NVIDIA Material Definition Language 1.2

Language Specification

Document version 1.2.2
April 28, 2015

Document build number 242503

LICENSE AGREEMENT

License Agreement for NVIDIA MDL Specification

IMPORTANT NOTICE – READ CAREFULLY: This License Agreement ("License") for the NVIDIA MDL Specification ("the Specification"), is the LICENSE which governs use of the Specification of NVIDIA Corporation and its subsidiaries ("NVIDIA") as set out below. By copying, or otherwise using the Specification, You (as defined below) agree to be bound by the terms of this LICENSE. If You do not agree to the terms of this LICENSE, do not copy or use the Specification.

RECITALS

This license permits you to use the Specification, without modification for the purposes of reading, writing and processing of content written in the language as described in the Specification, such content may include, without limitation, applications that author MDL content, edit MDL content including material parameter editors and applications using MDL content for rendering.

1. DEFINITIONS.
   Licensee. "Licensee," "You," or "Your" shall mean the entity or individual that uses the Specification.
2. LICENSE GRANT.
   2.1. NVIDIA hereby grants you the right, without charge, on a perpetual, non- exclusive and worldwide basis, to utilize the Specification for the purpose of developing, making, having made, using, marketing, importing, offering to sell or license, and selling or licensing, and to otherwise distribute, products complying with the Specification, in all cases subject to the conditions set forth in this Agreement and any relevant patent (save as set out below) and other intellectual property rights of third parties (which may include NVIDIA). This license grant does not include the right to sublicense, modify or create derivatives of the Specification. For the avoidance of doubt, products implementing this Specification are not deemed to be derivative works of the Specification.
   2.2. NVIDIA may have patents and/or patent applications that are necessary for you to license in order to make, sell, or distribute software programs that are based on the Specification ("Licensed Implementations").
   2.3. Except as provided below, NVIDIA hereby grants you a royalty-free license under NVIDIA's Necessary Claims to make, use, sell, offer to sell, import, and otherwise distribute Licensed Implementations. The term "Necessary Claims" means claims of a patent or patent application (including continuations, continuations-in-part, or reissues) that are owned or controlled by NVIDIA and that are necessarily infringed by the Licensed Implementation. A claim is necessarily infringed only when it is not possible to avoid infringing when implementing the Specification. Notwithstanding the foregoing, "Necessary Claims" do not include any claims that would require a payment of royalties by NVIDIA to unaffiliated third parties.
3. NO WARRANTIES.
   3.1. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, THE SPECIFICATION IS PROVIDED "AS IS" AND NVIDIA AND ITS SUPPLIERS DISCLAIM ALL WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSEACCURACY, COMPLETENESS AND NONINFRINGEMENT OF THIRD PARTY RIGHTS.
   3.2. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL NVIDIA OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THE SPECIFICATION, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
4. NO SUPPORT.
   NVIDIA has no obligation to support or to provide any updates of the SPECIFICATION.
5. FEEDBACK.
   In the event Licensee contacts NVIDIA to request Feedback (as defined below) on how to design, implement, or optimize Licensee's product for use with the SPECIFICATION, the following terms and conditions apply to the Feedback:
   5.1. Exchange of Feedback. Both parties agree that neither party has an obligation to give the other party any suggestions, comments or other feedback, whether orally or otherwise ("Feedback"), relating to (i) the SPECIFICATION; (ii) Licensee's products; (iii) Licensee's use of the SPECIFICATION; or (iv) optimization of Licensee's product with the SPECIFICATION. In the event either party provides Feedback to the other party, the party receiving the Feedback may use and include any Feedback that the other party voluntarily provides to improve the (i) SPECIFICATION or other related NVIDIA technologies, respectively for the benefit of NVIDIA; or (ii) Licensee's product or other related Licensee technologies, respectively for the benefit of Licensee. Accordingly, if either party provides Feedback to the other party, both parties agree that the other party and its respective licensees may freely use, reproduce, license, distribute, and otherwise commercialize the Feedback in the (i) SPECIFICATION or other related technologies; or (ii) Licensee's products or other related technologies, respectively, without the payment of any royalties or fees.

5.2. Residual Rights. Licensee agrees that NVIDIA shall be free to use any general knowledge, skills and experience, (including, but not limited to, ideas, concepts, know- how, or techniques) ("Residuals"), contained in the (i) Feedback provided by Licensee to NVIDIA; (ii) Licensee's products shared or disclosed to NVIDIA in connection with the Feedback; or (c) Licensee's confidential information voluntarily provided to NVIDIA in connection with the Feedback, which are retained in the memories of NVIDIA's employees, agents, or contractors who have had access to such (i) Feedback provided by Licensee to NVIDIA; (ii) Licensee's products; or (c) Licensee's confidential information voluntarily provided to NVIDIA, in connection with the Feedback. Subject to the terms and conditions of this Agreement, NVIDIA's employees, agents, or contractors shall not be prevented from using Residuals as part of such employee's, agent's or contractor's general knowledge, skills, experience, talent, and/or expertise. NVIDIA shall not have any obligation to limit or restrict the assignment of such employees, agents or contractors or to pay royalties for any work resulting from the use of Residuals. FEEDBACK FROM EITHER PARTY IS PROVIDED FOR THE OTHER PARTY'S USE "AS IS" AND BOTH PARTIES DISCLAIM ALL WARRANTIES, EXPRESS, IMPLIED AND STATUTORY INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. BOTH PARTIES DO NOT REPRESENT OR WARRANT THAT THE FEEDBACK WILL MEET THE OTHER PARTY'S REQUIREMENTS OR THAT THE OPERATION OR IMPLEMENTATION OF THE FEEDBACK WILL BE UNINTERRUPTED OR ERROR-FREE.

6. NO IMPLIED LICENSES.

   Under no circumstances should anything in this Agreement be construed as NVIDIA granting by implication, estoppel or otherwise, (i) a license to any NVIDIA product or technology other than the SPECIFICATION; or (ii) any additional license rights for the SPECIFICATION other than the licenses expressly granted in this Agreement.

7. THIRD PARTY RIGHTS.

   Without limiting the generality of Section 3 above, NVIDIA ASSUMES NO RESPONSIBILITY TO COMPILE, CONFIRM, UPDATE OR MAKE PUBLIC ANY THIRD PARTY ASSERTIONS OF PATENT OR OTHER INTELLECTUAL PROPERTY RIGHTS THAT MIGHT NOW OR IN THE FUTURE BE INFRINGED BY AN IMPLEMENTATION OF THE SPECIFICATION IN ITS CURRENT, OR IN ANY FUTURE FORM. IF ANY SUCH RIGHTS ARE DESCRIBED ON THE SPECIFICATION, NVIDIA TAKES NO POSITION AS TO THE VALIDITY OR INVALIDITY OF SUCH ASSERTIONS, OR THAT ALL SUCH ASSERTIONS THAT HAVE OR MAY BE MADE ARE SO LISTED.

8. TERMINATION OF LICENSE.

   This LICENSE will automatically terminate if Licensee fails to comply with any of the terms and conditions hereof. In such event, Licensee must destroy all copies of the SPECIFICATION and all of its component parts.

9. DEFENSIVE SUSPENSION.

   If Licensee commences or participates in any legal proceeding against NVIDIA, then NVIDIA may, in its sole discretion, suspend or terminate all license grants and any other rights provided under this LICENSE during the pendency of such legal proceedings.

