**GLSL –> HLSL conversion**

It would probably be possible to write an automatic conversion tool to turn a GLSL shader into an HLSL shader, but I’m not aware of one. So I’ve been simply stepping through each line of the shader and replacing any GLSL-specific functions with the HLSL equivalent. Fortunately, many functions are the same, others have direct equivalents and, as a shader is typically less than 100 lines long, what sounds like a manual process only takes a few minutes in practice.

Microsoft have published a very useful reference guide [here](https://msdn.microsoft.com/en-GB/library/windows/apps/dn166865.aspx) which details many of the general differences between GLSL and HLSL. Unity also have a useful page [here](http://docs.unity3d.com/Manual/SL-PlatformDifferences.html). The following is a list of specific issues I’ve encountered when converting Shadertoy shaders to Unity:

* Replace **iGlobalTime** shader input (“shader playback time in seconds”) with **\_Time.y**
* Replace **iResolution.xy** (“viewport resolution in pixels”) with **\_ScreenParams.xy**
* Replace **vec2** types with **float2**, **mat2** with **float2x2** etc.
* Replace **vec3(1)** shortcut constructors in which all elements have same value with explicit **float3(1,1,1)**
* Replace **Texture2D** with **Tex2D**
* Replace **atan(x,y)** with **atan2(y,x)** <- Note parameter ordering!
* Replace **mix()** with **lerp()**
* Replace **\*=** with **mul()**
* Remove third (bias) parameter from Texture2D lookups
* **mainImage(out vec4 fragColor, in vec2 fragCoord)** is the fragment shader function, equivalent to **float4 mainImage(float2 fragCoord : SV\_POSITION) : SV\_Target**
* UV coordinates in GLSL have 0 at the top and increase downwards, in HLSL 0 is at the bottom and increases upwards, so you may need to use **uv.y = 1 – uv.y** at some point.

Note that ShaderToys don’t have a vertex shader function – they are effectively full-screen pixel shaders which calculate the value at each UV coordinate in screenspace. As such, they are most suitable for use in a full-screen image effect (or, you can just apply them to a plane/quad if you want) in which the UVs range from 0-1.

# GLSL-to-HLSL reference

* 10/13/2015
* 11 minutes to read

[This article is for Windows 8.x and Windows Phone 8.x developers writing Windows Runtime apps. If you’re developing for Windows 10, see the [latest documentation](http://go.microsoft.com/fwlink/p/?linkid=619133)]

You port your OpenGL Shader Language (GLSL) code to Microsoft High Level Shader Language (HLSL) code when you [port your graphics architecture from OpenGL ES 2.0 to Direct3D 11](https://docs.microsoft.com/en-us/previous-versions/windows/apps/dn166874%28v%3dwin.10%29) to create a game for Windows 8. The GLSL that is referred to herein is compatible with OpenGL ES 2.0; the HLSL is compatible with Direct3D 11. For info about the differences between Direct3D 11 and previous versions of Direct3D, see [Feature mapping](https://docs.microsoft.com/en-us/previous-versions/windows/apps/dn166864%28v%3dwin.10%29).

* Comparing OpenGL ES 2.0 with Direct3D 11
* Porting GLSL variables to HLSL
* Porting GLSL types to HLSL
* Porting GLSL pre-defined global variables to HLSL
* Examples of porting GLSL variables to HLSL
  + Uniform, attribute, and varying in GLSL
  + Constant buffers and data transfers in HLSL
* Examples of porting OpenGL rendering code to Direct3D
* Related topics

## Comparing OpenGL ES 2.0 with Direct3D 11

OpenGL ES 2.0 and Direct3D 11 have many similarities. They both have similar rendering pipelines and graphics features. But Direct3D 11 is a rendering implementation and API, not a specification; OpenGL ES 2.0 is a rendering specification and API, not an implementation. Direct3D 11 and OpenGL ES 2.0 generally differ in these ways:

| **OpenGL ES 2.0** | **Direct3D 11** |
| --- | --- |
| Hardware and operating system agnostic specification with vendor provided implementations | Microsoft implementation of hardware abstraction and certification on Windows platforms |
| Abstracted for hardware diversity, runtime manages most resources | Direct access to hardware layout; app can manage resources and processing |
| Provides higher-level modules via third-party libraries (for example, Simple DirectMedia Layer (SDL)) | Higher-level modules, like Direct2D, are built upon lower modules to simplify development for Windows apps |
| Hardware vendors differentiate via extensions | Microsoft adds optional features to the API in a generic way so they aren't specific to any particular hardware vendor |

