complex Direct3D application.

The procedure from the previous tutorial inside InitDevice() and CleanupDevice() has been properly managed by splitting it into the following functions.

* IsD3D11DeviceAcceptable Callback
* ModifyDeviceSettings Callback
* OnD3D11CreateDevice Callback
* OnD3D11DestroyDevice Callback

Because not all resources are created at the same time, we can minimize overhead by reducing the number of repeated calls. We achieve this goal by recreating only resources that are context dependent. For the sake of simplicity, previous tutorials recreated everything whenever the screen was resized.

#### IsD3D11DeviceAcceptable Callback

This function is invoked for each combination of valid device settings that is found on the system. It allows the application to accept or reject each combination. For example, the application can reject all REF devices, or it can reject full screen device setting combinations.

In this tutorial, all devices are acceptable, because we do not need any advanced functionality.

#### ModifyDeviceSettings Callback

This callback function allows the application to change any device settings immediately before DXUT creates the device. In this tutorial, nothing needs to be done in the callback, because we do not need any advanced functionality.

#### OnD3D11CreateDevice Callback

This function is called after the Direct3D11 device is created. After the device is created, an application can use this callback to allocate resources, set buffers, and handle other necessary tasks.

In this tutorial, most of the InitDevice() function from tutorial 7 is copied into this callback function. The code for device and swap chain creation is omitted, because these functions are handled by DXUT. OnD3D11CreateDevice creates the effect, the vertex/index buffers, the textures, and the transformation matrices. The code has been copied from tutorial 7 with minimal alterations.

#### OnD3D11DestroyDevice Callback

This callback function is called immediately before the ID3D11Device is released by DXUT. This callback is used to release resources that were used by the device.

In this tutorial, OnD3D11DestroyDevice releases the resources that were created by the OnD3D11CreateDevice function. These resources include the vertex/index buffers, the layout, the textures, and the effect. The code is copied from the CleanupDevice() function, but it has been changed to use the SAFE\_RELEASE macro.

### Rendering

For rendering, DXUT provides two callback functions to initiate rendering for your application. The first, OnFrameMove, is called before each frame is rendered. It advances time in the application. The second, OnD3D11FrameRender, provides rendering for DXUT.

These two functions split the work of the Render() function into logical steps. OnFrameMove updates all the matrices for animation. OnD3D11FrameRender contains the rendering calls.

#### OnFrameMove Callback

This function is called before a frame is rendered. It is used to process the world state. However, its update frequency depends on the speed of the system. On faster systems, it is called more often per second. This means that any state update code must be regulated by time. Otherwise, it performs differently on a slower system than on a faster system.

Every time the function is called in the tutorial, the world is updated once, and the mesh color is adjusted. This code is copied directly from the Render() function in tutorial 7. Note that the rendering calls are not included.

#### OnD3D11FrameRender Callback

This function is called whenever a frame is redrawn. Within this function, effects are applied, resources are associated, and the drawing for the scene is called.

In this tutorial, this function contains the calls to clear the back buffer and stencil buffer, set up the matrices, and draw the cubes.

### Message Processing

Message processing is an intrinsic property of all window applications. DXUT exposes these messages for advanced applications.

#### MsgProc Callback

DXUT invokes this function when window messages are received. The function allows the application to handle messages as it sees fit.

In this tutorial, no additional messages are handled.

#### KeyboardProc Callback

This callback function is invoked by DXUT when a key is pressed. It can be used as a simple function to process keyboard commands.

The KeyboardProc callback is not used in this tutorial, because keyboard interaction is not necessary. However, there is a skeleton case for F1, which you can experiment with. Try to insert code to toggle the rotation of the cubes.

### Summary

The functions covered in this tutorial can get you started with a project on DXUT. Tutorials 9 and 10 cover more advanced DXUT features.

To start a new project using DXUT, you can use the Sample Browser to copy and rename any sample project in the DirectX SDK. The **EmptyProject11** sample is a good blank starting point for DXUT applications, or you can choose another sample.

## Tutorial09



This tutorial introduces the use of meshes to import source art. It demonstrates meshes by using DXUT. However, meshes can also be used without DXUT.

In the tutorial, you replace a cube in the center of the window with a model that is imported from a file. The model already contains a texture with mapped coordinates.

### Meshes

Listing each vertex within the source code is a tedious and crude method to define source art. Instead, there are methods to import already constructed models into your application.

A mesh is a collection of predefined vertex data which often includes additional information such as normal vectors, colors, materials, and texture coordinates. As long as the format of the mesh file is known, the mesh file can be imported and rendered.