10. ATTRIBUTION.

    Licensee shall, in a manner reasonably acceptable to NVIDIA prominently embed in all products produced in compliance with the Specification a notice stating that such product has been so produced.

11. MISCELLANEOUS.

    11.1. Notices. All notices required under this Agreement shall be in writing, and shall be deemed effective five days from deposit in the mails. Notices and correspondence to either party shall be sent to its address as it appears below.

    General Counsel, NVIDIA Corporation 2701 San Tomas Expressway
    Santa Clara, CA 95050.

    11.2. Export regulations. The Specification, or portions thereof, including technical data, may be subject to U.S. export control laws, including the U.S. Export Administration Act and its associated regulations, and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such regulations and acknowledges that it has the responsibility to obtain all export, re-export, import or other licenses in connection with its use of the Specification or any product complying with the Specification.

Table of Contents

1 Introduction

NVIDIA Material Definition Language (MDL) is a domain-specific language that describes the appearance of scene elements for a rendering process. MDL consists of a declarative component to describe materials as well as a procedural programming language to customize image texture lookups and procedural textures that provide the parametric input to the material declarations.

MDL is dedicated to providing an abstract material description that is well-defined and independent of particular rendering systems. The material description is comprehensive and flexible, yet specifically addresses modern renderers based on the simulation of physically based light transport. The declarative nature of the material description makes it easy for a renderer to fully understand a material, yet — if needed — to simplify and approximate the material to the best of the renderer's capabilities. The procedural programming language enables the aesthetic flexibility required by artists to design new materials.

Although it emphasizes physically plausible materials, MDL supports also traditional computer graphics techniques that are important in conventional rendering and modeling workflows, such as normal mapping and cut-outs.

MDL is designed for modern highly-parallel machine architectures. One important part is the declarative nature of the material description. Another part is its procedural language, which is restricted to the definition of pure functions, with access to the rendering state, that are free of side effects and global dependencies. Material and function evaluations can thus be easily compiled and executed on modern architectures.

1.1 Material building blocks

A material in MDL consists of a set of building blocks that describe how a renderer should compute the appearance of a geometrically described surface. A geometric surface is a mathematical idealization having no thickness that can only serve as a boundary between two volumes. Thus, the geometric definition of a sphere as a surface can instead be interpreted as the boundary of a spherically shaped volume.

MDL defines how light is affected by this boundary — reflected from the surface, refracted through the volume the surface defines, or a combination of both. The surface boundary can also define the extent of a medium through which the light passes and which participates in scattering and absorbing the light.

A geometric surface has no thickness. However, for the purposes of MDL, a surface can also be defined as having a thickness, though infinitesimally small. In MDL, this is said to be a *thin-walled* surface. The thin-walled property permits greater flexibility in defining the appearance of a surface that is not closed, and in which the surface is not the boundary of an object, but an object in itself. Because both sides of a thin-wall surface can be rendered, MDL also allows the two sides to possess different appearance properties.

Depending on the thin-walled property, MDL enables three categorically different materials with respect to how they interpret surfaces:

1. A surface is an interface between volumes. The volumetric properties defined in the material apply to the enclosed volume.
2. A surface represents a thin wall conceptually enclosing an infinitely thin volume with identical materials on both sides. Window glass, for example, can be modeled in this way.
3. A surface represents a thin wall with different materials on both sides, in which case both materials must have equal transmission interactions.

Material properties in MDL also include geometric properties of surfaces, such as cut-outs, displacements, or bump maps, which are commonly not modeled in the geometric description.

1.2 Material libraries and re-use of components

MDL has a well-defined module and package concept allowing for a comprehensive organization of complex material libraries. Together with the namespace concept, this allows for an easy deployment of material libraries from different providers that will still smoothly interoperate with future in-house material developments.

MDL modules contain materials and functions with their related types and constants. The re-use of those elements in MDL is important when building larger material libraries. Functions can be used to encapsulate other functions and change their signatures by changing their names, parameters, and return types and hiding details, such as unexposed parameters and their hidden settings. Materials can be used to encapsulate other materials, providing new names and parameters while, like functions, hiding unnecessary details.

1.3 Related documents

The document *NVIDIA Material Definition Language: Technical Introduction* illustrates the use of MDL through a series of images rendered using MDL materials and related code examples [1].

2 Terms and definitions

2.1 Typographical conventions

The grammar of many language elements is introduced at the beginning of a section using Wirth's extensions of Backus Normal Form. The left-hand side of a production is separated from the right hand side by a colon. Alternatives are separated by a vertical bar. Optional items are enclosed in square brackets. Curly braces indicate that the enclosed item may be repeated zero or more times.

Non-terminal and meta-symbols are given in *italic* font. Terminal symbols except identifiers, typenames, and literals are given in `teletype` font. See also Section 5 for the lexical structure of MDL.

For example:

struct_type_declaration : `struct` *simple_name* [*annotation_block*]
`{` {*struct_field_declarator*} `}` `;`

struct_field_declarator : *type* *simple_name* [`=` *expression*]
[*annotation_block*] `;`

2.2 File paths and resolution in the file system

MDL can reference external files with a string literal containing a *file path*. String literals are defined in Section 5.7.5 and are here restricted to a 7-bit ASCII encoding of printable characters.

A file path consists of a *file name* preceded by a, possibly empty, *directory path* that is separated by a slash '/' from the file name. A directory path consists of a sequence of *directory names* separated by slashes '/', optionally preceded by a single slash '/'. The convention of specifying '.' for the current directory and '..' for the parent directory is not allowed.

File paths beginning with a slash '/' are called *absolute file paths*, while those not beginning with a slash '/' are called *relative file paths*.

An integration of MDL defines an implementation specific sequence of *search paths* with an implementation specific order, which shall be consistent for all MDL file compilations. A search path is a directory path.

A file path uniquely identifies a file in the file system in the context of the sequence of search paths. Absolute and relative file paths are handled differently.

If the file path is an absolute file path, the search paths are searched in their respective order. If the file named by the search path concatenated with the file path exists, the search stops with this file as the result. If the file path has a non-empty directory path and the concatenation of the search path with this directory path exists, the search stops with the file not found. In all other cases, the search continues with the next search path in order.

If the file path is a relative path, it is first located relative to the MDL file that is currently processed and if that fails it is treated like an absolute file path.

Note 1: Above rules imply that if a directory path exists twice, under different search paths, that only files in the first search path will be found and not in the second. This is intentional in conjunction with the module system explained in Section 15, where modules are grouped in packages that correspond to

directories. Packages are better shielded from each other and multiple installations of partially different versions of the same package cannot influence each other since only the first version is found.

Note 2: Above rules are furthermore a prerequisite for the property of MDL that fully qualified names uniquely identify elements in the language independent of any context.

Note 3: MDL module names and package names have a correspondence to file names and directory names explained in Section 15. Consequently, those file and directory names are restricted to legal MDL identifiers as defined in Section 5.5.