GLSL and HLSL generally differ in these ways:

| **GLSL** | **HLSL** |
| --- | --- |
| Procedural, step-centric (C like) | Object oriented, data-centric (C++ like) |
| Shader compilation integrated into the graphics API | The HLSL compiler [compiles the shader](https://msdn.microsoft.com/en-us/library/Bb509633) to an intermediate binary representation before Direct3D passes it to the driver.  **Note**  This binary representation is hardware independent. It's typically compiled at app build time, rather than at app run time. |
| Variable storage modifiers | Constant buffers and data transfers via input layout declarations |
| Types  Typical vector type: vec2/3/4  lowp, mediump, highp | Typical vector type: float2/3/4  min10float, min16float |
| texture2D [Function] | [texture.Sample](https://msdn.microsoft.com/en-us/library/Bb509695) [datatype.Function] |
| sampler2D [datatype] | [Texture2D](https://msdn.microsoft.com/en-us/library/Ff471525) [datatype] |
| Row-major matrices (default) | Column-major matrices (default)  **Note**  Use the **row\_major** type-modifier to change the layout for one variable. For more info, see [Variable Syntax](https://msdn.microsoft.com/en-us/library/Bb509706). You can also specify a compiler flag or a pragma to change the global default. |
| Fragment shader | Pixel shader |

**Note**  HLSL has textures and samplers as two separate objects. In GLSL, like Direct3D 9, the texture binding is part of the sampler state.

In GLSL, you present much of the OpenGL state as pre-defined global variables. For example, with GLSL, you use the **gl\_Position** variable to specify vertex position and the **gl\_FragColor** variable to specify fragment color. In HLSL, you pass Direct3D state explicitly from the app code to the shader. For example, with Direct3D and HLSL, the input to the vertex shader must match the data format in the vertex buffer, and the structure of a constant buffer in the app code must match the structure of a constant buffer ([cbuffer](https://msdn.microsoft.com/en-us/library/Bb509581)) in shader code.

## Porting GLSL variables to HLSL

In GLSL, you apply modifiers (qualifiers) to a global shader variable declaration to give that variable a specific behavior in your shaders. In HLSL, you don’t need these modifiers because you define the flow of the shader with the arguments that you pass to your shader and that you return from your shader.

| **GLSL variable behavior** | **HLSL equivalent** |
| --- | --- |
| **uniform**  You pass a uniform variable from the app code into either or both vertex and fragment shaders. You must set the values of all uniforms before you draw any triangles with those shaders so their values stay the same throughout the drawing of a triangle mesh. These values are uniform. Some uniforms are set for the entire frame and others uniquely to one particular vertex-pixel shader pair.  Uniform variables are per-polygon variables. | Use constant buffer.  See [How to: Create a Constant Buffer](https://msdn.microsoft.com/en-us/library/Ff476896) and [Shader Constants](https://msdn.microsoft.com/en-us/library/Bb509581). |
| **varying**  You initialize a varying variable inside the vertex shader and pass it through to an identically named varying variable in the fragment shader. Because the vertex shader only sets the value of the varying variables at each vertex, the rasterizer interpolates those values (in a perspective-correct manner) to generate per fragment values to pass into the fragment shader. These variables vary across each triangle. | Use the structure that you return from your vertex shader as the input to your pixel shader. Make sure the semantic values match. |
| **attribute**  An attribute is a part of the description of a vertex that you pass from the app code to the vertex shader alone. Unlike a uniform, you set each attribute’s value for each vertex, which, in turn, allows each vertex to have a different value. Attribute variables are per-vertex variables. | Define a vertex buffer in your Direct3D app code and match it to the vertex input defined in the vertex shader. Optionally, define an index buffer. See [How to: Create a Vertex Buffer](https://msdn.microsoft.com/en-us/library/Ff476899) and [How to: Create an Index Buffer](https://msdn.microsoft.com/en-us/library/Ff476897).  Create an input layout in your Direct3D app code and match semantic values with those in the vertex input. See [Create the input layout](https://msdn.microsoft.com/en-us/library/Bb205117). |
| **const**  Constants that are compiled into the shader and never change. | Use a **static const**. **static** means the value isn't exposed to constant buffers, **const** means the shader can't change the value. So, the value is known at compile time based on its initializer. |

