

# Getting Started with XShaderCompiler

Lukas Hermanns

February 27, 2017



Contents

- 1 Introduction 3
- 2 Progress 4
  - 2.1 ToDo List . . . . . 4
- 3 Offline Compiler 5
  - 3.1 Commands . . . . . 5
  - 3.2 Example . . . . . 6
- 4 Limitations 7
  - 4.1 Tessellation Shaders . . . . . 7

# 1 Introduction

[XShaderCompiler](#) (“Cross Shader Compiler”) is a cross-compiler (also called trans-compiler), which translates HLSL code (DirectX High Level Shading Language, see [msdn.microsoft.com](http://msdn.microsoft.com)) of Shader Model 4 and 5 into GLSL code (OpenGL Shading Language, see [www.opengl.org](http://www.opengl.org)).

## 2 Progress

This project is still in its early steps. This document is written for [XShaderCompiler Version 0.06 Alpha](#).

### 2.1 ToDo List

See TODO.md file on [github.com](https://github.com)

## 3 Offline Compiler

The offline compiler (named `xsc`) can be used to cross-compile your shaders without building any custom application. It has similar commands like other common compilers (such as GCC), e.g. `-O` to enable optimization. To show the description of all commands, type simply `xsc` or `xsc --help` into a terminal or command line.

### 3.1 Commands

Here is a brief overview of the most important commands:

**-T, --target *TARGET***

Sets the shader target specified by *TARGET*. These values are adopted from LunarG's reference compiler `glslang`. Valid values for *TARGET* are:

**vert** (for Vertex Shader)

**tesc** (for Tessellation Control Shader, also called Hull Shader)

**tese** (for Tessellation Evaluation Shader, also called Domain Shader)

**geom** (for Geometry Shader)

**frag** (for Fragment Shader, also called Pixel Shader)

**comp** (for Compute Shader)

**-E, --entry *ENTRY***

Sets the shader entry point (i.e. main function) specified by *ENTRY*.

**-I, --include-path *PATH***

Adds the file path, specified by *PATH*, to the include search paths.

**-o, --output *FILE***

Sets the filename of the output file specified by *FILE*. The default value is "*FILE.ENTRY.TARGET*", where *FILE* is the filename of the input shader file, *ENTRY* is the shader entry point, and *TARGET* is the shader output target. The asterisk character `'*'` can be included to re-use the default value, e.g. `"OutputFolder/*"` will result into `"OutputFolder/FILE.ENTRY.TARGET"`

**-Vin, --version-in *VERSION***

Sets the input shader version specified by *VERSION*. Valid values for *VERSION* are:

**HLSL3** (for HLSL Shader Model 3)

**HLSL4** (for HLSL Shader Model 4)

**HLSL5** (for HLSL Shader Model 5) *default value*

**-Vout, --version-out *VERSION***

Sets the output shader version specified by *VERSION*. Valid values for *VERSION* are:

**GLSL** (for automatic deduction of the minimal required GLSL version) *default value*

**GLSL110** (for GLSL 1.10) *not supported yet*

**GLSL120** (for GLSL 1.20) *not supported yet*

**GLSL130** (for GLSL 1.30)

**GLSL140** (for GLSL 1.40)

**GLSL150** (for GLSL 1.50)

**GLSL330** (for GLSL 3.30)

**GLSL400** (for GLSL 4.00)

**GLSL410** (for GLSL 4.10)

**GLSL420** (for GLSL 4.20)

**GLSL430** (for GLSL 4.30)

**GLSL440** (for GLSL 4.40)

**GLSL450** (for GLSL 4.50)

## 3.2 Example

Here is a small use case example. Consider the following minimal HLSL vertex shader, stored in a file named “Example.hlsl”:

```
float4 VertexMain(float3 coord : COORD) : SV_Position
{
    return float4(coord, 1);
}
```

Now enter the following into your command prompt:

```
xsc -T vert -E VertexMain Example.hlsl
```

The resulting GLSL shader will be stored in a file named “Example.VertexMain.vert”, and looks like this:

```
#version 130

in vec3 coord;

void main()
{
    gl_Position = vec4(coord, 1);
}
```

## 4 Limitations

There are several limitations for your HLSL shaders you want to translate to GLSL with the [XShaderCompiler](#) which are described in this section.

### 4.1 Tessellation Shaders

*(The translation of tessellation shaders is currently in progress but here is a brief overview of the currently known limitations)*

The most tessellation attributes in HLSL are specified for the tessellation-control shader (alias “Hull Shader”), but a few of them are required for the tessellation-evaluation shader (alias “Domain Shader”). These are: `partitioning`, and `outputtopology`. Here is an example of an HLSL Tessellation Shader:

Example.hls1

```
[domain("quad")] // Required for Tessellation-Control (in GLSL)
[outputcontrolpoints(4)] // Required for Tessellation-Control (in GLSL)
[patchconstantfunc("PatchConstantFunc")] // Required for Tessellation-Control (in GLSL)
[partitioning("fractional_odd")] // Required for Tessellation-Evaluation (in GLSL)
[outputtopology("triangle_ccw")] // Required for Tessellation-Evaluation (in GLSL)
OutputHS HullShader(/* ... */)
{
    /* ... */
}

[domain("quad")] // Required for Tessellation-Evaluation (in GLSL)
OutputDS DomainShader(/* ... */)
{
    /* ... */
}
```

These attribute must be distributed into two GLSL shaders:

Example.HullShader.tesc

```
layout(vertices = 4) in;
/* ... */
```

Example.DomainShader.tese

```
layout(quads, fractional_odd_spacing, ccw) in;
/* ... */
```

The information for `fractional_odd_spacing` and `ccw` in the `Example.DomainShader.tese` shader file are taken from the tessellation-control shader, although a tessellation-evaluation shader is written. That means, both the tessellation-control- *and* the tessellation-evaluation shaders must be contained in the same shader source file (or at least in one of the included files) to guarantee a full translation of all information. Otherwise default values will be used.

To specify the secondary entry point (in the above example “HullShader”) use the `secondaryEntryPoint` member in the `Xsc::ShaderInput` structure or the “-E2, --entry2 ENTRY” shell command.