2.3 Texture files

A texture file can be referenced in MDL using a file path, as explained in Section 2.2. The file name shall end in one of the following file extensions, separated by a dot '.' from the base name, and the file shall be in the corresponding format:

Extension	File format
`png`	ISO/IEC 15948:2004 - Information technology — Computer graphics and image processing — Portable Network Graphics (PNG): Functional specification. Also as RFC 2083, PNG (Portable Network Graphics) Specification, Version 1.0, (March 1997).
`exr`	OpenEXR `http://www.openexr.com/`
`jpg` `jpeg`	ISO/IEC 10918-1:1994, Digital Compression and Coding of Continuous-Tone Still Images. ISO/IEC CD 10918-5, Information technology – Digital compression and coding of continuous-tone still images: JPEG File Interchange Format (JFIF).
`ptx`	PTEX `http://ptex.us/`

Note: MDL integrations can support more texture file formats in the scene description and pass those textures into function or material parameters of suitable texture types.

2.4 Light profile files

A light profile file can be referenced in MDL using a file path, as explained in Section 2.2. The file name shall end in one of the following file extensions, separated by a dot '.' from the base name, and the file shall be in the corresponding format:

Extension	File format
`ies`	IES LM-63-02 Standard File Format for Electronic Transfer of Photometric Data and Related Information, Illuminating Engineering Society.

Note: MDL integrations can support more light profile file formats in the scene description and pass those into function or material parameters of a light profile type.

2.5 BSDF measurement data files

A bidirectional scattering distribution function (BSDF) measurement data file can be referenced in MDL using a file path, as explained in Section 2.2. The file name shall end in one of the following file extensions, separated by a dot '.' from the base name, and the file shall be in the corresponding format:

Extension	File format
`mbsdf`	MBSDF file format documented in Appendix B, Section 23.

Note: MDL integrations can support more BSDF measurement data file formats in the scene description and pass those into function or material parameters of a BSDF measurement type.

2.6 Quantities and units

Light transport simulation is performed in radiometric units. Material or function parameters may accept values in photometric units, but these values will eventually be converted to radiometric units before interacting with the renderer.

One of the most fundamental quantities in light transport is *radiance* leaving from (*exitant*) or arriving at (*incident*) a point x in direction ω. It is usually denoted by

$$L(x,\omega) \quad \left[\frac{W}{m^2 sr}\right]$$

and measured in watts per square meter per steradian. Note that exitant radiance usually differs strongly from incident radiance, because the former includes interaction of light with the local surface, while the latter does not.

Spectral rendering uses *spectral radiance* in $\left[W \cdot m^{-2} \cdot sr^{-1} \cdot nm^{-1}\right]$, which additionally depends on the wavelength in nanometers. The same applies to all following quantities. Radiance is obtained from spectral radiance by integrating over an interval of wavelengths. The integrand is commonly weighted by some kind of spectral response function.

The following sections follow conventional notation and omit the explicit reference to wavelength dependence for brevity.

Integrating incoming radiance L_i over a set of unit directions $\Omega \subseteq S^2$ around a point leads to *irradiance*

$$E(x) = \int_\Omega L_i(x,-\omega) d\bar{\sigma}(\omega) \quad \left[\frac{W}{m^2}\right],$$

where θ is the angle between the normal and ω, and

$$d\bar{\sigma}(\omega) = \begin{cases} d\sigma(\omega) & \textit{in the volume}, \\ d\sigma^{\perp}(\omega) = |\cos\theta|\, d\sigma(\omega) & \textit{on the surface}. \end{cases}$$

The measure $d\bar{\sigma}$ is simply a short-hand for the solid angle measure in the volume and the projected solid angle measure on the surface.

The similar integral of exitant radiance L_o, the radiance leaving a surface, is referred to as *radiant exitance* $M(x)$. This quantity is especially useful when describing light sources.

Finally, integrating irradiance (or radiant exitance) over surface area results in *power*

$$\Phi = \int_A E(x) dx \quad [W].$$

The unit used for distances depends on the context and can be *meters*, world space *scene units*, or implicit through some coordinate space transformation. Distances for the volume absorption coefficient and

the volume scattering coefficient of the MDL material model in Section 13.3 are in meters in world space. The radius of the rounded corner state function in Section 19.3 is in meters in world space. Distances in internal space or object space can be transformed into scene units in world space using the coordinate space transformations in Section 19.2. Scene units can be multiplied with the result of the `::state::meters_per_scene_unit()` state function in Section 19.2 to convert them to meters. The `::state::scene_unit_per_meters()` state function returns the reciprocal value.

Angles are specified in radians in the range from `0` to 2π, for example, as arguments to the trigonometric standard library functions. Rotations are specified in the range from `0` to 1, where 1 means a full rotation by an angle of 2π. Rotations are used as function or material parameters that can be textured, for example, the `rotation` parameter of the `simple_glossy_bsdf` elemental BSDF of Section 21.1.3.

3 Runtime model

MDL modules offer functions and materials for runtime execution. Both can have parameters that are provided with values at runtime. Besides parameters, state functions are used to communicate rendering state to the functions and materials defined in MDL.

3.1 Functions

Functions in MDL are conventional functions with input parameters and a return value. In addition, functions have read-only access to a runtime state through state functions. The runtime state provides standardized access to important values of the rendering context, such as the current position or texture coordinates.

An application integrating MDL can call functions at runtime with concrete arguments for the parameters and a runtime state. The function computes a return value to be used by the application.

More specifically, functions in MDL are pure in the sense of being side-effect free and are state-less in the sense that they always compute the same return value for the same argument values and runtime state.

Functions can be *overloaded*, in which the same name is used for more than one function, but where the functions differ in their parameter lists. Overload call resolution is explained in Section 12.4.

A relevant aspect of the MDL type system for the runtime integration is the handling of size-deferred arrays (see Section 7). Depending on the capabilities of the MDL integration, size-deferred arrays may not exist at runtime and overloaded functions are provided instead for each concrete array size used. This may imply that calling a function with a size-deferred array parameter requires a re-compilation of this function for the specific array size in the call. Calling the function again with a different array value, but of equal array size, should not require a re-compilation. Otherwise, the MDL design does not imply that a re-compilation would be necessary for other parameter types.

3.2 Materials

Materials in MDL consist of predefined building blocks with parameters that can be combined in flexible ways. The building blocks, their parameters and combinations are explained in the material model section (see Section 13).

The material building blocks have input parameters. These parameters can be set to literal values and the return value of function calls, including state function calls. The parameters to the function calls themselves can also be set to these types of values.

A material is instantiated when its input parameters are bound to specific values. This happens typically when a material is assigned to an object in the scene description. At instantiation, material input parameters can be set to literal values or the return value of function calls, including state function calls.

An application integrating MDL can instantiate materials and inspect material instances. Material inspection allows the application to understand the complete structure of how the material building blocks are combined and how all parameters of these building blocks are set. If a parameter is set to the return value of a function call, the application can retrieve the arguments provided to that function and call the function with the proper runtime state. The runtime state depends on the material building block where the function call result is needed.

Although material definitions cannot be overloaded in MDL, an MDL integration may choose to replace a material with parameters of size-deferred array types by a set of materials with concrete array sizes similar

to how it may handle functions with size-deferred array parameter types.

3.3 Distance unit conversion between scene and MDL

Distances in MDL may expect their value in meters, world space scene units, or others defined by transformations. In particular the integration of scene units into MDL requires the runtime to know the ratio between a scene unit and a meter, for example, from an applications setting or a rendering option in the scene.

