P3_Tutorial4 pt.2

Part 1 looked at using surfaces to animate; this tutorial aims to take you through the basics of Direct3D animation using the Sprite interface.

In this tutorial we are going to use the sprite interfaces to animate a spinning wheel, just like we did using surfaces.
We are also going to learn a more efficient way of coding than constantly creating a backbuffer – We use methods called from the `IDirect3DDevice9` they are called BeginScene() and EndScene().

To use sprites we need to load them up as textures which requires code similar to that used in P3_Tutorial3pt.3

We are using DirectX 9.0 without the summer update. One of the main things to change with the summer update was the ID3DXSprite Interface and its methods such as Draw(). These tutorials are therefore not compatible with the summer update.

Using Sprites

We need to specify our header files, global variables and function prototypes.

```
//////////////////////////////////////////////////
// Sprite Animation App
//////////////////////////////////////////////////
#include <windows.h>
#include <d3d9.h>
#include <d3dx9.h>
#include <dxerr9.h> // included for errors

// Function prototypes.
LRESULT WINAPI WndProc(HWND hWnd, UINT msg,
    WPARAM wParam, LPARAM lParam);
void RegisterWindowClass(HINSTANCE hInstance);
void CreateAppWindow(HINSTANCE hInstance);
WPARAM StartMessageLoop();
HRESULT InitFullScreenDirect3D();
void Render();
void CleanUpDirect3D();
BOOL IsTimeForNextFrame();

// Global variables.
HWND g_hWnd;
IDirect3D9* g_pDirect3D = NULL;
IDirect3DDevice9* g_pDirect3DDevice = NULL;
ID3DXSprite* g_pBitmapSprites = NULL;
// Declare Array of 8 textures to hold bitmaps
IDirect3DTexture9* g_pTexture[8];
// Declare Array of 8 surfaces to hold bitmaps
//IDirect3DSurface9* g_pBitmapSurfaces[8];

HRESULT g_hResult = D3D_OK;
char g_szErrorMsg[256];

// ** globals for sprites - passed as params in the Draw() method
```

```
// ** they determine such things as how, where the sprite is drawn
// global vectors - vectors are necessary for sprite class
D3DXVECTOR2                    g_vCentre(64.0f, 64.0f);
// Sprite translation - set in middle of screen
D3DXVECTOR2                    g_vTranslation(100.0f, 50.0f);
// normally we would need a vector for scaling but by setting
// the relevant Draw() method param to NULL scaling is (1.0,1.0)
// i.e. no scaling is used and the source image size is preserved
// D3DXVECTOR2                         g_vScaling;
// global float of the angle of rotation in radians
float                          g_fRotation = 0;
```

Next is our WinMain() function, where everything originates from. This has been stripped down with code we normally see added to functions that are called from WinMain().
WinMain() is therefore only acting to initialise some of our variables and to call other functions.

```
//////////////////////////////////////////////////////
// WinMain()
//////////////////////////////////////////////////////
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE, LPSTR, INT)
{
      // initialise our array of textures to NULL
      for (int x=0; x<8; ++x)
            g_pTexture[x] = NULL;

    RegisterWindowClass(hInstance);
    CreateAppWindow(hInstance);
    ShowWindow(g_hWnd, SW_SHOWDEFAULT);
    UpdateWindow(g_hWnd);
    HRESULT hResult = InitFullScreenDirect3D();

    if (SUCCEEDED(hResult))
        WPARAM result = StartMessageLoop();

    CleanUpDirect3D();
    CloseWindow(g_hWnd);

    if (g_hResult != D3D_OK)
        DXTRACE_ERR(g_szErrorMsg, g_hResult);

    return 0;
}
```

Next we add a function to handle any windows messages, as this is platform we are operating one it is required - `WndProc()`

The next code to add is the code to register our window, this function is called from WinMain() and registers our window with the Windows operating system - `RegisterWindowClass()`

This function is again called by WinMain() and this creates our window - `CreateAppWindow()`

The next code to add is the message handler - `StartMessageLoop()`

These functions have not changed; therefore the code is not included!

