High-Level Shader Language Specification

Working Draft

November 8, 2023

Contents

1 Introduction [Intro]

1 The High Level Shader Language (HLSL) is the GPU programming language provided in conjunction with the DirectX runtime. Over many years its use has expanded to cover every major rendering API across all major development platforms. Despite its popularity and long history HLSL has never had a formal language specification. This document seeks to change that.

2 HLSL draws heavy inspiration originally from ISO/IEC 9899:2018 and later from ISO/IEC 14882:2020 with additions specific to graphics and parallel computation programming. The language is also influenced to a lesser degree by other popular graphics and parallel programming languages.

3 HLSL has two reference implementations which this specification draws heavily from. The original reference implementation Legacy DirectX Shader Compiler (FXC) has been in use since DirectX 9. The more recent reference implementation DirectX Shader Compiler (DXC) has been the primary shader compiler since DirectX 12.

4 In writing this specification bias is leaned toward the language behavior of DXC rather than the behavior of FXC, although that can vary by context.

5 In very rare instances this spec will be aspirational, and may diverge from both reference implementation behaviors. This will only be done in instances where there is an intent to alter implementation behavior in the future. Since this document and the implementations are living sources, one or the other may be ahead in different regards at any point in time.

1.1 Scope [Intro.Scope]

1 This document specifies the requirements for implementations of HLSL. The HLSL specification is based on and highly influenced by the specifications for the C Programming Language (C) and the C++ Programming Language (C++).

2 This document covers both describing the language grammar and semantics for HLSL, and (in later sections) the standard library of data types used in shader programming.

1.2 Normative References [Intro.Refs]

1 The following referenced documents provide significant influence on this document and should be used in conjunction with interpreting this standard.

- ISO/IEC 9899:2018, *Programming languages - C*
- ISO/IEC 14882:2020, *Programming languages - C++*
- DirectX Specifications, *https://microsoft.github.io/DirectX-Specs/*

1.3 Terms and definitions [Intro.Terms]

1 This document aims to use terms consistent with their definitions in ISO/IEC 9899:2018 and ISO/IEC 14882:2020. In cases where the definitions are unclear, or where this document diverges this section, the remaining sections in this chapter, and the attached 5.1.

1.4 Runtime Targeting [Intro.Runtime]

1 HLSL emerged from the evolution of DirectX to grant greater control over GPU geometry and color processing. It gained popularity because it targeted a common hardware description which all conforming drivers were required to support. This common hardware description, called a Shader Model, is an integral part of the description for HLSL . Some HLSL features require specific Shader Model features, and are only supported by compilers when targeting those Shader Model versions or later.

1.5 Single Program Multiple Data (SPMD) Programming Model [Intro.Model]

1 HLSL uses a Single Program Multiple Data (SPMD) programming model where a program describes operations on a single element of data, but when the program executes it executes across more than one element at a time. This programming model is useful due to GPUs largely being Single Instruction Multiple Data (SIMD) hardware architectures where each instruction natively executes across multiple data elements at the same time.

2 There are many different terms of art for describing the elements of a GPU architecture and the way they relate to the SPMD program model. In this document we will use the terms as defined in the following subsections.

1.5.1 Lane [Intro.Model.Lane]

1 A Lane represents a single computed element in an SPMD program. In a traditional programming model it would be analogous to a thread of execution, however it differs in one key way. In multi-threaded programming threads advance independent of each other. In SPMD programs, a group of Lanes execute instructions in lock step because each instruction is a SIMD instruction computing the results for multiple Lanes simultaneously.

1.5.2 Wave [Intro.Model.Wave]

1 A grouping of Lanes for execution is called a Wave. Wave sizes vary by hardware architecture. Some hardware implementations support multiple wave sizes. Generally wave sizes are powers of two, but there is no requirement that be the case. HLSL is explicitly designed to run on hardware with arbitrary Wave sizes.

1.5.3 Quad [Intro.Model.Quad]

1 A Quad is a subdivision of four Lanes in a Wave which are computing adjacent values. In pixel shaders a Quad may represent four adjacent pixels and Quad operations allow passing data between adjacent lanes. In compute shaders quads may be one or two dimensional depending on the workload dimensionality described in the `numthreads` attribute on the entry function. (FIXME: Add reference to attribute)

1.5.4 Thread Group [Intro.Model.Group]

1 A grouping of Waves executing the same shader to produce a combined result is called a Thread Group. Thread Groups are executed on separate SIMD hardware and are not instruction locked with other Thread Groups.