In GLSL, variables without modifiers are just ordinary global variables that are private to each shader.

When you pass data to textures ([Texture2D](https://msdn.microsoft.com/en-us/library/Ff471525) in HLSL) and their associated samplers ([SamplerState](https://msdn.microsoft.com/en-us/library/Bb509644) in HLSL), you typically declare them as global variables in the pixel shader.

## Porting GLSL types to HLSL

Use this table to port your GLSL types to HLSL.

| **GLSL type** | **HLSL type** |
| --- | --- |
| scalar types: float, int, bool | scalar types: float, int, bool  also, uint, double  For more info, see [Scalar Types](https://msdn.microsoft.com/en-us/library/Bb509646). |
| vector type   * floating-point vector: vec2, vec3, vec4 * Boolean vector: bvec2, bvec3, bvec4 * signed integer vector: ivec2, ivec3, ivec4 | vector type   * float2, float3, float4, and float1 * bool2, bool3, bool4, and bool1 * int2, int3, int4, and int1 * These types also have vector expansions similar to float, bool, and int:   + uint   + min10float, min16float   + min12int, min16int   + min16uint   For more info, see [Vector Type](https://msdn.microsoft.com/en-us/library/Bb509707) and [Keywords](https://msdn.microsoft.com/en-us/library/Bb509568).  vector is also type defined as float4 (typedef vector <float, 4> vector;). For more info, see [User-Defined Type](https://msdn.microsoft.com/en-us/library/Bb509702). |
| matrix type   * mat2: 2x2 float matrix * mat3: 3x3 float matrix * mat4: 4x4 float matrix | matrix type   * float2x2 * float3x3 * float4x4 * also, float1x1, float1x2, float1x3, float1x4, float2x1, float2x3, float2x4, float3x1, float3x2, float3x4, float4x1, float4x2, float4x3 * These types also have matrix expansions similar to float:   + int, uint, bool   + min10float, min16float   + min12int, min16int   + min16uint   You can also use the [matrix type](https://msdn.microsoft.com/en-us/library/Bb509623) to define a matrix.  For example: matrix <float, 2, 2> fMatrix = {0.0f, 0.1, 2.1f, 2.2f};  matrix is also type defined as float4x4 (typedef matrix <float, 4, 4> matrix;). For more info, see [User-Defined Type](https://msdn.microsoft.com/en-us/library/Bb509702). |
| precision qualifiers for float, int, sampler   * highp   This qualifier provides minimum precision requirements that are greater than that provided by min16float and less than a full 32-bit float. Equivalent in HLSL is:  highp float -> float  highp int -> int   * mediump   This qualifier applied to float and int is equivalent to min16float and min12int in HLSL. Minimum 10 bits of mantissa, not like min10float.   * lowp   This qualifier applied to float provides a floating point range of -2 to 2. Equivalent to min10float in HLSL. | precision types   * min16float: minimum 16-bit floating point value * min10float   Minimum fixed-point signed 2.8 bit value (2 bits of whole number and 8 bits fractional component). The 8-bit fractional component can be inclusive of 1 instead of exclusive to give it the full inclusive range of -2 to 2.   * min16int: minimum 16-bit signed integer * min12int: minimum 12-bit signed integer   This type is for 10Level9 ([9\_x feature levels](https://msdn.microsoft.com/en-us/library/Ff476876)) in which integers are represented by floating point numbers. This is the precision you can get when you emulate an integer with a 16-bit floating point number.   * min16uint: minimum 16-bit unsigned integer   For more info, see [Scalar Types](https://msdn.microsoft.com/en-us/library/Bb509646) and [Using HLSL minimum precision](https://msdn.microsoft.com/en-us/library/Hh968108). |
| sampler2D | [Texture2D](https://msdn.microsoft.com/en-us/library/Ff471525) |
| samplerCube | [TextureCube](https://msdn.microsoft.com/en-us/library/Bb509700) |