Meshes are used to maintain source art separately from application code, so that the art can be reused. Therefore, meshes are stored in separate files. They are rarely generated in an application itself. Instead, they are created in 3D authoring software. The cube that was generated in previous tutorials can be saved into a separate file, and then re-loaded. However, importing can be much more efficient, especially if the mesh scales in complexity. One of the biggest benefits is the ability to import the texture coordinates, so that they match up perfectly. These can easily be specified in authoring applications.

DXUT handles meshes by using the CDXUTSDKMesh class. This class includes functions to import, render, and destroy a mesh. The file format we use for importing is the .SDKMESH format. See the [Samples Content Exporter](http://go.microsoft.com/fwlink/?LinkId=226208) for a utility that can generate these files from Autodesdk FBX files.

### Creating the Mesh

To import the model, we first create a CDXUTSDKMesh object. It is called g\_Mesh in this case, but in general the object name and the source art name should match, for easier association.

CDXUTSDKMesh g\_Mesh;

Next, we call the Create function to read the SDKMESH file and store it into the object. The Create function requires that we pass in the D3D11Device and the name of the file holding our mesh

In this case, we import a file that is called "tiny.sdkmesh". The format for the vertices that define this mesh contains vertex coordinates, normals, and texture coordinates. We must specify our input layout to match, as shown in the following example.

// Define the input layout

const D3D11\_INPUT\_ELEMENT\_DESC layout[] =

{

{ L"POSITION", 0, DXGI\_FORMAT\_R32G32B32\_FLOAT, 0, 0, D3D11\_INPUT\_PER\_VERTEX\_DATA, 0 },

{ L"NORMAL", 0, DXGI\_FORMAT\_R32G32B32\_FLOAT, 0, 12, D3D11\_INPUT\_PER\_VERTEX\_DATA, 0 },

{ L"TEXCOORD0", 0, DXGI\_FORMAT\_R32G32\_FLOAT, 0, 24, D3D11\_INPUT\_PER\_VERTEX\_DATA, 0 },

};

After the layout is defined, we call the Create function to import the model.

// Load the mesh

V\_RETURN( g\_Mesh.Create( pd3dDevice, L"tiny.sdkmesh" ) );

If there are no errors, g\_Mesh now contains the vertex and index buffer for our newly imported mesh, as well as textures and materials. Next we begin rendering.

### Rendering the Mesh

In the previous tutorials, because we had explicit control of the vertex buffer and the index buffers, we had to set them up properly before every frame. However, with a mesh, these elements are abstracted. Therefore, we only have to provide the effects technique that should be used to render the mesh, and it does all the work.

The difference from the previous tutorial is that we removed the portion of OnD3D11FrameRender where it began rendering the cube. We replaced it with the mesh rendering call. It is always good practice to set the correct input layout before any mesh is rendered. This ensures that the layout of the mesh matches the input assembler.

//

// Set the Vertex Layout

//    pd3dDevice->IASetInputLayout( g\_pVertexLayout );

The actual rendering is done by calling the Render function inside the CDXUTSDKMesh class. After the technique's buffers are correctly associated, it can be rendered.

//

// Render the mesh

//

UINT Strides[1];

UINT Offsets[1];

ID3D11Buffer\* pVB[1];

pVB[0] = g\_Mesh.GetVB11( 0, 0 );

Strides[0] = ( UINT )g\_Mesh.GetVertexStride( 0, 0 );

Offsets[0] = 0;

pd3dImmediateContext->IASetVertexBuffers( 0, 1, pVB, Strides, Offsets );

pd3dImmediateContext->IASetIndexBuffer( g\_Mesh.GetIB11( 0 ), g\_Mesh.GetIBFormat11( 0 ), 0 );

Because we are not using the Effects 11 library, the application is responsible for setting up the shaders and constants. After that is done, we can render the mesh one subset at a time.

for( UINT subset = 0; subset < g\_Mesh.GetNumSubsets( 0 ); ++subset )

{

auto pSubset = g\_Mesh.GetSubset( 0, subset );

auto PrimType = g\_Mesh.GetPrimitiveType11( ( SDKMESH\_PRIMITIVE\_TYPE )pSubset->PrimitiveType );

pd3dImmediateContext->IASetPrimitiveTopology( PrimType );

// Ignores most of the material information in them mesh to use

// only a simple shader

auto pDiffuseRV = g\_Mesh.GetMaterial( pSubset->MaterialID )->pDiffuseRV11;

pd3dImmediateContext->PSSetShaderResources( 0, 1, &pDiffuseRV );

pd3dImmediateContext->DrawIndexed( ( UINT )pSubset->IndexCount, 0, (UINT)pSubset->VertexStart );

}

If the world matrix is properly set for the model, we can now see that the cube in our previous example is replaced by this more complicated mesh. Notice that the mesh is much bigger than the cube. Try applying a scale to the world matrix to get the mesh to fit the screen better.