1.5.5 Dispatch [Intro.Model.Dispatch]

1 A grouping of Thread Groups which represents the full execution of a HLSL program and results in a completed result for all input data elements.

1.6 HLSL Memory Models [Intro.Memory]

1 Memory accesses for Shader Model 5.0 and earlier operate on 128-bit slots aligned on 128-bit boundaries. This optimized for the common case in early shaders where data being processed on the GPU was usually 4-element vectors of 32-bit data types.

2 On modern hardware memory access restrictions are loosened, and reads of 32-bit multiples are supported starting with Shader Model 5.1 and reads of 16-bit multiples are supported with Shader Model 6.0. Shader Model features are fully documented in the DirectX Specifications, and this document will not attempt to elaborate further.

1.7 Common Definitions [Intro.Defs]

1 The following definitions are consistent between HLSL and the ISO/IEC 9899:2018 and ISO/IEC 14882:2020 specifications, however they are included here for reader convenience.

1.7.1 Diagnostic Message [Intro.Defs.Diags]

1 An implementation defined message belonging to a subset of the implementation's output messages which communicates diagnostic information to the user.

1.7.2 Ill-formed Program [Intro.Defs.IllFormed]

1 A program that is not well formed, for which the implementation is expected to return unsuccessfully and produce one or more diagnostic messages.

1.7.3 Implementation-defined Behavior [Intro.Defs.ImpDef]

1 Behavior of a well formed program and correct data which may vary by the implementation, and the implementation is expected to document the behavior.

1.7.4 Implementation Limits [Intro.Defs.ImpLimits]

1 Restrictions imposed upon programs by the implementation.

1.7.5 Undefined Behavior [Intro.Defs.Undefined]

1 Behavior of invalid program constructs or incorrect data which this standard imposes no requirements, or does not sufficiently detail.

1.7.6 Unspecified Behavior [Intro.Defs.Unspecified]

1 Behavior of a well formed program and correct data which may vary by the implementation, and the implementation is not expected to document the behavior.

1.7.7 Well-formed Program [Intro.Defs.WellFormed]

1 An HLSL program constructed according to the syntax rules, diagnosable semantic rules, and the One Definition Rule.

2 Lexical Conventions [Lex]

2.1 Unit of Translation [Lex.Translation]

1 The text of HLSL programs is collected in *source* and *header* files. The distinction between source and header files is social and not technical. An implementation will construct a *translation unit* from a single source file and any included source or header files referenced via the `#include` preprocessing directive conforming to the ISO/IEC 9899:2018 preprocessor specification.

2 An implementation may implicitly include additional sources as required to expose the HLSL library functionality as defined in (FIXME: Add reference to library chapter).

2.2 Phases of Translation [Lex.Phases]

1 HLSL inherits the phases of translation from ISO/IEC 14882:2020, with minor alterations, specifically the removal of support for trigraph and digraph sequences. Below is a description of the phases.

1. Source files are characters that are mapped to the basic source character set in an implementation-defined manner.
2. Any sequence of backslash (\) immediately followed by a new line is deleted, resulting in splicing lines together.
3. Tokenization occurs and comments are isolated. If a source file ends in a partial comment or preprocessor token the program is ill-formed and a diagnostic shall be issued. Each comment block shall be treated as a single white-space character.
4. Preprocessing directives are executed, macros are expanded, `pragma` and other unary operator expressions are executed. Processing of `#include` directives results in all preceding steps being executed on the resolved file, and can continue recursively. Finally all preprocessing directives are removed from the source.
5. Character and string literal specifiers are converted into the appropriate character set for the execution environment.
6. Adjacent string literal tokens are concatenated.
7. White-space is no longer significant. Syntactic and semantic analysis occurs translating the whole translation unit into an implementation-defined representation.
8. The translation unit is processed to determine required instantiations, the definitions of the required instantiations are located, and the translation and instantiation units are merged. The program is ill-formed if any required instantiation cannot be located or fails during instantiation.
9. External references are resolved, library references linked, and all translation output is collected into a single output.