## Porting GLSL pre-defined global variables to HLSL

Use this table to port GLSL pre-defined global variables to HLSL.

| **GLSL pre-defined global variable** | **HLSL semantics** |
| --- | --- |
| **gl\_Position**  This variable is type **vec4**.  Vertex position  for example - gl\_Position = position; | SV\_Position  POSITION in Direct3D 9  This semantic is type **float4**.  Vertex shader output  Vertex position  for example - float4 vPosition : SV\_Position; |
| **gl\_PointSize**  This variable is type **float**.  Point size | PSIZE  No meaning unless you target Direct3D 9  This semantic is type **float**.  Vertex shader output  Point size |
| **gl\_FragColor**  This variable is type **vec4**.  Fragment color  for example - gl\_FragColor = vec4(colorVarying, 1.0); | SV\_Target  COLOR in Direct3D 9  This semantic is type **float4**.  Pixel shader output  Pixel color  for example - float4 Color[4] : SV\_Target; |
| **gl\_FragData[n]**  This variable is type **vec4**.  Fragment color for color attachment n | SV\_Target[n]  This semantic is type **float4**.  Pixel shader output value that is stored in n render target, where 0 <= n <= 7. |
| **gl\_FragCoord**  This variable is type **vec4**.  Fragment position within frame buffer | SV\_Position  Not available in Direct3D 9  This semantic is type **float4**.  Pixel shader input  Screen space coordinates  for example - float4 screenSpace : SV\_Position |
| **gl\_FrontFacing**  This variable is type **bool**.  Determines whether fragment belongs to a front-facing primitive. | SV\_IsFrontFace  VFACE in Direct3D 9  SV\_IsFrontFace is type **bool**.  VFACE is type **float**.  Pixel shader input  Primitive facing |
| **gl\_PointCoord**  This variable is type **vec2**.  Fragment position within a point (point rasterization only) | SV\_Position  VPOS in Direct3D 9  SV\_Position is type **float4**.  VPOS is type **float2**.  Pixel shader input  The pixel or sample position in screen space  for example - float4 pos : SV\_Position |
| **gl\_FragDepth**  This variable is type **float**.  Depth buffer data | SV\_Depth  DEPTH in Direct3D 9  SV\_Depth is type **float**.  Pixel shader output  Depth buffer data |

You use semantics to specify position, color, and so on for vertex shader input and pixel shader input. You must match the semantics values in the input layout with the vertex shader input. For examples, see Examples of porting GLSL variables to HLSL. For more info about the HLSL semantics, see [Semantics](https://msdn.microsoft.com/en-us/library/Bb509647).

## Examples of porting GLSL variables to HLSL

Here we show examples of using GLSL variables in OpenGL/GLSL code and then the equivalent example in Direct3D/HLSL code.

### Uniform, attribute, and varying in GLSL

OpenGL app code

// Uniform values can be set in app code and then processed in the shader code.

uniform mat4 model;

uniform mat4 view;

uniform mat4 projection;

// Incoming position of vertex

attribute vec4 position;

// Incoming color for the vertex

attribute vec3 color;

// The varying variable tells the shader pipeline to pass it

// on to the fragment shader.

varying vec3 colorVarying;

GLSL vertex shader code

//The shader entry point is the main method.

void main()

{

colorVarying = color; //Use the varying variable to pass the color to the fragment shader

gl\_Position = position; //Copy the position to the gl\_Position pre-defined global variable

}

GLSL fragment shader code

void main()

{

//Pad the colorVarying vec3 with a 1.0 for alpha to create a vec4 color

//and assign that color to the gl\_FragColor pre-defined global variable

//This color then becomes the fragment's color.

gl\_FragColor = vec4(colorVarying, 1.0);

}

### Constant buffers and data transfers in HLSL

Here is an example of how you pass data to the HLSL vertex shader that then flows through to the pixel shader. In your app code, define a vertex and a constant buffer. Then, in your vertex shader code, define the constant buffer as a [cbuffer](https://msdn.microsoft.com/en-us/library/Bb509581) and store the per-vertex data and the pixel shader input data. Here we use structures called **VertexShaderInput** and **PixelShaderInput**.