### Destroying the Mesh

Like all objects, the CDXUTSDKMesh object must be destroyed after usage. This is done by calling the Destroy function.

g\_Mesh.Destroy();

## Tutorial10



This tutorial covers the advanced DXUT concepts. Most of the functionality that is demonstrated in this tutorial is optional. However, it can be used to enhance your application with minimal cost. DXUT provides a simple sprite based GUI system and a device settings dialog. In addition, it provides a few types of camera classes.

In this tutorial, you create a fully functional GUI to modify settings by using the device and scene. There will be buttons, sliders, and text to demonstrate these capabilities.

### DXUT Camera

The CModelViewerCamera class within DXUT is provided to simplify the management of the view and projection transformations. It also provides GUI functionality.

CModelViewerCamera g\_Camera;  // A model viewing camera

The first function that the camera provides is the creation of the view and projection matrices. With the camera, there is no need to worry about these matrices. Instead, specify the viewer location, the view itself, and the size of the window. Then, pass these parameters to the camera object, which creates the matrices behind the scene.

The following example sets the view portion of the camera. This includes location and view.

// Initialize the camera

static const XMVECTORF32 s\_Eye = { 0.0f, 3.0f, -800.0f, 0.f };  
static const XMVECTORF32 s\_At = { 0.0f, 1.0f, 0.0f, 0.f };  
g\_Camera.SetViewParams( s\_Eye, s\_At );

Next, specify the projection portion of the camera. We need to provide the viewing angle, the aspect ratio, and the near and far clipping planes for the view frustum. This is the same information that was required in previous tutorials. However, in this tutorial we do not worry about creating the matrices themselves.

// Setup the camera's projection parameters

float fAspectRatio = pBackBufferSurfaceDesc->Width / (FLOAT)pBackBufferSurfaceDesc->Height;

g\_Camera.SetProjParams( XM\_PI / 4, fAspectRatio, 0.1f, 1000.0f );

g\_Camera.SetWindow( pBackBufferSurfaceDesc->Width, pBackBufferSurfaceDesc->Height );

The camera also creates masks for simple mouse feedback. Here, we specify three mouse buttons to utilize for the mouse operations that are provided—model rotation, zooming, and camera rotation. Try compiling the project and playing with each button to understand what each operation does.

g\_Camera.SetButtonMasks( MOUSE\_LEFT\_BUTTON, MOUSE\_WHEEL, MOUSE\_MIDDLE\_BUTTON );

After the buttons are set, the camera listens for mouse inputs and acts accordingly. To respond to user input, add a listener to the MsgProc callback function. This is the function that DXUT routes messages to.

// Pass all remaining windows messages to camera so it can respond

// to user input  
g\_Camera.HandleMessages( hWnd, uMsg, wParam, lParam );

Finally, after all the required data is entered into the camera, we extract the actual matrices for the transformations. We grab the projection matrix and the view matrix, together with the associated functions. The camera object is responsible for computing the matrices themselves.

XMMATRIX mView = g\_Camera.GetViewMatrix();  
XMMATRIX mProj = g\_Camera.GetProjMatrix();  
XMMATRIX mWorldViewProjection = g\_World \* mView \* mProj;

### DXUT Dialogs

User interaction can be accomplished by using the CDXUTDialog class. This contains controls in a dialog that accepts user input, and passes it to the application to handle. First, the dialog class is instantiated. Then, individual controls can be added.

#### Declarations

In this tutorial, two dialogs are added. One is called g\_HUD, and it contains the same code as the Direct3D 11 samples. The other is called g\_SampleUI, and it demonstrates functions that are specific to this tutorial. The second dialog is used to control the "puffiness" of the model. It sets a variable that is passed into the shaders.

    CDXUTDialog g\_HUD;      // manages the 3D UI  
    CDXUTDialog g\_SampleUI; // dialog for sample specific controls

The dialogs are controlled by a class called CDXUTDialogResourceManager. It passes messages and handles resources that are shared by the dialogs.