2.3 Character Sets [Lex.CharSet]

1 The *basic source character set* is a subset of the ASCII character set. The table below lists the valid characters and their ASCII values:

Hex ASCII Value	Character Name	Glyph or C Escape Sequence
0x09	Horizontal Tab	`\t`
0x0A	Line Feed	`\n`
0x0D	Carriage Return	`\r`
0x20	Space	
0x21	Exclamation Mark	!
0x22	Quotation Mark	"
0x23	Number Sign	#
0x25	Percent Sign	%
0x26	Ampersand	&
0x27	Apostrophe	'
0x28	Left Parenthesis	(
0x29	Right Parenthesis	)
0x2A	Asterisk	*
0x2B	Plus Sign	+
0x2C	Comma	,
0x2D	Hyphen-Minus	-
0x2E	Full Stop	.
0x2F	Solidus	/
0x30 .. 0x39	Digit Zero .. Nine	0 1 2 3 4 5 6 7 8 9
0x3A	Colon	:
0x3B	Semicolon	;
0x3C	Less-than Sign	<
0x3D	Equals Sign	=
0x3E	Greater-than Sign	>
0x3F	Question Mark	?
0x41 .. 0x5A	Latin Capital Letter A .. Z	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0x5B	Left Square Bracket	[
0x5C	Reverse Solidus	\
0x5D	Right Square Bracket	[
0x5E	Circumflex Accent	^
0x5F	Underscore	_
0x61 .. 0x7A	Latin Small Letter a .. z	a b c d e f g h i j k l m n o p q r s t u v w x y z
0x7B	Left Curly Bracket	{
0x7C	Vertical Line	|
0x7D	Right Curly Bracket	}

2 An implementation may allow source files to be written in alternate *extended character sets* as long as that set is a superset of the *basic character set*. The *translation character set* is an *extended character set* or the *basic character set* as chosen by the implementation.

2.4 Preprocessing Tokens [Lex.PPTokens]

preprocessing-token:
- *header-name*
- *identifier*
- *pp-number*
- *character-literal*
- *string-literal*
- *preprocessing-op-or-punc*
- each non-whitespace character from the *translation character set* that cannot be one of the above

[1]

[1]The preprocessor is inherited from C++ 11 with no grammar extensions. It is specified here only for completeness.

1 Each preprocessing token that is converted to a token shall have the lexical form of a keyword, an identifier, a constant, a string literal or an operator or punctuator.

2 Preprocessing tokens are the minimal lexical elements of the language during translation phases 3 through 6 (2.2). Preprocessing tokens can be separated by whitespace in the form of comments, white space characters, or both. White space may appear within a preprocessing token only as part of a header name or between the quotation characters in a character constant or string literal.

3 Header name preprocessing tokens are only recognized within `#include` preprocessing directives, `__has_include` expressions, and implementation-defined locations within `#pragma` directives. In those contexts, a sequence of characters that could be either a header name or a string literal is recognized as a header name.

2.5 Tokens [Lex.Tokens]

token:
  identifier
  keyword
  literal
  operator-or-punctuator

1 There are five kinds of tokens: identifiers, keywords, literals, and operators or punctuators. All whitespace characters and comments are ignored except as they separate tokens.

2.6 Comments [Lex.Comments]

1 The characters `/*` start a comment which terminates with the characters `*\`. The characters `//` start a comment which terminates at the next new line.

2.7 Header Names [Lex.Headers]

header-name:
  < *h-char-sequence* >
  " *h-char-sequence* "

h-char-sequence:
  h-char
  h-char-sequence h-char

h-char:
  any character in the *translation character set* except newline or >

q-char-sequence:
  q-char
  q-char-sequence q-char

q-char:
  any character in the *translation character set* except newline or "

1 Character sequences in header names are mapped to header files or external source file names in an implementation defined way.

3 Basic Concepts [Basic]

3.1 Lvalues and rvalues [Basic.lval]

1 Expressions are classified by the type(s) of values they produce. The valid types of values produced by expressions are:

1. An *lvalue* represents a function or object.
2. An *rvalue* represents a temporary object.
3. An *xvalue* (expiring value) represents an object near the end of its lifetime.
4. A *cxvalue* (casted expiring value) is an *xvalue* which, on expiration, assigns its value to a bound *lvalue*.
5. A *glvalue* is an *lvalue*, *xvalue*, or *cxvalue*.
6. A *prvalue* is an *rvalue* that is not an *xvalue*.

4 Standard Conversions [Conv]

1 HLSL inherits standard conversions similar to ISO/IEC 14882:2020. This chapter enumerates the full set of conversions. A *standard conversion sequence* is a sequence of standard conversions in the following order:

1. Zero or one conversion of either lvalue-to-rvalue, array-to-pointer or function-to-pointer.
2. Zero or one conversion of either integral conversion, floating point conversion, floating point-integral conversion, or boolean conversion, derived-to-base-lvalue, vector splat, vector truncation, or flat conversion[1].
3. Zero or one conversion of either component-wise integral conversion, component-wise floating point conversion, component-wise floating point-integral conversion, or component-wise boolean conversion[2].
4. Zero or one qualification conversion.