4 Overview

mdl : *mdl_version* {*import*} {[`export`] *global_declaration*}

mdl_version : `mdl` *floating_literal* `;`

import : `import` *qualified_import* {`,` *qualified_import*} `;`
| [`export`] `using` *qualified_name*
`import` (`*` | *simple_name* {`,` *simple_name*}) `;`

qualified_import : [`::`] *simple_name* {`::` *simple_name*} [`::` `*`]

qualified_name : [`::`] *simple_name* {`::` *simple_name*}

simple_name : *IDENT*

global_declaration : *annotation_declaration*
| *constant_declaration*
| *type_declaration*
| *function_declaration*

A compilation unit in Material Definition Language (MDL) is a module. A module consists of a mandatory MDL version declaration, a sequence of import declarations, and a sequence of global declarations. Imported elements as well as those from the global declarations can be exported to be used outside of the module itself in other modules or by a renderer.

4.1 MDL version declaration

Each MDL module starts with a MDL version declaration, only preceded by optional white space or comments. The version declaration identifies the module as written in the corresponding MDL version. The version itself consists of the major version number followed by a dot and the minor version number.

The following example illustrates how a MDL module can start that follows the version of this specification document:

```
mdl 1.2;
```

4.2 Import declarations

In MDL, all identifiers and all typenames live in modules with the exception of the types available as built-in reserved words. A module defines a namespace and shields identifiers from naming conflicts. Modules can be used as such or they can be organized in packages, which define a namespace as well and nest the module namespace or sub-package namespace within their own. Modules and packages are explained in detail in Section 15.

Declarations inside a module need to be marked for export before they can be used outside of the module, and other modules need to import those declarations before they can be used.

Import declarations can import individual declarations or all declarations from other modules. Depending on the particular form of the import declarations used, the imported declarations can only be referred to

by their qualified identifiers or by their unqualified identifiers. A qualified identifier includes the module and package names as namespace prefixes separated by the scope operator '::'. See Section 15.1 for the details of import declarations.

Modules implemented using different versions of MDL can be freely mixed with one restriction: Declarations that are not legal in a modules version of MDL cannot be imported from another module even though the declaration would be legal in that modules version of MDL. In other words, a module can only see declarations of other modules that are legal in its version of MDL. Details are explained in Section 15.4.

MDL's import mechanism does not offer any name-conflict resolution mechanisms, i.e., an identifier or type can only be imported in the unqualified form from one module. The purpose of this policy is to have a well-defined module system that enables packaging and re-use of material libraries by independent providers.

4.3 Global declarations

Global declarations can be any of:

- global constants, see Section 6.6,
- type declarations in the form of structure type declarations, see Section 8, enumeration type declarations, see Section 9, or typedef declarations, see Section 10,
- function declarations, Section 12,
- material definitions, Section 13, that are syntactically similar to function declarations and represented this way in the grammar, and
- annotation declarations, see Section 14.

Note: MDL does not have global variables (besides global constants) nor global material instances. The instancing of materials with concrete parameter values is left to the integration with the renderer or scene graph representation.

5 Lexical structure

This section describes the lexical structure of MDL.

5.1 Character set

A MDL source file is a sequence of characters from a character set. This set comprises at least the following characters:

1. the 52 uppercase and lowercase alphabetic characters:

   ```
   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

   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
   ```

2. the 10 decimal digits:

   ```
   0 1 2 3 4 5 6 7 8 9
   ```

3. the blank or space character
4. the 32 graphic characters:

Character	Name	Character	Name
!	exclamation point	"	double quote
#	number sign	$	dollar sign
%	percent	&	ampersand
’	single quote	(	left parenthesis
)	right parenthesis	*	asterisk
+	plus	,	comma
-	hyphen or minus	.	period
/	slash	:	colon
;	semicolon	<	less than
=	equal	>	greater than
?	question mark	@	at symbol
[	left bracket	\	backslash
]	right bracket	^	circumflex
_	underscore	‘	backquote
{	left brace	|	vertical bar
}	right brace	~	tilde

There must also be some way of dividing the source program into lines, typically a newline character or a sequence of newline and carriage return characters. Line endings are significant in delimiting preprocessor directives and one type of comments (see below).

The blank (space) character, tabulators, line endings, and comments (see below) are collective known as *whitespace*. Beyond separating tokens (see below) and the significance of line endings in preprocessor directives, whitespace is ignored.

5.2 Comments

There are two kinds of comments:

- A comment introduced by `//` extends to the end of the line.
- A comment introduced by `/*` extends to the first occurrence of `*/`.

Occurrence of `//` or `/*` inside a string literal (see below) does not introduce a comment.

5.3 Tokens

The characters making up a MDL program are collected into lexical tokens according to the rules presented in the following sections. There are six classes of tokens: operators, separators, identifiers, typenames, reserved words, and literals.

The compiler always uses the longest possible sequence of characters when reading from left to right to form a token, even if that does not result in a legal MDL program. For example, the sequence of characters "`a--b`" is interpreted as the tokens "`a`", "`--`", and "`b`", which is not a legal MDL expression, even though the sequence of tokens "`a`", "`-`", "`-`", "`b`" might constitute a legal expression.

5.4 Operators and separators

These are the operators of MDL grouped by precedence, from highest to lowest:

Operation Name	Operator Expression
scope resolution	*identifier*::*identifier*
scope resolution	*identifier*::*typename*
global scope resolution	::*qualified_name*
global scope resolution	::*qualified_type*
member selection	*expression*.*identifier*
subscripting	*expression*[*expression*]
function call	*qualified_name*(*expression*)
value construction	*qualified_type*(*expression*)
postfix increment	*lvalue*++
postfix decrement	*lvalue*--
prefix increment	++*lvalue*
prefix decrement	--*lvalue*
logical not	!*expression*
unary minus	-*expression*
unary plus	+*expression*
bitwise complement	~ *expression*
multiply	*expression* * *expression*
divide	*expression* / *expression*
modulo	*expression* % *expression*
add	*expression* + *expression*
subtract	*expression* - *expression*
left-shift	*expression* << *expression*
signed right-shift	*expression* >> *expression*
unsigned right-shift	*expression* >>> *expression*
less than	*expression* < *expression*
less than or equal	*expression* <= *expression*
greater than	*expression* > *expression*
greater than or equal	*expression* >= *expression*
equal	*expression* == *expression*
not equal	*expression* != *expression*
bitwise and	*expression* & *expression*
bitwise xor	*expression* ^ *expression*
bitwise or	*expression* | *expression*
logical and	*expression* && *expression*
logical or	*expression* || *expression*
conditional expression	*expression* ? *expression* : *expression*
simple assignment	*lvalue* = *expression*
multiply and assign	*lvalue* *= *expression*
divide and assign	*lvalue* /= *expression*
modulo and assign	*lvalue* %= *expression*
add and assign	*lvalue* += *expression*
subtract and assign	*lvalue* -= *expression*
left-shift and assign	*expression* <<= *expression*
signed right-shift and assign	*expression* >>= *expression*
unsigned right-shift and assign	*expression* >>>= *expression*
bitwise and and assign	*lvalue* &= *expression*
bitwise xor and assign	*lvalue* ^= *expression*
bitwise or and assign	*lvalue* |= *expression*
sequencing	*expression* , *expression*

Unary operators and assignment operators are right-associative; all others are left-associative.