    CDXUTDialogResourceManager g\_DialogResourceManager; // manager for shared resources of dialogs

Finally, a new callback function is associated with the events that are processed by the GUI. This function is used to handle the interaction between the controls.

    void CALLBACK OnGUIEvent( UINT nEvent, int nControlID, CDXUTControl\* pControl, void\* pUserContext );

#### Dialog Initialization

Because more utilities have been introduced, and need to be initialized, this tutorial moves the initialization of these modules to a separate function, called InitApp().

The controls for each dialog are initialized inside this function. Each dialog calls its Init function, and passes in the resource manager to specify where the control should be placed. It also sets the callback function to process the GUI responses. In this case, the associated callback function is OnGUIEvent.

    g\_HUD.Init( &g\_DialogResourceManager );  
    g\_SampleUI.Init( &g\_DialogResourceManager );  
    g\_HUD.SetCallback( OnGUIEvent );  
    g\_SampleUI.SetCallback( OnGUIEvent );

After each dialog is initialized, it can insert the controls to use. The HUD adds three buttons for the basic functionality: toggle fullscreen, toggle reference (software) renderer, and change device.

To add a button, specify the IDC identifier to use, a string to display, the coordinates, the width and length, and optionally a keyboard shortcut to associate with the button.

Note that the coordinates are relative to the anchor of the dialog.

    int iY = 10;  
    g\_HUD.AddButton( IDC\_TOGGLEFULLSCREEN, L"Toggle full screen", 0, iY, 170, 22 );  
    g\_HUD.AddButton( IDC\_CHANGEDEVICE, L"Change device (F2)", 0, iY += 26, 170, 22, VK\_F2 );  
    g\_HUD.AddButton( IDC\_TOGGLEREF, L"Toggle REF (F3)", 0, iY += 26, 170, 22, VK\_F3 );  
    g\_HUD.AddButton( IDC\_TOGGLEWARP, L"Toggle WARP (F4)", 0, iY += 26, 170, 22, VK\_F4 );

Similarly for the sample UI, three controls are added—one static text, one slider, and one checkbox.

The static text parameters are the IDC identifier, the string, the coordinates, and the width and height.

The slider parameters are the IDC identifier, the coordinates, the width and height, then the min and max values of the slider, and finally the variable to store the result.

The checkbox parameters are the IDC identifier, a string label, the coordinates, the width and height, and the Boolean variable to store the result.

    iY = 10;  
    WCHAR sz[100];  
    iY += 24;  
    swprintf\_s( sz, 100, L"Puffiness: %0.2f", g\_fModelPuffiness );  
    g\_SampleUI.AddStatic( IDC\_PUFF\_STATIC, sz, 0 , iY += 26, 170, 22 );  
    g\_SampleUI.AddSlider( IDC\_PUFF\_SCALE, 50, iY += 26, 100, 22, 0, 2000, ( int )( g\_fModelPuffiness \* 100.0f ) );

    iY += 24;  
    g\_SampleUI.AddCheckBox( IDC\_TOGGLESPIN, L"Toggle Spinning", 0, iY += 26, 170, 22, g\_bSpinning );

After the dialogs are initialized, they must be placed on the screen. This is done during the OnD3D11ResizedSwapChain call, because the screen coordinates can change every time the swap chain is recreated—for example, if the window is resized.

    g\_HUD.SetLocation( pBackBufferSurfaceDesc->Width-170, 0 );  
    g\_HUD.SetSize( 170, 170 );  
    g\_SampleUI.SetLocation( pBackBufferSurfaceDesc->Width-170, pBackBufferSurfaceDesc->Height-300 );  
    g\_SampleUI.SetSize( 170, 300 );

Finally, the dialogs must be identified in the OnD3D11FrameRender function. This allows the dialogs to be drawn so that the user can actually see them.

    //  
    // Render the UI  
    //  
    g\_HUD.OnRender( fElapsedTime );   
    g\_SampleUI.OnRender( fElapsedTime );

### Resource Manager Initialization

The resource manager must be initialized at every callback that is associated with initialization and destruction. This is because the GUI must be recreated whenever a device is created, or a swap chain is recreated. The CDXUTDialogResourceManager class contains functions that correspond to each callback. Each function has the same name as its corresponding callback. All we need to do to is insert the code to call them in the appropriate places.