Standard conversion sequences are applied to expressions, if necessary, to convert it to a required destination type.

4.1 Lvalue-to-rvalue conversion [Conv.lval]

1 A glvalue of a non-function type `T` can be converted to a prvalue. The program is ill-formed if `T` is an incomplete type. If the glvalue refers to an object that is not of type `T` and is not an object of a type derived from `T`, the program is ill-formed. If the glvalue refers to an object that is uninitialized, the behavior is undefined. Otherwise the prvalue is of type `T`.

2 If the glvalue refers to an array of type `T`, the prvalue will refer to a copy of the array, not memory referred to by the glvalue.

4.2 Array-to-pointer conversion [Conv.array]

1 An lvalue or rvalue of type `T[]` (bounded or unbounded), can be converted to a prvalue of type pointer to `T`. [*Note: HLSL does not support grammar for specifying pointer or reference types, however they are used in the type system and must be described in language rules.*]

4.3 Integral conversion [Conv.iconv]

1 A glvalue of an integer type can be converted to a cxvalue of any other non-enumeration integer type. A prvalue of an integer type can be converted to a prvalue of any other integer type.

2 If the destination type is unsigned, integer conversion maintains the bit pattern of the source value in the destination type truncating or extending the value to the destination type.

3 If the destination type is signed, the value is unchanged if the destination type can represent the source value. If the destination type cannot represent the source value, the result is implementation-defined.

4 If the source type is `bool`, the values `true` and `false` are converted to one and zero respectively.

4.4 Floating point conversion [Conv.fconv]

1 A glvalue of a floating point type can be converted to a cxvalue of any other floating point type. A prvalue of a floating point type can be converted to a prvalue of any other floating point type.

2 If the source value can be exactly represented in the destination type, the conversion produces the exact representation of the source value. If the source value cannot be exactly represented, the conversion to a best-approximation of the source value is implementation defined.

[1] This differs from C++ with the addition of vector splat and truncation casting and flat conversions.
[2] C++ does not support this conversion in the sequence for component-wise conversion of vector and matrix types.

4.5 Floating point-integral conversion [Conv.fpint]

1 A glvalue of floating point type can be converted to a cxvalue of integer type. A prvalue of floating point type can be converted to a prvalue of integer type. Conversion of floating point values to integer values truncates by discarding the fractional value. The behavior is undefined if the truncated value cannot be represented in the destination type.

2 A glvalue of integer type can be converted to a cxvalue of floating point type. A prvalue of integer type can be converted to a prvalue of floating point type. If the destination type can exactly represent the source value, the result is the exact value. If the destination type cannot exactly represent the source value, the conversion to a best-approximation of the source value is implementation defined.

4.6 Boolean conversion [Conv.bool]

1 A glvalue of arithmetic type can be converted to a cxvalue of boolean type. A prvalue of arithmetic or unscoped enumeration type can be converted to a prvalue of boolean type. A zero value is converted to `false`; all other values are converted to `true`.

4.7 Vector splat conversion [Conv.vsplat]

1 A glvalue of type `T` can be converted to a cxvalue of type `vector<T,x>` or a prvalue of type `T` can be converted to a prvalue of type `vector<T,x>`. The destination value is the source value replicated into each element of the destination.

2 A glvalue of type `T` can be converted to a cxvalue of type `matrix<T,x,y>` or a prvalue of type `T` can be converted to a prvalue of type `matrix<T,x,y>`. The destination value is the source value replicated into each element of the destination.

4.8 Vector and matrix truncation conversion [Conv.vtrunc]

1 A glvalue of type `vector<T,x>` can be converted to a cxvalue of type `vector<T,y>`, or a prvalue of type `vector<T,x>` can be converted to a prvalue of type `vector<T,y>` only if `x` is less than `y`.

2 A glvalue of type `matrix<T,x,y>` can be converted to a cxvalue of type `matrix<T,z,w>`, or a prvalue of type `matrix<T,x,y>` can be converted to a prvalue of type `matrix<T,z,w>` only if $x \leq z$ and $y \leq w$

4.9 Component-wise conversions [Conv.cwise]

1 A glvalue of type `vector<T,x>` can be converted to a cxvalue of type `vector<V,x>`, or a prvalue of type `vector<T,x>` can be converted to a prvalue of type `vector<V,x>`. The source value is converted by performing the appropriate conversion of each element of type `T` to an element of type `V` following the rules for standard conversions in chapter 4.