The next function coded is `InitDirect3D` to initialise the elements of our program. It checks for errors and returns relevant error code.

```
//////////////////////////////////////////////////////
// InitFullScreenDirect3D()
//////////////////////////////////////////////////////
HRESULT InitFullScreenDirect3D()
{
    g_pDirect3D = Direct3DCreate9(D3D_SDK_VERSION);
    if (g_pDirect3D == NULL)
    {
        MessageBox(g_hWnd,
            "Couldn't create DirectX object.",
            "DirectX Error", MB_OK);
        return E_FAIL;
    }

    HRESULT hResult = g_pDirect3D-
>CheckDeviceType(D3DADAPTER_DEFAULT,
    D3DDEVTYPE_HAL, D3DFMT_X8R8G8B8, D3DFMT_X8R8G8B8, FALSE);

    if (hResult != D3D_OK)
    {
        MessageBox(g_hWnd,
            "Sorry. This program won't\nrun on your system.",
            "DirectX Error", MB_OK);
        return E_FAIL;
    }

    D3DPRESENT_PARAMETERS D3DPresentParams;
    ZeroMemory(&D3DPresentParams, sizeof(D3DPRESENT_PARAMETERS));
    D3DPresentParams.Windowed = FALSE;
    D3DPresentParams.BackBufferCount = 1;
    D3DPresentParams.BackBufferWidth = 800;
    D3DPresentParams.BackBufferHeight = 600;
    D3DPresentParams.BackBufferFormat = D3DFMT_X8R8G8B8;
    D3DPresentParams.SwapEffect = D3DSWAPEFFECT_DISCARD;
    D3DPresentParams.hDeviceWindow = g_hWnd;

    hResult = g_pDirect3D->CreateDevice(D3DADAPTER_DEFAULT,
    D3DDEVTYPE_HAL, g_hWnd, D3DCREATE_SOFTWARE_VERTEXPROCESSING,
        &D3DPresentParams, &g_pDirect3DDevice);

    if (FAILED(hResult))
    {
        MessageBox(g_hWnd,
            "Failed to create Direct3D device.",
            "DirectX Error", MB_OK);
        return E_FAIL;
    }

            // Get a pointer to the Sprite interface
      hResult = D3DXCreateSprite(g_pDirect3DDevice,
&g_pBitmapSprites);
```

```
      if ( FAILED(hResult) )
      {
            MessageBox(g_hWnd, "Cannot create sprite interface",
"DirectX Error", MB_OK);
            return E_FAIL;
      }

      /* This loop loads the surfaces with images, which are the
      8 animation frames. */
    for (int x=0; x<8; ++x)
    { // this section of code loads the surfaces with a bitmaps frame
        char filename[15];    // where we are going to store our
filename
            // dyanamically create our filename
        wsprintf(filename, "%s%d%s", "Frame0", x+1, ".bmp");

            g_hResult =
D3DXCreateTextureFromFileEx(g_pDirect3DDevice, filename,
                  D3DX_DEFAULT, D3DX_DEFAULT, 1, 0, D3DFMT_A8R8G8B8,
                  D3DPOOL_MANAGED, D3DX_DEFAULT, D3DX_DEFAULT,
0xFF000000,
                  NULL, NULL, &g_pTexture[x]);

            if (FAILED(g_hResult))
        {
            strcpy(g_szErrorMsg, "Couldn't load bitmap file.");
                  PostQuitMessage(WM_QUIT);
            break;
        }
    }

    g_pDirect3DDevice->Clear(0, NULL, D3DCLEAR_TARGET,
        D3DCOLOR_XRGB(0,0,0), 1.0f, 0);

    return D3D_OK;
}
```

This function cleans up any pointers and memory used by DirectX and is called before the program closes

```
//////////////////////////////////////////////////////
// CleanUpDirect3D()
//////////////////////////////////////////////////////
void CleanUpDirect3D()
{
    for (int x=0; x<8; ++x)
    {
        if (g_pTexture[8])
            g_pTexture[8]->Release();
    }
      if (g_pBitmapSprites)
        g_pBitmapSprites->Release();
    if (g_pDirect3DDevice)
        g_pDirect3DDevice->Release();
    if (g_pDirect3D)
        g_pDirect3D->Release();
}
```

Render is the other function that is to under go a large change. As render is performing the drawing it now needs to load the textures into the sprites, present them to the scene and then display.

```
//////////////////////////////////////////////////
// Render()
//////////////////////////////////////////////////
void Render()
{     // call the timer function - when it returns
      // true it is time to display our next frame
    if (IsTimeForNextFrame())
    { // update our frame counter
        static frameNum = 0;
        frameNum = frameNum + 1;
        if (frameNum == 8)
            frameNum = 0;

            g_pDirect3DDevice->Clear(0, NULL, D3DCLEAR_TARGET,
            D3DCOLOR_XRGB(0,0,0), 1.0f, 0);

// Begin rendering the scene - more efficient than using
// backbuffer surfaces, has the same effect with the added bonus
// that drawing done after BeginScene and before EndScene
// is draw to that seen - present then presents the new scene
            if( SUCCEEDED( g_pDirect3DDevice->BeginScene() ) )
            {
            // Draw function has changed radically with summer update
            // only takes 5 params now - see msdn for definition
                  g_hResult = g_pBitmapSprites ->
            Draw(g_pTexture[frameNum], NULL, NULL, &g_vCentre,
            g_fRotation, &g_vTranslation, 0xFFFFFFFF);

                  if (FAILED(g_hResult))
                  {
                        strcpy(g_szErrorMsg, "Error copying image
                  buffer.");
                        PostQuitMessage(WM_QUIT);
                        return;
                  }
              // End the scene
                  g_pDirect3DDevice->EndScene();
            }

      // present the new scene to the screen
        g_pDirect3DDevice->Present(NULL, NULL, NULL, NULL);
    }
}
```

The last function is a very basic timer that just uses a for loop, it is called from the render function so it knows when to update and display the next frame:

```
//////////////////////////////////////////////////
// IsTimeForNextFrame()
//////////////////////////////////////////////////
BOOL IsTimeForNextFrame()
{
      // a very basic timer that uses a for loop
      // professional timers will be covered later
    static DWORD count = 0;
```

```
    count = count + 1;
    if (count > 100000)
    {
        count = 0;
        return TRUE;
    }
    else
      {
        return FALSE;
      }
}
```

Exercises

1) Modify the code so that the animation runs at half speed
2) Modify the program so that it displays only a 200x200 image from the centre of each frame
3) Move the display so that the animation runs in the lower left corner of the screen

Work to do

Make a sprite class that contains all the necessary functionality to create and control multiple sprites via instances of the class.
The class should be declared in a header and methods then defined in a source file.