The separators of MDL are "`{`", "`}`", "`:`", and "`;`".

5.5 Identifiers and typenames

An *identifier* is an alphabetic character followed by a possibly empty sequence of alphabetic characters, decimal digits, and underscores, that is neither a typename nor a reserved word (see below). Declarations in the MDL syntax expect an identifier that may actually be a typename in case the declaration shadows this typename, such as a typename from a different scope.

A *typename* has the same lexical structure as an identifier, but is the name of a built-in type, a material name, a BSDF, EDF or VDF class name, or a type defined by the user with a structure, enumeration, or typedef declaration.

5.6 Reserved words

These are the reserved words of MDL that can only be used in the way defined in this document:

`annotation`	`double2x2`	`float`	`in`	`return`
`bool`	`double2x3`	`float2`	`int`	`string`
`bool2`	`double3`	`float2x2`	`int2`	`struct`
`bool3`	`double3x2`	`float2x3`	`int3`	`switch`
`bool4`	`double3x3`	`float3`	`int4`	`texture_2d`
`break`	`double3x4`	`float3x2`	`intensity_mode`	`texture_3d`
`bsdf`	`double4`	`float3x3`	`intensity_power`	`texture_cube`
`bsdf_measurement`	`double4x3`	`float3x4`	`intensity_radiant_exitance`	`texture_ptex`
`case`	`double4x4`	`float4`	`let`	`true`
`color`	`double4x2`	`float4x3`	`light_profile`	`typedef`
`const`	`double2x4`	`float4x4`	`material`	`uniform`
`continue`	`edf`	`float4x2`	`material_emission`	`using`
`default`	`else`	`float2x4`	`material_geometry`	`varying`
`do`	`enum`	`for`	`material_surface`	`vdf`
`double`	`export`	`if`	`material_volume`	`while`
`double2`	`false`	`import`	`mdl`	

These are additional words of MDL that are reserved for future use or to avoid misleading use. Using them results in an error.

`auto`	`friend`	`half4`	`namespace`	`shader`	`typeid`
`catch`	`goto`	`half4x3`	`native`	`short`	`typename`
`char`	`graph`	`half4x4`	`new`	`signed`	`union`
`class`	`half`	`half4x2`	`operator`	`sizeof`	`unsigned`
`const_cast`	`half2`	`half2x4`	`out`	`static`	`virtual`
`delete`	`half2x2`	`inline`	`phenomenon`	`static_cast`	`void`
`dynamic_cast`	`half2x3`	`inout`	`private`	`technique`	`volatile`
`explicit`	`half3`	`lambda`	`protected`	`template`	`wchar_t`
`extern`	`half3x2`	`long`	`public`	`this`	
`external`	`half3x3`	`module`	`reinterpret_cast`	`throw`	
`foreach`	`half3x4`	`mutable`	`sampler`	`try`	

5.7 Literals

literal_expression	:	*boolean_literal* | *enum_literal* | *integer_literal* | *floating_literal* | *string_literal* {*string_literal*}
boolean_literal	:	`true` | `false`
enum_literal	:	`intensity_radiant_exitance` | `intensity_power`
integer_literal	:	*INTEGER_LITERAL*
floating_literal	:	*FLOATING_LITERAL*
string_literal	:	*STRING_LITERAL*

Literals are a means to directly denote values of simple types.

5.7.1 Boolean literals

The Boolean literals are `true` and `false`.

5.7.2 Enumerator literals

Two enumerator literals, `intensity_radiant_exitance` and `intensity_power`, are reserved words. They are literals of the enumerator type `intensity_mode`.

5.7.3 Integer literals

Integer literals can be given in octal, decimal, or hexadecimal base.

A decimal literal is a non-empty sequence of decimal digits.

An octal literal is the digit zero followed by a non-empty sequence of octal digits (the digits from zero to seven inclusive).

A hexadecimal literal is the digit zero, followed by the character `x` or `X`, followed by a non-empty sequence of the *hexadecimal digits*, defined as the decimal digits and the letters `a`, `b`, `c`, `d`, `e`, `f`, `A`, `B`, `C`, `D`, `E`, and `F`.

5.7.4 Floating-point literals

A floating point literal is a possibly empty sequence of decimal digits, optionally followed by a decimal point (the character period) and a possibly empty sequence of decimal digits, optionally followed by an exponent given by the letter `e` or `E`, an optional sign (`-` or `+`) and a non-empty sequence of decimal digits, optionally followed by a type suffix. Either the decimal point or the exponent need to be present. There has to be at least one digit preceding or following the decimal point. The type suffix can be the letter `f`, `F`, `d`, or `D`.

A floating-point literal without type suffix or with type suffix `f` or `F` is of type `float`. A floating-point literal with type suffix `d` or `D` is of type `double`. Floating-point types are described in Section 6.7 on page 23.

A floating-point literal of type `float` is able to hold a value of type `double`, which is used in case the literal is converted to a double value.

5.7.5 String literals

A string literal is a possibly empty sequence of UTF-8 encoded characters not including a double quote or escape sequences, enclosed in double quotes. A string literal may not include a line ending. A string literal may be further restricted to 7-bit ASCII encoding or a fixed set of choices depending on context. String literals can be concatenated by juxtaposition. An escape sequence is a backslash followed by one of the following escape codes:

Escape Code	Translation	Escape Code	Translation
`a`	alert (e.g., bell)	`t`	horizontal tab
`b`	backspace	`\`	backslash
`f`	form feed	`'`	single quote
`n`	newline	`"`	double quote

6 Variables and data types

type : [*frequency_qualifier*] *array_type*

array_type : *simple_type* [[[*conditional_expression* | < *simple_name* >]]]

simple_type : [: :] *relative_type*

relative_type :
`bool` | `bool2` | `bool3` | `bool4`
| `int` | `int2` | `int3` | `int4`
| `float` | `float2` | `float3` | `float4`
| `float2x2` | `float2x3` | `float2x4`
| `float3x2` | `float3x3` | `float3x4`
| `float4x2` | `float4x3` | `float4x4`
| `double` | `double2` | `double3` | `double4`
| `double2x2` | `double2x3` | `double2x4`
| `double3x2` | `double3x3` | `double3x4`
| `double4x2` | `double4x3` | `double4x4`
| `color` | `string` | `bsdf` | `edf` | `vdf`
| `light_profile` | `bsdf_measurement`
| `material` | `material_emission` | `material_geometry`
| `material_surface` | `material_volume` | `intensity_mode`
| `texture_2d` | `texture_3d` | `texture_cube` | `texture_ptex`
| *IDENT* [: : *relative_type*]

frequency_qualifier : `varying` | `uniform`

MDL defines a collection of built-in data types tailored for the kind of tasks materials need to accomplish. In addition, these types can be used to define custom structures as described on page 42 unless noted otherwise in the description of the individual data types.