2 A glvalue of type `matrix<T,x,y>` can be converted to a cxvalue of type `matrix<V,x,y>`, or a prvalue of type `matrix<T,x,y>` can be converted to a prvalue of type `matrix<V,x,y>`. The source value is converted by performing the appropriate conversion of each element of type `T` to an element of type `V` following the rules for standard conversions in chapter 4.

4.10 Qualification conversion [Conv.qual]

A prvalue of type "*cv1* `T`" can be converted to a prvalue of type "*cv2* `T`" if type "*cv2* `T`" is more cv-qualified than "*cv1* `T`".

5 Expressions [Expr]

1 This chapter defines the formulations of expressions and the behavior of operators when they are not overloaded. Only member operators may be overloaded[1]. Operator overloading does not alter the rules for operators defined by this standard.

2 An expression may also be an *unevaluated operand* when it appears in some contexts. An *unevaluated operand* is a expression which is not evaluated in the program[2].

3 Whenever a *glvalue* appears in an expression that expects a *prvalue*, a standard conversion sequence is applied based on the rules in 4.

5.1 Usual Arithmetic Conversions [Expr.conv]

1 Binary operators for arithmetic and enumeration type require that both operands are of a common type. When the types do not match the *usual arithmetic conversions* are applied to yield a common type. When *usual arithmetic conversions* are applied to vector operands they behave as component-wise conversions (4.9). The *usual arithmetic conversions* are:

- If either operand is of scoped enumeration type no conversion is performed, and the expression is ill-formed if the types do not match.
- If either operand is a `vector<T,X>`, vector extension is performed with the following rules:
  - If both vectors are of the same length, no extension is required.
  - If one operand is a vector and the other operand is a scalar, the scalar is extended to a vector via a Splat conversion (4.7).
  - Otherwise, if both operands are vectors of different lengths, the expression is ill-formed.
- If either operand is of type `double` or `vector<double, X>`, the other operator shall be converted to match.
- Otherwise, if either operand is of type `float` or `vector<float, X>`, the other operand shall be converted to match.
- Otherwise, if either operand is of type `half` or `vector<half, X>`, the other operand shall be converted to match.
- Otherwise, integer promotions are performed on each scalar or vector operand following the appropriate scalar or component-wise conversion (4).
  - If both operands are scalar or vector elements of signed or unsigned types, the operand of lesser integer conversion rank shall be converted to the type of the operand with greater rank.
  - Otherwise, if both the operand of unsigned scalar or vector element type is of greater rank than the operand of signed scalar or vector element type, the signed operand is converted to the type of the unsigned operand.
  - Otherwise, if the operand of signed scalar or vector element type is able to represent all values of the operand of unsigned scalar or vector element type, the unsigned operand is converted to the type of the signed operand.
  - Otherwise, both operands are converted to a scalar or vector type of the unsigned integer type corresponding to the type of the operand with signed integer scalar or vector element type.

[1]This will change in the future, but this document assumes current behavior.

[2]The operand to `sizeof(...)` is a good example of an *unevaluated operand*. In the code `sizeof(Foo())`, the call to `Foo()` is never evaluated in the program.

Acronyms

API Application Programming Interface. 13

C C Programming Language. 2

C++ C++ Programming Language. 2

DXC DirectX Shader Compiler. 2

FXC Legacy DirectX Shader Compiler. 2

HLSL High Level Shader Language. 1–3, 5, 9

SIMD Single Instruction Multiple Data. 3

SPMD Single Program Multiple Data. 1, 3, 13

Glossary

DirectX DirectX is the multimedia API introduced with Windows 95.. 2, 3

Dispatch A group of one or more Thread Groups which comprise the largest unit of a shader execution. Also called: grid, compute space or index space.. 1, 3

ISO/IEC 14882:2020 ISO C++ standard. 2, 3, 5, 9

ISO/IEC 9899:2018 ISO C standard. 2, 3, 5

Lane The computation performed on a single element as described in the SPMD program. Also called: thread.. 1, 3, 13

Quad A group of four Lanes which form a cluster of adjacent computations in the data topology. Also called: quad-group or quad-wave. . 1, 3

Shader Model Versioned hardware description included as part of the DirectX specification, which is used for code generation to a common set of features across a range of vendors.. 2, 3

Thread Group A group of Lanes which may be subdivided into one or more Waves and comprise a larger computation. Also known as: group, workgroup, block or thread block.. 1, 3, 13

Wave A group of Lanes which execute together. The number of Lanes in a Wave varies by hardware implementation. Also called: warp, SIMD-group, subgroup, or wavefront.. 1, 3, 13