Direct3D app code

C++

struct ConstantBuffer

{

XMFLOAT4X4 model;

XMFLOAT4X4 view;

XMFLOAT4X4 projection;

};

struct SimpleCubeVertex

{

XMFLOAT3 pos; // position

XMFLOAT3 color; // color

};

// Create an input layout that matches the layout defined in the vertex shader code.

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

{

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

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

};

// Create vertex and index buffers that define a geometry.

HLSL vertex shader code

cbuffer ModelViewProjectionCB : register( b0 )

{

matrix model;

matrix view;

matrix projection;

};

// The POSITION and COLOR semantics must match the semantics in the input layout Direct3D app code.

struct VertexShaderInput

{

float3 pos : POSITION; // Incoming position of vertex

float3 color : COLOR; // Incoming color for the vertex

};

struct PixelShaderInput

{

float4 pos : SV\_Position; // Copy the vertex position.

float4 color : COLOR; // Pass the color to the pixel shader.

};

PixelShaderInput main(VertexShaderInput input)

{

PixelShaderInput vertexShaderOutput;

// shader source code

return vertexShaderOutput;

}

HLSL pixel shader code

// Collect input from the vertex shader.

// The COLOR semantic must match the semantic in the vertex shader code.

struct PixelShaderInput

{

    float4 pos : SV\_Position;

    float4 color : COLOR; // Color for the pixel

};

// Set the pixel color value for the renter target.

float4 main(PixelShaderInput input) : SV\_Target

{

    return input.color;

}

## Examples of porting OpenGL rendering code to Direct3D

Here we show an example of rendering in OpenGL ES 2.0 code and then the equivalent example in Direct3D 11 code.

OpenGL rendering code

// Bind shaders to the pipeline.

// Both vertex shader and fragment shader are in a program.

glUseProgram(m\_shader->getProgram());

// Input asssembly

// Get the position and color attributes of the vertex.

m\_positionLocation = glGetAttribLocation(m\_shader->getProgram(), "position");

glEnableVertexAttribArray(m\_positionLocation);

m\_colorLocation = glGetAttribColor(m\_shader->getProgram(), "color");

glEnableVertexAttribArray(m\_colorLocation);

// Bind the vertex buffer object to the input assembler.

glBindBuffer(GL\_ARRAY\_BUFFER, m\_geometryBuffer);

glVertexAttribPointer(m\_positionLocation, 4, GL\_FLOAT, GL\_FALSE, 0, NULL);

glBindBuffer(GL\_ARRAY\_BUFFER, m\_colorBuffer);

glVertexAttribPointer(m\_colorLocation, 3, GL\_FLOAT, GL\_FALSE, 0, NULL);

// Draw a triangle with 3 vertices.

glDrawArray(GL\_TRIANGLES, 0, 3);

Direct3D rendering code

C++

// Bind the vertex shader and pixel shader to the pipeline.

m\_d3dDeviceContext->VSSetShader(vertexShader.Get(),nullptr,0);

m\_d3dDeviceContext->PSSetShader(pixelShader.Get(),nullptr,0);

// Declare the inputs that the shaders expect.

m\_d3dDeviceContext->IASetInputLayout(inputLayout.Get());

m\_d3dDeviceContext->IASetVertexBuffers(0, 1, vertexBuffer.GetAddressOf(), &stride, &offset);

// Set the primitive’s topology.

m\_d3dDeviceContext->IASetPrimitiveTopology(D3D11\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST);

// Draw a triangle with 3 vertices. triangleVertices is an array of 3 vertices.

m\_d3dDeviceContext->Draw(ARRAYSIZE(triangleVertices),0);