SimpleBezier11

<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. Newer versions of this sample for PC, UWP and Xbox can be found here:**

<https://github.com/Microsoft/Xbox-ATG-Samples>

*This sample requires Feature Level 11.0 or better.*

# Description



DirectX 11 introduces three new stages to the graphics pipeline: the Hull-Shader Stage, the fixed-function Tessellator Stage, and the Domain-Shader Stage. Together, these stages operate in between the vertex shader and geometry shader to provide the flexibility to implement a variety of different surface representations, including Bezier patches and subdivision surfaces. This sample demonstrates the basic usage of the DirectX 11 tessellation feature to render a simple cubic Bezier patch.

## Setup

The first and most drastic change required when using the tessellator is to the input mesh data. Instead of using triangles, you can now use any surface representation for your mesh data. This sample uses four cubic Bezier patches, each with 16 control points. After that, you must set the hull and domain shader and tell DirectX the input topology using the new D3D11\_PRIMITIVE\_TOPOLOGY types.

pd3dImmediateContext->HSSetShader( pHullShader, NULL, 0 );  
pd3dImmediateContext->DSSetShader( pDomainShader, NULL, 0 );  
pd3dImmediateContext->IASetPrimitiveTopology( D3D11\_PRIMITIVE\_TOPOLOGY\_16\_CONTROL\_POINT\_PATCHLIST );

### Vertex Shader

The first step of the pipeline is the vertex shader. This is largely unchanged, except for the distinction that it now operates on the input control points from the mesh before they continue on to the domain shader. For example, you might implement control point skinning in the vertex shader. The input to the vertex shader comes from the vertex buffer. The output from the vertex shader will go into the hull shader.

### Hull Shader

The hull shader is the first of the new stages added in DirectX 11. It operates in two phases; the first is per output control point, controlled by the outputcontrolpoints attribute. The input comes from the transformed control points from the vertex shader. The output control points will continue on to the domain shader. For a simple Bezier patch, nothing extra needs to be done and control points can be passed straight to the domain shader to be evaluated. For more complex surfaces, the hull shader might perform a basis conversion or generate the control points from adjacency information. The number of input control points and output control points are not necessarily the same, but in this sample they are.

The second phase of the hull shader is the constant function. It is invoked once per patch and specified with the **patchconstantfunc** attribute. The simplest constant function will output the tessellation factors **SV\_TessFactor** and **SV\_InsideTessFactor** that the fixed-function tessellator stage will use.

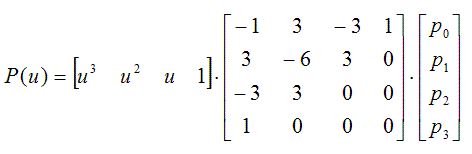
### Fixed-Function Tessellator

The next state of the pipeline is the fixed-function tessellator. This portion subdivides the input domain into the number primitives requested by the tessellation factor values. This stage does not care about the control points at all; it is concerned only with generating the output primitives and UVW coordinates for the domain shader. The input consists of the tessellation factors from **SV\_TessFactor** and **SV\_InsideTessFactor** and the partitioning mode domain from the **partitioning** and **domain** attributes respectively. From this it will output the **SV\_DomainLocation** normalized UVW coordinates for the domain shader.

### Domain Shader

The domain shader is the final stage of the tessellation pipeline added in DirectX 11. This shader operates on the output vertices created by the fixed-function tessellator. The domain shader is responsible for calculating the final vertex's position and attributes from the control points. This is where the Bezier surface is evaluated to calculate the positions.

The input control points come from the hull shader and the input UVW coordinates come from fixed-function tessellator. Using this information, the Bezier surface can be evaluated in the U and V directions with matrix form of the cubic Bezier curve function:



Finally, the surface normal can be evaluated as the cross-product of the U and V derivatives.

### Geometry Shader

The geometry shader is not illustrated in this sample; however, when used, it operates after the domain shader and takes the domain shader's outputs as its inputs.

# 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

[Direct3D 11 Tessellation](https://walbourn.github.io/direct3d-11-tessellation/)

[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/)