Tutorial 10: Advanced DXUT

Collapse All

Summary

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.

Source

(SDK root)\Samples\C++\Direct3D10\Tutorials\Tutorial10

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
D3DXVECTOR3 Eye( 0.0f, 0.0f, -800.0f );
D3DXVECTOR3 At( 0.0f, 0.0f, 0.0f );
g_Camera.SetViewParams( &Eye, &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( D3DX_PI/4, fAspectRatio, 0.1f, 5000.0f );
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.

```
g_pProjectionVariable->SetMatrix( (float*)g_Camera.GetProjMatrix() );
g_pViewVariable->SetMatrix( (float*)g_Camera.GetViewMatrix() );
```

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 10 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", 35, iY, 125, 22 );
g_HUD.AddButton( IDC_TOGGLEREF, L"Toggle REF (F3)", 35, iY += 24, 125, 22 );
g_HUD.AddButton( IDC_CHANGEDEVICE, L"Change device (F2)", 35, iY += 24, 125, 22, VK_F2 );
```

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;
StringCchPrintf( sz, 100, L"Puffiness: %0.2f", g_fModelPuffiness );
g_SampleUI.AddStatic( IDC_PUFF_STATIC, sz, 35, iY += 24, 125, 22 );
g_SampleUI.AddSlider( IDC_PUFF_SCALE, 50, iY += 24, 100, 22, 0, 2000, (int)(g_fModelPuffiness*100.0f) );

iY += 24;
g_SampleUI.AddCheckBox( IDC_TOGGLESPIN, L"Toggle Spinning", 35, iY += 24, 125, 22, g_bSpinning );
```

After the dialogs are initialized, they must be placed on the screen. This is done during the OnD3D10ResizedSwapChain 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 OnD3D10FrameRender 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.OnD3D10CreateDevice( pd3dDevice ) );
V_RETURN( g_DialogResourceManager.OnD3D10ResizedSwapChain( pd3dDevice, pBackBufferSurfaceDesc ) );
g_DialogResourceManager.OnD3D10ReleasingSwapChain();
g_DialogResourceManager.OnD3D10DestroyDevice();
```

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_CHANGEDEVICE:     g_D3DSettingsDlg.SetActive( !g_D3DSettingsDlg.IsActive() ); break;

        case IDC_TOGGLESPIN:
        {
            g_bSpinning = g_SampleUI.GetCheckBox( IDC_TOGGLESPIN )->GetChecked();
            break;
        }

        case IDC_PUFF_SCALE:
        {
            g_fModelPuffiness = (float) (g_SampleUI.GetSlider( IDC_PUFF_SCALE )->GetValue() * 0.01f);

            WCHAR sz[100];
            StringCchPrintf( sz, 100, L"Puffiness: %0.2f", g_fModelPuffiness );
            g_SampleUI.GetStatic( IDC_PUFF_STATIC )->SetText( sz );

            g_pPuffiness->SetFloat( g_fModelPuffiness );
            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_D3DSettingsDlg;       // 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 OnD3D10CreateDevice and OnD3D10ResizedSwapChain. Likewise, changes to destroyed objects must also be notified. Therefore, we need the appropriate calls within OnD3D10DestroyDevice.

```
g_D3DSettingsDlg.Init( &g_DialogResourceManager );

V_RETURN( g_D3DSettingsDlg.OnD3D10CreateDevice( pd3dDevice ) );
V_RETURN( g_D3DSettingsDlg.OnD3D10ResizedSwapChain( pd3dDevice, pBackBufferSurfaceDesc ) );

g_D3DSettingsDlg.OnD3D10DestroyDevice();
```

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_D3DSettingsDlg.IsActive() )
{
    g_D3DSettingsDlg.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, CDXUTD3D10TextHelper, 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 RenderText10.

```
CDXUTD3D10TextHelper txtHelper( g_pFont, g_pSprite, 15 );
```

Initialization

The first parameter passed in is the font to draw. This font is of the type ID3DXFont, which is provided by D3DX. To initialize the font, call D3DX10CreateFont, and pass in the device, height, width, weight, mipmap levels (generally 1), italics, character set, precision, quality, pitch and family, font face name, and the pointer to the object. Only the first four and the last two of these are really significant.

```
V_RETURN( D3DX10CreateFont( pd3dDevice, 15, 0, FW_BOLD, 1, FALSE, DEFAULT_CHARSET,
                            OUT_DEFAULT_PRECIS, DEFAULT_QUALITY, DEFAULT_PITCH | FF_DONTCARE,
                            L"Arial", &g_pFont ) );
```

The second parameter requires that we initialize an ID3DXSprite class. To do this, we call D3DX10CreateSprite. The only things the function requires as parameters are the device, the maximum number of sprites ever drawn in one frame, and the pointer to the object.

```
// Initialize the sprite
V_RETURN( D3DX10CreateSprite( pd3dDevice, MAX_SPRITES, &g_pSprite ) );
```

As with all other objects, the font and sprite must be destroyed after we have finished with them. This can be accomplished by using the SAFE_RELEASE macro.

```
SAFE_RELEASE( g_pFont );
SAFE_RELEASE( g_pSprite );
```

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 OnD3D10FrameRender. Here, we call RenderText10 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.

```
// Output statistics
txtHelper.Begin();
txtHelper.SetInsertionPos( 2, 0 );
```

```
    txtHelper.SetForegroundColor( D3DXCOLOR( 1.0f, 1.0f, 0.0f, 1.0f ) );
    txtHelper.DrawTextLine( DXUTGetFrameStats() );
    txtHelper.DrawTextLine( DXUTGetDeviceStats() );
```

There is another method to provide text output, which is similar to printf. Format the string by using special characters, and then insert variables into the string. Use DrawFormattedTextLine for this purpose.

```
    txtHelper.SetForegroundColor( D3DXCOLOR( 1.0f, 1.0f, 1.0f, 1.0f ) );
    txtHelper.DrawFormattedTextLine( L"fTime: %0.1f  sin(fTime): %0.4f", fTime, sin(fTime) );
```

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.