    V\_RETURN( g\_DialogResourceManager.OnD3D11CreateDevice( pd3dDevice, pd3dImmediateContext ) );  
    V\_RETURN( g\_DialogResourceManager.OnD3D11ResizedSwapChain( pd3dDevice, pBackBufferSurfaceDesc ) );  
    g\_DialogResourceManager.OnD3D11ReleasingSwapChain();  
    g\_DialogResourceManager.OnD3D11DestroyDevice();

#### Responding to GUI Events

After everything is initialized, we can write code to handle the GUI interaction. During initialization of the dialogs, we set the callback function to OnGUIEvent. Now we create the OnGUIEvent function, which listens for events that are related to the GUI and then processes them. (The GUI is invoked by the framework.)

OnGUIEvent is a simple function that contains a case statement for each IDC identifier that was listed when the dialogs were created. Each case statement contains the handler code, with the assumption that the user interacts with the control. The code here is very similar to the Win32 code that handles controls.

The controls that are related to the HUD call functions that are built into DXUT. There is a DXUT function to switch between fullscreen and windowed mode, to switch the reference software renderer on and off, and to change the device settings.

The SampleUI dialog contains custom code to manipulate the variables that are associated with the slider. It gathers the value, updates the associated text, and passes the value to the shader.

void CALLBACK OnGUIEvent( UINT nEvent, int nControlID,

CDXUTControl\* pControl, void\* pUserContext )  
{  
    switch( nControlID )  
    {  
        case IDC\_TOGGLEFULLSCREEN:  
            DXUTToggleFullScreen();  
            break;  
        case IDC\_TOGGLEREF:  
            DXUTToggleREF();  
            break;  
        case IDC\_TOGGLEWARP:  
            DXUTToggleWARP();  
            break;  
        case IDC\_CHANGEDEVICE:  
            g\_SettingsDlg.SetActive( !g\_SettingsDlg.IsActive() );  
            break;

        case IDC\_TOGGLESPIN:  
        {  
            g\_bSpinning =

g\_SampleUI.GetCheckBox(IDC\_TOGGLESPIN)->GetChecked();  
            break;  
        }

        case IDC\_PUFF\_SCALE:  
        {  
            WCHAR sz[100];  
            g\_fModelPuffiness = ( float )(

g\_SampleUI.GetSlider( IDC\_PUFF\_SCALE )->GetValue() \* 0.01f );  
            swprintf\_s( sz, 100, L"Puffiness: %0.2f", g\_fModelPuffiness );  
            g\_SampleUI.GetStatic( IDC\_PUFF\_STATIC )->SetText( sz );  
            break;  
        }

    }  
}

#### Updating Message Processing

Now we have dialog messages and user interactions. Messages that are passed to the application must be handled by the dialogs. The relevant code is handled within the MsgProc callback that is provided by DXUT. In previous tutorials, this section is empty, because there are no messages to be processed. Now, we must make sure that messages intended for the resource manager and dialogs are properly routed.

No special message processing code is required. We just call the MsgProcs for each dialog, to ensure that the message is handled. This is done by calling the MsgProc function that corresponds to each class. Note that the function provides a flag to notify the framework that no further processing of the message is required, and therefore the framework can exit.

    LRESULT CALLBACK MsgProc( HWND hWnd, UINT uMsg, WPARAM wParam,

LPARAM lParam, bool\* pbNoFurtherProcessing, void\* pUserContext )  
    {  
        // Always allow dialog resource manager calls to handle

// global messages so GUI state is updated correctly  
        \*pbNoFurtherProcessing = g\_DialogResourceManager.MsgProc( hWnd, uMsg,

wParam, lParam );  
        if( \*pbNoFurtherProcessing )  
            return 0;  
      
        if( g\_D3DSettingsDlg.IsActive() )  
        {  
            g\_D3DSettingsDlg.MsgProc( hWnd, uMsg, wParam, lParam );  
            return 0;  
        }  
      
        // Give the dialogs a chance to handle the message first  
        \*pbNoFurtherProcessing = g\_HUD.MsgProc( hWnd, uMsg, wParam, lParam );  
        if( \*pbNoFurtherProcessing )  
            return 0;  
        \*pbNoFurtherProcessing = g\_SampleUI.MsgProc( hWnd, uMsg,

wParam, lParam );  
        if( \*pbNoFurtherProcessing )  
            return 0;  
      
        if( uMsg == WM\_CHAR && wParam == '1' )  
            DXUTToggleFullScreen();  
      
        return 0;  
    }

### 3D Settings Dialog

There is a special built-in dialog that controls the settings of the Direct3D device. This dialog is provided by DXUT as CD3DSettingsDlg. It functions like a custom dialog, but it provides all the options that users need to modify settings.