Below is a list of built-in MDL data types with a brief description of each type:

- `float`, `double` – A single real number.
- `int` – An integer, including positive and negative numbers as well as zero.
- `bool` – A Boolean value that is either `true` or `false`.
- `float2`, `float3`, `float4`, `double2`, `double3`, `double4` – Vectors with real number elements.
- `int2`, `int3`, `int4` – Integer vectors.
- `bool2`, `bool3`, `bool4` – Boolean vectors.
- `float2x2`, `float2x3`, `float3x2`, `float3x3`, `float4x3`, `float3x4`, `float2x4`, `float4x2`, `float4x4`, `double2x2`, `double2x3`, `double3x2`, `double3x3`, `double4x3`, `double3x4`, `double2x4`, `double4x2`, `double4x4` – Matrices of various dimensions.
- `texture_2d`, `texture_3d`, `texture_cube`, `texture_ptex`, – Texture map samplers.
- `color` – An implementation independent representation of color.
- `string` – A character string.

- `light_profile` – A description of light emittance for real world lights; used for parameters of emission descriptions only.
- `bsdf_measurement` – A description of a measured bidirectional scattering distribution function (BSDF) that can be used with the corresponding elemental distribution function to render the measurement.
- `bsdf`, `edf`, `vdf` – Reference types for a BSDF, EDF, or VDF, respectively; used for parameters of materials only.
- `material`, `material_geometry`, `material_surface`, `material_emission`, `material_volume` – Struct-like types describing a material and its components.
- `intensity_mode` – An enumeration type used for one of the fields in the `material_emission` type to define the units used to control the emission intensity of lights.

Atomic types are keywords in MDL.

6.1 Constructors

A value of a MDL data type can be constructed by calling the type's constructor. The constructor syntax consists of the type name, followed by a comma separated list of arguments contained in parenthesis. For example, the following expression evaluates to a value of type `float3`:

```
float3(0.3, 0.1, 0.7)
```

Constructor calls are analogous to function calls. They allow, in addition to the above argument passing by position a style where arguments are passed by name. In this case, each argument consists of the parameter name separated from the initializing expression by a colon. The exact rules for argument passing are explained in Section 12.1. For example, the following expression evaluates to the same value of type `float3` as the previous example:

```
float3(x: 0.3, y: 0.1, z: 0.7)
```

Types can have several overloaded constructors. Two kinds of constructors exist for all types: the default constructor and the copy constructor. The default constructor takes no parameter. It is used to initialize values if no other initializer is provided. The copy constructor takes a parameter of identical type. It is used when a value is copied.

Constructors are defined specifically for each type in MDL; there are no user-defined constructors in MDL. The individual constructors are documented in the following sections for each corresponding type.

6.2 Variables

variable_declaration : *type variable_declarator*
{ , *variable_declarator*} ;

variable_declarator : *simple_name* [*argument_list* | = *assignment_expression*]
[*annotation_block*]

A type followed by an identifier and semicolon declares a variable of that name and type in MDL. Multiple variables of the same type can be declared by separating them with commas.

Variables can be used in MDL for local variables in functions (Section 12) and as temporary values in `let`-expressions (Section 13.8).

When declaring a variable the type's constructor can be invoked by appending the constructor parameters, enclosed in parenthesis, to the variable name in the declaration. If a variable declaration contains an initializer, it will be treated as if the variable was constructed taking the right-hand side of the assignment statement as the constructor's parameter.

For example, the following three cases are identical, which is to invoke the `float`-type constructor with the literal value `0.0` as a parameter and initialize the variable with the resulting value:

```
float x(0.0);
float y = float(0.0);
float z = 0.0;
```

Variables and parameters do not have to be initialized with an explicit constructor or initializer. In that case, the type's default constructor will be used to initialize the value. Note: In MDL, values are always initialized.

In the following sections, detailed descriptions of each type identify the overloaded constructor versions supported by that type.

6.3 Uniform and varying values and types

A value of a MDL data type can be *uniform* or *varying*. A common source for varying values are varying state functions, see Section 19, or varying function or material parameters, see Section 12.2 and 13.5.

Being uniform or varying is a type property and can be declared with the `uniform` and `varying` type modifiers, respectively. A variable of a uniform type can only be set to a uniform value. A variable of a varying type can be set to a varying value as well as to a uniform value. The resulting value in the variable is then always considered varying.

For example:

```
uniform float x = 0.0; // x is a uniform value
varying float y = x;   // y is a varying value
```

A type without a `uniform` or `varying` type modifier is *auto-typed*, that is, its uniform or varying property is determined by the actual use of the type. For variable declaration, the considered use of the variable includes all its occurrences; if any occurrence of the variable requires the variable to be varying then the type of the variable will be considered varying. This implies that possible uses of the variable where it would be required to be uniform are considered errors in the program.

For example:

```
float3 x = 0.0;         // auto-typed
uniform float3 y = x;   // ! error, because varying x cannot be assigned to uniform
x = state::normal();    // x becomes varying by assigning the varying state::normal
```

6.4 Operators

expression : *assignment_expression* { , *assignment_expression* }

assignment_expression : *logical_or_expression*
[? *expression* : *assignment_expression*
| *assignment_operator assignment_expression*
]

assignment_operator : = | *= | /= | %= | += | -= | <<= | >>= | >>>= | &= | ^= | |=

conditional_expression : *logical_or_expression* [? *expression* : *assignment_expression*]

logical_or_expression : *logical_and_expression* { || *logical_and_expression* }

logical_and_expression : *inclusive_or_expression* { && *inclusive_or_expression* }

inclusive_or_expression : *exclusive_or_expression* { | *exclusive_or_expression* }

exclusive_or_expression : *and_expression* { ^ *and_expression* }

and_expression : *equality_expression* { & *equality_expression* }

equality_expression : *relational_expression* {(== | !=) *relational_expression* }

relational_expression : *shift_expression* {(< | <= | >= | >) *shift_expression* }

shift_expression : *additive_expression* {(<< | >> | >>>) *additive_expression* }

additive_expression : *multiplicative_expression* {(+ | -) *multiplicative_expression* }

multiplicative_expression : *unary_expression* {(* | / | %) *unary_expression* }

unary_expression : *postfix_expression*
| (~ | ! | + | - | ++ | --) *unary_expression*
| *let_expression*

postfix_expression :
primary_expression
{ ++
| --
| . *simple_name*
| *argument_list*
| [*expression*]
}

primary_expression : *literal_expression*
| *simple_type* `[ ]`
| `(` *expression* `)`

expression_statement : [*expression*] `;`

Most MDL types support the arithmetic, assignment, and comparison operators listed in Section 5.4.

Operators are defined in global scope. Operators may have overloaded version for different built-in types of operands. Additional programmer-defined overloads are not allowed.

Some MDL types have member variables, which are accessed by placing a period character '`.`' after the type value followed by the name of the member. For example `v.x` accesses the `x` member of `v`.

The comma operator and ternary conditional operator (represented by the '`?`' character) are also supported for use in expressions. The comma operator evaluates to the value of the right-most expression in the comma-separated list of expressions and shares its type. The '`?`' operator evaluates to the value of the second or third operands depending on the result of the first operand, which must be of type `bool`. If the first operand is `true` the result is the second operand otherwise it is the third operand. The second and third operand must have the same type, which defines the type of the expression. Whether only the second or the third operand that is selected by the '`?`' operator is evaluated or whether both are evaluated is implementation dependent. Expressions in these operands should therefore be restricted to have no side effects.

In the following sections, detailed descriptions of each type identify the operators supported by that type.

