DynamicShaderLinkage11

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

This is the DirectX SDK's Direct3D 11 sample 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 sample. This is not intended for use with Windows Store apps, Windows RT, or universal Windows apps.**

# Description

This Direct3D 11 sample demonstrates use of Shader Model 5 shader interfaces and Direct3D 11 support for linking shader interface methods at runtime.

This sample uses the [*DXUT for Direct3D 11*](http://go.microsoft.com/fwlink/?LinkId=320437) framework for Win32 desktop applications. There is an Effects 11 version of this sample as well. This sample requires Feature Level 11.0 or better to show the DLS feature, but has a static lighting fallback for Feature Level 10.0 or better.

## https://code.msdn.microsoft.com/site/view/file/96133/1/DynamicShaderLinkage11.JPG

## Coping With The Shader Permutation Explosion

Rendering systems need to deal with a wide range of complexity when managing shaders while providing the greatest possible opportunity to optimize shader code. This becomes an even greater challenge when you factor in the need to support a variety of different materials in a rendered scene across the broad range of available hardware configurations. To address these challenges, shader developers have often resorted to one of two general approaches. Either creating fully featured "uber-shaders" which trade off some performance for flexibility or specifically creating individual shaders for each geometry stream, material type or light type combination needed.

Uber-shaders handle this combinatorial problem by recompiling the same shader with different preprocessor defines, while the latter method uses brute-force developer power to the same end. This shader permutation explosion has often been a problem for developers who must now manage thousands of different shader permutations within their game and asset pipeline.

Direct3D 11 and shader model 5 introduces Object Oriented language constructs and provides runtime support of shader linking to assist developers in tackling these development problems. This sample uses shader interfaces to help manage the number of shader permutations created.

## Shader Model 5 Interfaces and Classes

In the pixel shader, we first define base interfaces for different light and material types:

   //--------------------------------------------------------------------------------------  
   // Interfaces  
   //--------------------------------------------------------------------------------------  
   interface iBaseLight  
   {  
   float3 IlluminateAmbient(float3 vNormal);

   float3 IlluminateDiffuse(float3 vNormal);

   float3 IlluminateSpecular(float3 vNormal, int specularPower );

   };  
     
   // ...  
   interface iBaseMaterial  
   {  
   float3 GetAmbientColor(float2 vTexcoord);

   float3 GetDiffuseColor(float2 vTexcoord);

   int GetSpecularPower();

   };

   // ...

Next we build specialized classes based on these interfaces, adding to the functionality where needed:

//--------------------------------------------------------------------------------------

// Classes

//--------------------------------------------------------------------------------------

class cAmbientLight : iBaseLight

{

float3 m\_vLightColor;

bool     m\_bEnable;

   float3 IlluminateAmbient(float3 vNormal);

   float3 IlluminateDiffuse(float3 vNormal);

   float3 IlluminateSpecular(float3 vNormal, int specularPower );

   };

   // ...

   class cBaseMaterial : iBaseMaterial

   {

   float3 m\_vColor;

   int      m\_iSpecPower;

   float3 GetAmbientColor(float2 vTexcoord);

   float3 GetDiffuseColor(float2 vTexcoord);

   int GetSpecularPower();

   };

   // ....

## Abstract Instances and The Shaders Main Function

The pixel shaders main function uses abstract instances of the interfaces for computation. These interfaces instances are made concrete by the application code at shader bind time:

//--------------------------------------------------------------------------------------  
// Abstract Interface Instances for dyamic linkage / permutation  
//--------------------------------------------------------------------------------------  
iBaseLight     g\_abstractAmbientLighting;  
iBaseLight     g\_abstractDirectLighting;  
iBaseMaterial  g\_abstractMaterial;

//--------------------------------------------------------------------------------------  
// Pixel Shader  
//--------------------------------------------------------------------------------------  
float4 PSMain( PS\_INPUT Input ) : SV\_TARGET  
{  
 // Compute the Ambient term  
 float3   Ambient = (float3)0.0f;

Ambient = g\_abstractMaterial.GetAmbientColor( Input.vTexcoord ) \* g\_abstractAmbientLighting.IlluminateAmbient( Input.vNormal );

// Accumulate the Diffuse contribution  
 float3   Diffuse = (float3)0.0f;

Diffuse += g\_abstractMaterial.GetDiffuseColor( Input.vTexcoord ) \* g\_abstractDirectLighting.IlluminateDiffuse( Input.vNormal );

// Compute the Specular contribution  
 float3   Specular = (float3)0.0f;  
 Specular += g\_abstractDirectLighting.IlluminateSpecular( Input.vNormal, g\_abstractMaterial.GetSpecularPower() );

// Accumulate the lighting with saturation  
 float3 Lighting = saturate( Ambient + Diffuse + Specular );

return float4(Lighting,1.0f);

}

Here we are simply accumulating the lighting for the pixel across the generic / abstract light instances.