    CD3DSettingsDlg         g\_SettingsDlg;       // Device settings dialog

Initialization is much like other dialogs. Simply call the Init function. However, every time Direct3D changes its swap chain or device, the dialog must also be updated. Therefore, it must include an appropriately named call within OnD3D11CreateDevice and OnD3D11ResizedSwapChain. Likewise, changes to destroyed objects must also be notified. Therefore, we need the appropriate calls within OnD3D11DestroyDevice.

    g\_SettingsDlg.Init( &g\_DialogResourceManager );  
      
    V\_RETURN( g\_SettingsDlg.OnD3D11CreateDevice( pd3dDevice ) );  
    V\_RETURN( g\_SettingsDlg.OnD3D11ResizedSwapChain( pd3dDevice, pBackBufferSurfaceDesc ) );  
      
    g\_SettingsDlg.OnD3D11DestroyDevice();

On the rendering side, to switch the appearance of the dialog, we use a flag called IsActive(). If this flag is set to false, then the panel is not rendered. Switching the panel is handled by the HUD dialog. The IDC\_CHANGEDEVICE that is associated with the HUD controls this flag.

    if( g\_SettingsDlg.IsActive() )  
    {  
        g\_SettingsDlg.MsgProc( hWnd, uMsg, wParam, lParam );  
        return 0;  
    }

After the initialization steps are complete, you can include the dialog in your application. Try compiling the tutorial and interacting with the Change Settings panel to see its effect. The reconstruction of the Direct3D device or swap chain is done internally by DXUT.

### Text Rendering

An application is not very interesting if the user has no idea what to do. DXUT includes a utility class to draw 2D text onto the screen, for feedback to the user. This class, CDXUTTextHelper, allows you to draw lines of text anywhere on the screen, by using simple string inputs. First, we instantiate the class. Because text rendering can be isolated from most of the initialization procedures, we keep most of the code within RenderText.

CDXUTTextHelper\* g\_pTxtHelper = nullptr;

g\_pTxtHelper = new CDXUTTextHelper( pd3dDevice, pd3dImmediateContext, &g\_DialogResourceManager, 15 );

#### Rendering

The text in this sample includes statistics on the rendering. There is also a help section that explains how to manipulate the model by using the mouse.

The rendering calls must be done within OnD3D11FrameRender. Here, we call RenderText within the frame render call.

The first section is always rendered first. The first call to text rendering is Begin(). This notifies the engine to start sending text to the screen. Next, we set the position of the cursor and the color of the text. Now we can draw.

Text string output is performed by calling DrawTextLine. Pass in the string, and output that corresponds to the string is provided at the current position. The cursor is incremented while text is written. For example, if the string contains "\n", the cursor is automatically moved to the next line.

    g\_pTxtHelper->Begin();  
    g\_pTxtHelper->SetInsertionPos( 5, 5 );  
    g\_pTxtHelper->SetForegroundColor( Colors::Yellow );  
    g\_pTxtHelper->DrawTextLine( DXUTGetFrameStats( DXUTIsVsyncEnabled() ) );  
    g\_pTxtHelper->DrawTextLine( DXUTGetDeviceStats() );

Because the help text is drawn in the same way, we do not need to review its code.

You can reposition the pointer at any time by calling SetInsertionPos.

When you are satisfied with the text output, call End() to notify the engine.

    g\_pTxtHelper->End();

# Dependencies

DXUT-based samples typically make use of runtime HLSL compilation. Build-time compilation is recommended for all production Direct3D applications, but for experimentation and samples development runtime HLSL compilation is preferred. Therefore, the D3DCompile\*.DLL must be available in the search path when these programs are executed.

* When using the Windows 8.x SDK and targeting Windows Vista or later, you can include the D3DCompile\_46 or D3DCompile\_47 DLL side-by-side with your application copying the file from the REDIST folder.

%ProgramFiles(x86)%\Windows kits\8.0\Redist\D3D\arm, x86 or x64

%ProgramFiles(x86)%\Windows kits\8.1\Redist\D3D\arm, x86 or x64

%ProgramFiles(x86)%\Windows kits\10\Redist\D3D\arm, x86 or x64

# More Information

[Where is the DirectX SDK (2015 Edition)?](https://walbourn.github.io/where-is-the-directx-sdk-2015-edition/)

[DXUT for Win32 Desktop Update](https://walbourn.github.io/dxut-for-win32-desktop-update/)

[Games for Windows and DirectX SDK blog](https://walbourn.github.io/)