6.5 Constant expressions

MDL makes use of *constant expressions* in some places. The type of a constant expression is one of:

- One of the following types:

```
bool        bool2      bool3      bool4
int         int2       int3       int4
float       float2     float3     float4
double      double2    double3    double4
color
float2x2    float2x3   float2x4
float3x2    float3x3   float3x4
float4x2    float4x3   float4x4
double2x2   double2x3  double2x4
double3x2   double3x3  double3x4
double4x2   double4x3  double4x4
string
```

- An array type with a base type that is a legal type for a constant expression.
- A user-defined struct type where all fields have a type that is a legal type for a constant expression.
- A user defined enumeration.

A constant expression may consist of:

- bool, int, float, double and string literals (see Section 5.7).
- Constructors for base types, vectors, matrices, colors, and strings.
- Constructors for user defined structs with constant expressions as arguments and where all parameters without arguments have constant expression default initializer.
- Enumeration values.
- Indexing with constant expressions into constant expressions of array type.
- Selection of struct fields from constant expressions of struct type.
- The build-in operators `&&`, `||`, `<`, `<=`, `==`, `!=`, `>=`, `>`, `+`, `-`, `*`, `/`, and `%`. A division by zero is an error.

6.6 Global constants

constant_declaration : `const` *array_type constant_declarator* { , *constant_declarator*} ;

constant_declarator : *simple_name* (*argument_list* | = *conditional_expression*)
[*annotation_block*]

Constants can be declared globally using a variable declaration preceded by the `const` keyword. The `uniform` or `varying` type modifiers are not allowed on the variable type. The initialization is restricted to a constant expression. For example:

```
const float example_constant = 2 * 3.14159;
```


6.7 Scalars – float, double, int and bool

A `float` or `double` represents an approximation of a mathematical "real" number. MDL does not define the amount of precision nor the smallest or largest values representable by these types however it is guaranteed that `double` will have at least as much precision as `float`. The `float` and `double` types all represent a numeric value, but also provide a hint to the compiler to indicate variables that may require more or less precision.

Different platforms may choose to use different representations including floating or fixed point. Some platforms may use relatively low precision floating point for performance reasons.

An `int` represents an integer number. MDL requires that `int` has at last 16 bits available for bitwise operators. Besides that, MDL does not define the underlying representation of `int` values nor does it define the smallest or largest possible values that can be represented.

A `bool` represents a single Boolean value with possible values `true` and `false`.

6.7.1 Constructors

A scalar can be zero initialized with the default constructor, or set to `false` in the case of the `bool` type. A scalar can be initialized from any other scalar value even if the result entails a loss of precision.

In the following, all constructors are explained in detail, while implicit conversions, which apply in addition, are documented in Section 6.7.2.

`float()`
`double()`
`int()`

The default constructor creates a zero-initialized scalar.

`bool()`

The default constructor creates a `bool` value initialized to `false`.

```
float(float value)
float(double value)
double(double value)
```

A `float` or `double` can be constructed from any other scalar value, which may result in a loss of precision. When floating-point values are converted to floating-point types of less precision, the value is rounded to one of the two nearest values. It is implementation dependent to which of the two values it is rounded.

```
int(int value)
int(float value)
int(double value)
```

An `int` can be constructed from any other scalar value, which may result in a loss of precision. When floating-point values are converted to `int`, the fractional part is discarded. The resulting value is undefined if the truncated value cannot be represented as `int`.

```
bool(int value)
bool(float value)
bool(double value)
```

An `bool` can be constructed from any other scalar value. It is initialized to `false` for a numeric value equal to zero, and it is initialized to `true` for all non-zero numeric values.

`bool(bool value)`

An `bool` can be constructed from another `bool` value.

For example:

```
float x(5);
int   y(x);
bool  z(x);
```

All three constructor calls above are legal and result in three variables initialized with the following values:

Variable	Value
`x`	`5.0`
`y`	`5`
`z`	`true`

6.7.2 Conversion

When required by use in an expression, an scalar value will be implicitly converted to another scalar type provided there is no loss of precision. For example, a `float` value can be automatically converted to the `double` type but an explicit constructor call or initialization is needed to convert a `double` value to the `float` type. Note: With the undefined precision of the scalar types, the conversion from an `int` value to the `float` or the `double` type may have in fact some loss of precision, but shall be allowed implicitly.

The following table lists the types each scalar type can be automatically converted to:

Type	Can be converted to
`bool`	`int`, `float`, or `double`
`int`	`float`, or `double`
`float`	`double`

Note: In addition to these implicit conversion, arithmetic and other operators provide overloads, for example, to support mixed scalar-vector operations. (See vector operations in Section 6.8.4 or mixed scalar-matrix operations in Section 6.9.4.)

6.7.3 Operators

The `float`, `double`, and `int` types support the following operators:

```
=     /     /=    +     +=    -
-=    *     *=    ==    !=    <=
<     >=    >     ++    --
```

The `int` type additionally supports the modulo and bitwise operators:

%	%=	<<	>>	>>>	<<=	>>=	>>>=
~	&	^	|	&=	^=	|=	

The `bool` type supports the following operators:

=	==	!=	&&
!	||		

6.8 Vectors – float, double, int, and bool

MDL provides two, three, and four component vector types with either `float`, `double`, `int`, or `bool` component types. Vectors are named by taking the component type name and appending the dimension of the vector, which can be 2, 3, or 4.

For example:

```
float3 f3;  // a three-dimensional vector of floats
int2   i2;  // a two-dimensional vector of ints
bool4  b4;  // a four-dimensional vector of bools
```

6.8.1 Constructors

A vector can be zero initialized with the default constructor. A vector can be initialized from a single scalar, a series of scalars of the same number as the number of vector components, or a vector of the same dimension. In addition, a `float3` vector can be initialized from a value of type `color`.

In the following, all constructors are explained in detail with the vector type `float3` as a representative for all vector types, while implicit conversions, which apply in addition, are documented in Section 6.8.2.

`float3()` The default constructor creates a zero-initialized vector.

`float3(float value)`

All components of the vector are initialized with the scalar `value`.

`float3(float x, float y, float z)`

The components of the vector are initialized with the values of `x`, `y`, and `z`. The fourth parameter in the case of a `float4` vector is named `w`.

`float3(float3 value)`
`float3(double3 value)`

A vector can be constructed from any other vector of equal dimension, which may result in a loss of precision.

`float3(color value)`

A vector of type `float3` can be constructed from a value of type `color`. The `'x'`, `'y'`, and `'z'` components will be assigned the red, green, and blue color component values, respectively, in the linear sRGB color model. Note that this conversion may have significant runtime costs depending on the internal color representation of the `color` type. See Section 6.11 for more details on the `color` type.

Note: This constructor does not exist for the other vector types.

Some examples:


```
bool   b1 = true;           // a Boolean value to work with
int    i0 = 0,   i4 = 4;    // some scalar values to work with
float  s2 = 2.0, s3 = 3.0;  // more scalar values
float4 v4(b1, s2, s3, i4);  // 4-float constructor, implicit conversions to float
float3 v3(i0, b1, s2);      // 3-float constructor, implicit conversion of i0
bool3  vb3(v3);             // conversion of equal sized vectors with lost precision
```

These three vector constructor calls result in the three vectors initialized with the following values:

Variable	Value
v4	<1.0, 2.0, 3.0, 4.0>
v3	<0.0, 1.0, 2.0>
vb3	<false, true, true>

6.8.2 Conversion

When required by use in an expression, a value of a vector type will be implicitly converted to another vector type of the same length provided the element types allow implicit conversion.