## Application Code and Shader Interfaces

Once the shader has been compiled and loaded, the sample application code makes use of the shader reflection layer to acquire both the concrete and abstract shader class instance offsets within the compiled shader. These offsets are then used to dynamically configure the shader code path according to user selected options. The linkage happens directly in the Direct3D 11 runtime and is specified during the PSSetShader call

### Acquisition of the Shader Interfaces

After shader compilation we must acquire the offsets for the abstract instances of our permutable shader objects:

// use shader reflection to get data locations for the interface array  
ID3D11ShaderReflection\* pReflector = NULL;  
V\_RETURN( D3DReflect( pPixelShaderBuffer->GetBufferPointer(), pPixelShaderBuffer->GetBufferSize(),  
IID\_ID3D11ShaderReflection, (void\*\*) &pReflector) );

g\_iNumPSInterfaces = pReflector->GetNumInterfaceSlots();  
g\_dynamicLinkageArray = (ID3D11ClassInstance\*\*) malloc( sizeof(ID3D11ClassInstance\*) \* g\_iNumPSInterfaces );  
if (g\_dynamicLinkageArray == NULL)  
return E\_FAIL;

ID3D11ShaderReflectionVariable\* pAmbientLightingVar = pReflector->GetVariableByName("g\_abstractAmbientLighting");  
g\_iAmbientLightingOffset = pAmbientLightingVar->GetInterfaceSlot(0);

ID3D11ShaderReflectionVariable\* pDirectLightingVar = pReflector->GetVariableByName("g\_abstractDirectLighting");  
g\_iDirectLightingOffset = pDirectLightingVar->GetInterfaceSlot(0);

ID3D11ShaderReflectionVariable\* pMaterialVar = pReflector->GetVariableByName("g\_abstractMaterial");  
g\_iMaterialOffset = pMaterialVar->GetInterfaceSlot(0);  
             
// ...

Next we enumerate all possible permutations of material object that exist in the shader:

// Material Dynamic Permutation  
enum E\_MATERIAL\_TYPES  
{  
   MATERIAL\_PLASTIC,  
   MATERIAL\_PLASTIC\_TEXTURED,  
   MATERIAL\_PLASTIC\_LIGHTING\_ONLY,

   MATERIAL\_ROUGH,  
   MATERIAL\_ROUGH\_TEXTURED,  
   MATERIAL\_ROUGH\_LIGHTING\_ONLY,

   MATERIAL\_TYPE\_COUNT  
};  
char\*  g\_pMaterialClassNames[ MATERIAL\_TYPE\_COUNT ] =  
{  
   "g\_plasticMaterial",             // cPlasticMaterial  
   "g\_plasticTexturedMaterial",     // cPlasticTexturedMaterial  
   "g\_plasticLightingOnlyMaterial", // cPlasticLightingOnlyMaterial  
   "g\_roughMaterial",               // cRoughMaterial  
   "g\_roughTexturedMaterial",       // cRoughTexturedMaterial  
   "g\_roughLightingOnlyMaterial"    // cRoughLightingOnlyMaterial  
};  
E\_MATERIAL\_TYPES            g\_iMaterial = MATERIAL\_PLASTIC\_TEXTURED;

// ...

// Acquire the material Class Instances for all possible material settings  
for( UINT i=0; i < MATERIAL\_TYPE\_COUNT; i++)  
{  
   g\_pPSClassLinkage->GetClassInstance( g\_pMaterialClassNames[i], 0, &g\_pMaterialClasses[i] );  
}

### Specifying Linkage to the Runtime

Users specify the specific lighting and material configuration through the samples user interface. In the application code, at Shader bind time the following selectively sets shader code bindings according to user specified settings:

// Setup the Shader Linkage based on the user settings for Lighting  
    // Ambient Lighting First - Constant or Hemi?  
    if ( g\_bHemiAmbientLighting )  
    g\_dynamicLinkageArray[g\_iAmbientLightingOffset] = g\_pHemiAmbientLightClass;  
    else  
    g\_dynamicLinkageArray[g\_iAmbientLightingOffset] = g\_pAmbientLightClass;

    // Direct Light - None or Directional TODO: or Spot or Environment?  
    if (g\_bDirectLighting)  
    g\_dynamicLinkageArray[g\_iDirectLightingOffset] = g\_pDirectionalLightClass;  
    else  
    g\_dynamicLinkageArray[g\_iDirectLightingOffset] = g\_pAmbientLightClass;

    // Use the selected material class instance  
    if (g\_bLightingOnly)  
    {  
    switch( g\_iMaterial )  
    {  
    case MATERIAL\_PLASTIC:  
    case MATERIAL\_PLASTIC\_TEXTURED:  
    g\_dynamicLinkageArray[g\_iMaterialOffset] = g\_pMaterialClasses[ MATERIAL\_PLASTIC\_LIGHTING\_ONLY ];  
    break;  
    case MATERIAL\_ROUGH:  
    case MATERIAL\_ROUGH\_TEXTURED:  
    g\_dynamicLinkageArray[g\_iMaterialOffset] = g\_pMaterialClasses[ MATERIAL\_ROUGH\_LIGHTING\_ONLY ];  
    break;  
    }  
    }  
    else  
    g\_dynamicLinkageArray[g\_iMaterialOffset] =     g\_pMaterialClasses[ g\_iMaterial ] ;

The Direct3D 11 runtime efficiently links each of the selected methods at source level, inlining and optimizing the shader code as much as possible to provide an optimal shader for the GPU to execute.

# 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/)