

High-Level Shader Language Specification

Working Draft

December 1, 2023

Contents

1	Introduction	3
1.1	Scope	3
1.2	Normative References	3
1.3	Terms and definitions	3
1.4	Common Definitions	3
1.4.1	Correct Data	4
1.4.2	Diagnostic Message	4
1.4.3	Ill-formed Program	4
1.4.4	Implementation-defined Behavior	4
1.4.5	Implementation Limits	4
1.4.6	Undefined Behavior	4
1.4.7	Unspecified Behavior	4
1.4.8	Well-formed Program	4
1.4.9	Runtime Implementation	4
1.5	Runtime Targeting	4
1.6	Single Program Multiple Data Programming Model	5
1.6.1	SPMD Terminology	5
1.6.2	SPMD Execution Model	6
1.7	HLSL Memory Models	6
1.7.1	Memory Spaces	6
2	Lexical Conventions	8
2.1	Unit of Translation	8
2.2	Phases of Translation	8
2.3	Character Sets	8
2.4	Preprocessing Tokens	9
2.5	Tokens	10
2.6	Comments	10
2.7	Header Names	10
2.8	Preprocessing numbers	10
2.9	Literals	11
2.9.1	Literal Classifications	11
2.9.2	Vector Literals	11
3	Basic Concepts	12
3.1	Lvalues and rvalues	12
4	Standard Conversions	13
4.1	Lvalue-to-rvalue conversion	13
4.2	Array-to-pointer conversion	13
4.3	Integral conversion	13
4.4	Floating point conversion	13
4.5	Floating point-integral conversion	14
4.6	Boolean conversion	14
4.7	Vector splat conversion	14
4.8	Vector and matrix truncation conversion	14
4.9	Component-wise conversions	14
4.10	Qualification conversion	14
5	Expressions	15
5.1	Usual Arithmetic Conversions	15

5.2	Primary Expressions	16
5.2.1	Literals	16
5.2.2	This	16
5.2.3	Parenthesis	16
6	Declarations	17
6.1	Attributes	17
6.1.1	Entry Attributes	17
7	Runtime	18

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 7.

1.4 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.

1.4.1 Correct Data [Intro.Defs.CorrectData]

1 Data is correct if it represents values that have specified or unspecified but not undefined behavior for all the operations in which it is used. Data that is the result of undefined behavior is not correct, and may be treated as undefined.

1.4.2 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.

1.4.3 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.

1.4.4 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.

1.4.5 Implementation Limits [Intro.Defs.ImpLimits]

1 Restrictions imposed upon programs by the implementation of either the compiler or runtime environment. The compiler may seek to surface runtime-imposed limits to the user for improved user experience.

1.4.6 Undefined Behavior [Intro.Defs.Undefined]

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

1.4.7 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.

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

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

1.4.9 Runtime Implementation [Intro.Defs.Runtime]

1 A runtime implementation refers to a full-stack implementation of a software runtime that can facilitate the execution of HLSL programs. This broad definition includes libraries and device driver implementations. The HLSL specification does not distinguish between the user-facing programming interfaces and the vendor-specific backing implementation.

1.5 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.

1.6 Single Program Multiple Data 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.

1.6.1 SPMD Terminology [Intro.Model.Terms]

Host and Device [Intro.Model.Terms.HostDevice]

1 HLSL is a data-parallel programming language designed for programming auxiliary processors in a larger system. In this context the *host* refers to the primary processing unit that runs the application which in turn uses a runtime to execute HLSL programs on a supported *device*. There is no strict requirement that the host and device be different physical hardware, although they commonly are. The separation of host and device in this specification is useful for defining the execution and memory model as well as specific semantics of language constructs.

Lane [Intro.Model.Terms.Lane]

2 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 may execute instructions in lockstep because each instruction may be a SIMD instruction computing the results for multiple Lanes simultaneously, or synchronizing execution across multiple Lanes or Waves. A Lane has an associated *lane state* which denotes the execution status of the lane (1.6.1).

