# High-Level Shader Language Specification

Working Draft

September 7, 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 1.7.1.

## 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 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.

### 1.4.2 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.

### 1.4.3 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.

### 1.4.4 Implementation Limits [Intro.Defs.ImpLimits]

**1** Restrictions imposed upon programs by the implementation.

### 1.4.5 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.

### 1.4.6 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.

### 1.4.7 Well-formed Program [Intro.Defs.WellFormed]

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

### 1.4.8 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 execute instructions in lock step because each instruction is a SIMD instruction computing the results for multiple Lanes simultaneously.

**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. 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. 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 described in the `numthreads` attribute on the entry function. (FIXME: Add reference to attribute)

**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 dimensions of a Thread Group are defined by the runtime.

**Dispatch** **[Intro.Model.Terms.Dispatch]**

**6** 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.

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

**1** A runtime implementation will provide an implementation-defined mechanism for defining a Dispatch. In an implementation-defined way a runtime will manage hardware resources and schedule execution to conform to the behaviors defined in this specification. 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.

#### 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. Some device memory may be visible to the host. Device memory that is visible to the host may have additional synchronization concerns.

#### 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.

#### Host Memory [Intro.Memory.Spaces.Host]

**6** Host memory is memory that is directly addressable by the host. Reads and writes to host memory may occur in any order except as restricted by synchronization intrinsics.

# Acronyms

| | |
|---|---|
| **API** | Application Programming Interface. 7 |
| **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. 1–5 |
| **SIMD** | Single Instruction Multiple Data. 3, 4 |
| **SIMT** | Single Instruction Multiple Thread. 4 |
| **SPMD** | Single Program Multiple Data. 1, 3, 4, 7 |

# Glossary

**DirectX** DirectX is the multimedia API introduced with Windows 95.. 2–4

**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.. 4

**ISO/IEC 14882:2020** ISO C++ standard. 2

**ISO/IEC 9899:2018** ISO C standard. 2

**Lane** The computation performed on a single element as described in the SPMD program. Also called: thread.. 4, 5, 7

**Quad** A group of four Lanes which form a cluster of adjacent computations in the data topology. Also called: quad-group or quad-wave. . 4

**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.. 3, 4

**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.. 4, 5, 7

**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.. 4, 7