Wave [Intro.Model.Terms.Wave]

3 A grouping of Lanes for execution is called a Wave. Wave sizes vary by hardware architecture. Some hardware implementations support multiple wave sizes. Wave sizes must be powers of two. There is no overall minimum wave size requirement, although some language features do have minimum lane size requirements. HLSL is explicitly designed to run on hardware with arbitrary Wave sizes. Hardware architectures may implement Waves as Single Instruction Multiple Thread (SIMT) where each thread executes instructions in lock-step. This is not a requirement of the model. Some constructs in HLSL require synchronized execution. Such constructs will explicitly specify that requirement.

Quad [Intro.Model.Terms.Quad]

4 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. Quad operations require four active Lanes.

Thread Group [Intro.Model.Terms.Group]

5 A grouping of Lanes executing the same shader to produce a combined result is called a Thread Group. Thread Groups are independent of SIMD hardware specifications. The dimensions of a Thread Group are defined in three dimensions. The maximum extent along each dimension of a Thread Group, and the total size of a Thread Group are implementation limits defined by the runtime and enforced by the compiler. If a Thread Group's size is not a whole multiple of the hardware Wave size, the unused hardware Lanes are implicitly inactive.

6 If a Thread Group size is smaller than the Wave size, or if the Thread Group size is not an even multiple of the Wave size, the remaining Lane are *inactive* Lanes.

Dispatch [Intro.Model.Terms.Dispatch]

7 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.

Lane States**[Intro.Model.Terms.LaneState]**

8 Lanes may be in three primary states: *active*, *helper*, *inactive*, and *predicated off*.

9 An *active* Lane is enabled to perform computations and produce output results based on the initial launch conditions and program control flow.

10 A *helper* Lane is a lane which would not be executed by the initial launch conditions except that its computations are required for adjacent pixel operations in pixel fragment shaders. A *helper* Lane will execute all computations but will not perform writes to buffers, and any outputs it produces are discarded. *Helper* lanes may be required for Lane-cooperative operations to execute correctly.

11 A *inactive* Lane is a lane that is not executed by the initial launch conditions. This occurs when there is not sufficient inputs to fill the all Lanes in the Wave.

12 A *predicated off* Lane is a lane that is not being executed due to program control flow. A Lane may be *predicated off* when control flow for the Lanes in a Wave diverge and one or more lanes are temporarily not executing.

1.6.2 SPMD Execution Model**[Intro.Model.Exec]**

1 A runtime implementation shall provide an implementation-defined mechanism for defining a Dispatch. A runtime shall manage hardware resources and schedule execution to conform to the behaviors defined in this specification in an implementation-defined way. A runtime implementation may sort the Thread Groups of a Dispatch into Waves in an implementation-defined way. During execution no guarantees are made that all Lanes in a Wave are actively executing.

1.7 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.

1.7.1 Memory Spaces**[Intro.Memory.Spaces]**

1 HLSL programs manipulate data stored in five distinct memory spaces: thread, threadgroup, device, constant and host.

Thread Memory**[Intro.Memory.Spaces.Thread]**

2 Thread memory is local to the Lane. It is the default memory space used to store local variables. Thread memory cannot be directly read from other threads without the use of intrinsics to synchronize execution and memory.

Thread Group Memory**[Intro.Memory.Spaces.Group]**

3 Thread Group memory is denoted in HLSL with the `groupshared` keyword. The underlying memory for any declaration annotated with `groupshared` is shared across an entire Thread Group. Reads and writes to Thread Group Memory, may occur in any order except as restricted by synchronization intrinsics or other memory annotations.

Device Memory**[Intro.Memory.Spaces.Device]**

4 Device memory is memory available to all Lanes executing on the device. This memory may be read or written to by multiple Thread Groups that are executing concurrently. Reads and writes to device memory may occur in any order except as restricted by synchronization intrinsics or other memory annotations. Some device memory may be visible to the host. Device memory that is visible to the host may have additional synchronization concerns for host visibility.

Constant Memory**[Intro.Memory.Spaces.Constant]**

5 Constant memory is similar to device memory in that it is available to all Lanes executing on the device. Constant memory is read-only, and an implementation can assume that constant memory is immutable and cannot change during execution.

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 (7).

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

¹The 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.

2.8 Preprocessing numbers

[Lex.PPNumber]

pp-number:

```

digit
. digit
pp-number ' digit
pp-number ' non-digit
pp-number e sign
pp-number E sign
pp-number p sign
pp-number P sign
pp-number .

```

- 1 Preprocessing numbers begin with a digit or period (.), and may be followed by valid identifier characters and floating point literal suffixes (e+, e-, E+, E-, p+, p-, P+, and P-). Preprocessing number tokens lexically include all *integer-literal* and *floating-point-literal* tokens.
- 2 Preprocessing numbers do not have types or values. Types and values are assigned to *integer-literal*, *floating-point-literal*, and *vector-literal* tokens on successful conversion from preprocessing numbers.
- 3 A preprocessing number cannot end in a period (.) if the immediate next token is a *scalar-element-sequence* (2.9.2). In this situation the *pp-number* token is truncated to end before the period².

2.9 Literals

[Lex.Literals]

2.9.1 Literal Classifications

[Lex.Literal.Kinds]

```

literal:
  integer-literal
  character-literal
  floating-point-literal
  string-literal
  boolean-literal
  vector-literal

```

2.9.2 Vector Literals

[Lex.Literal.Vector]

```

vector-literal:
  integer-literal . scalar-element-sequence
  floating-point-literal . scalar-element-sequence

scalar-element-sequence:
  scalar-element-sequence-x
  scalar-element-sequence-r

scalar-element-sequence-x:
  x
  scalar-element-sequence-x x

scalar-element-sequence-r:
  r
  scalar-element-sequence-r r

```

- 1 A *vector-literal* is an *integer-literal* or *floating-point* literal followed by a period (.) and a *scalar-element-sequence*.
- 2 A *scalar-element-sequence* is a *vector-swizzle-sequence* where only the first vector element accessor is valid (x or r). A *scalar-element-sequence* is equivalent to a vector splat conversion performed on the *integer-literal* or *floating-point-literal* value (4.7).

²This grammar formulation is not context-free and requires an LL(2) parser.

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 conversion¹.
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 conversion².
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.

¹This differs from C++ with the addition of vector splat and truncation casting and flat conversions.

²C++ 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 overloaded¹. 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 program².
- 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 operand 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.

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

²The 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.

5.2 Primary Expressions

[Expr.Primary]

primary-expression:
literal
this
(expression)
id-expression

5.2.1 Literals

[Expr.Primary.Literal]

1 The type of a *literal* is determined based on the grammar forms specified in 2.9.1.

5.2.2 This

[Expr.Primary.This]

1 The keyword `this` names a reference to the implicit object of non-static member functions. The `this` parameter is always a *prvalue* of non-*cv-qualifiedtype*.³

2 A `this` expression shall not appear outside the declaration of a non-static member function.

5.2.3 Parenthesis

[Expr.Primary.Paren]

1 An expression (*E*) enclosed in parenthesis has the same type, result and value category as *E* without the enclosing parenthesis. A parenthesized expression may be used in the same contexts with the same meaning as the same non-parenthesized expression.

³HLSL Specs Proposal 0007 proposes adopting C++-like syntax and semantics for *cv-qualified this* references.

6 Declarations

[Decl]

6.1 Attributes

[Decl.Attr]

6.1.1 Entry Attributes

[Decl.Attr.Entry]

7 Runtime

[Runtime]

Acronyms

API Application Programming Interface. 20

C C Programming Language. 3

C++ C++ Programming Language. 3

DXC DirectX Shader Compiler. 3

FXC Legacy DirectX Shader Compiler. 3

HLSL High Level Shader Language. 1, 3–8, 13

SIMD Single Instruction Multiple Data. 5

SIMT Single Instruction Multiple Thread. 5

SPMD Single Program Multiple Data. 1, 5, 6, 20

Glossary

DirectX DirectX is the multimedia API introduced with Windows 95.. 3, 4, 6

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.. 5, 6

ISO/IEC 14882:2020 ISO C++ standard. 3, 8, 13

ISO/IEC 9899:2018 ISO C standard. 3, 8

Lane The computation performed on a single element as described in the SPMD program. Also called: thread.. 5–7, 20

Quad A group of four Lanes which form a cluster of adjacent computations in the data topology. Also called: quad-group or quad-wave. . 5

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.. 4, 6

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.. 5, 6, 20

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.. 5, 6, 20