The following table lists the automatic conversion rules for each vector type:

Type	Can be converted to
bool vectors	int, float, or double vectors
int vectors	float or double vectors
float vectors	double vectors

6.8.3 Members

The vector types support member variables to access their components and follow a common scheme to determine which members are available for each vector type.

The 'x', 'y', 'z', and 'w' members provide access to up to four components. A particular vector type will only support the first n components where n is the dimension of the vector. For example, float2 supports the 'x' and 'y' members.

Vector components can also be accessed using array indices and the array index can be a variable.

```
float4 v = ...;
float sum = 0.0;
for (int i=0; i<4; i++)
    sum += v[i];
```

In this example the vector v has its components summed using a loop.

6.8.4 Operators

Vectors support math operators in a component-wise fashion. The operator is applied to each component of the operand vectors independently and the result is a vector of the same size as the operands. Vectors support comparison operators returning a scalar `bool` value. The operand vectors must be the same size or one must be a scalar (in which case it is automatically promoted to a vector with the same dimension as the other operand).

The `float2, float3, float4, double2, double3, double4, int2, int3` and `int4` types support the following operators:

```
=       /       /=      +       +=      -       -=
*       *=      ==      !=      ++      --
```

The `int2, int3`, and `int4` types additionally support the modulo and bitwise operators, where the right-hand side of the shift operators must be a value of type `int`.

```
%       %=      <<      >>      >>>     <<=     >>=     >>>=
~       &       ^       |       &=      ^=      |=
```

The `bool2, bool3` and `bool4` types support the following operators:

```
        =       ==      !=      &&      !       ||
```

The following example illustrates in its second line how implicit conversions of scalar values and above operator overloads work together. The `int` literal value `1` is implicitly converted to a `float` type to match the only applicable subtraction operator that takes a scalar on the left-hand side and a `float3` vector on the right-hand side.

```
float3 x(1, 2, 3);
float3 y = x - 1;
```

The resulting value of the y variable is `(0, 1, 2)`.

6.9 Matrices – float and double

MDL provides several matrix types with column and row sizes ranging from two to four. Matrix elements can be of type `float` or `double`. Matrix types are named `type[columns]x[rows]` where `type` is one of `float` or `double`, `[columns]` is the number of columns and `[rows]` is the number of rows.

Specifically, the built-in matrix types are: `float2x2`, `float2x3`, `float3x2`, `float3x3`, `float3x4`, `float4x2`, `float2x4`, `float4x3`, `float4x4`, `double2x2`, `double2x3`, `double3x2`, `double3x3`, `double3x4`, `double4x2`, `double2x4`, `double4x3`, and `double4x4`.

The matrix type `float4x4` is used to represent coordinate-system transformations in Section 19.2.

Note: The naming convention, the column-major order implied below, and the coordinate-system transformation conventions in Section 19.2 are compliant with the respective OpenGL conventions.

6.9.1 Constructors

A matrix can be default constructed or constructed from a single scalar, a series of scalars of the same number as the number of matrix elements, a series of vectors of the same number as the number of columns of the matrix, or a matrix of the same dimensions.

In the following, all constructors are explained in detail with the matrix type `float3x2` as a representative for all matrix types, while implicit conversions, which apply in addition, are documented in Section 6.9.2.

`float3x2()`

The default constructor creates a zero-initialized matrix.

`float3x2(float value)`

The diagonal elements of the matrix are initialized with the scalar `value` while the other elements are initialized with zero.

For example, `float3x2(1.0)` results in the following matrix value:

$$\begin{pmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \end{pmatrix}$$

Or, `float4x4(1.0)` results in the 4×4 identity matrix.

`float3x2(float m00, float m01, float m10, float m11, float m20, float m21)`

A matrix can be constructed from a series of scalars in column-major order where the number of scalars passed to the constructor is the same as the number of elements of the matrix.

For example, `float4x3(1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.5, 0.0)` results in the following matrix value:

$$\begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.5 \\ 0.0 & 0.0 & 1.0 & 0.0 \end{pmatrix}$$

`float3x2(float2 col0, float2 col1, float2 col2)`

A matrix can be constructed from a series of vectors where each vector represents a column of the matrix. The dimension of the vectors must be the same as the size of columns in the matrix.

In the following example, the matrix `mat` gets the same value as in the previous example of the element-wise constructor:

```
float3 col0(1.0, 0.0, 0.0);
float3 col1(0.0, 1.0, 0.0);
float3 col2(0.0, 0.0, 1.0);
float3 col3(0.0, 0.5, 0.0);
float4x3 mat(col0, col1, col2, col3);
```

`float3x2(float3x2 value)`
`float3x2(double3x2 value)`

A matrix can be constructed from any other matrix of equal dimensions, which may result in a loss of precision.

6.9.2 Conversion

When required by use in an expression, a value of the matrix type will be implicitly converted to another matrix type of the same dimensions, provided that the element types allow implicit conversion. For example, the legality of the implicit conversion from a value of type float to one of type double allows float matrices to be converted into double matrices.

6.9.3 Members

The matrix types use array notation to provide access to their members, which are columns of the matrix. An index of zero refers to the first column of the matrix and indices of up to $n - 1$ (where n is the number of columns) provide access to the remaining columns.

The data type of matrix columns are vectors with dimension equal to the number of rows of the matrix. Since columns are vectors and vectors support array syntax to access vertex elements, individual elements of a matrix can be accessed with syntax similar to a multidimensional array.

For example:

```
float4x3 mat(1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.5, 0.0);
float3 col;
float element;

// col will equal <0.0, 1.0, 0.0> after the assignment
col = mat[1];

// element will equal 0.5 after the assignment
element = mat[3][1];
```


6.9.4 Operators

The matrix types support the following operators based on linear algebra:

```
=       /       /=      +       +=      -
-=      *       *=      ==      !=
```

The multiplication operator multiplies two matrices, a matrix and a vector, or a matrix and a scalar in a linear algebra style multiplication. When multiplying two matrices, the number of columns of the matrix on the left must be equal to the number of rows of the matrix on the right. The result of multiplying a `TxN` matrix with a `MxT` matrix is a `MxN` matrix. A vector can be multiplied on the right or left side provided the number of elements is equal to the number of rows when the vector is on the left side of the matrix and the number of elements is equal to the number of columns when the vector is on the right. A matrix can be multiplied component-wise with scalar.

The division operator is supported to divide a matrix component-wise by a scalar.

The equality and inequality comparison operators, `==` and `!=`, return a scalar `bool`. They require that the operands are matrices of the same dimension or one operand is a scalar. The matrix elements are compared component-wise. A scalar operand is first converted to a matrix of the same type using the corresponding constructor.

The assignment, add and subtract operators are applied component-wise and require that the operands are matrices of the same dimension.

6.10 String

A `string` is a sequence of characters of arbitrary length. MDL uses strings primarily as parameters to identify options or user defined categories. Because of the restricted need for strings in a material language, and the fact that some platforms have limited or no support for strings, MDL defines a limited set of string handling functionality.

String literals and the MDL character set are described in Section 5.

6.10.1 Constructors

`string()` The default constructor creates the empty string `""`.

`string(string value)`

A string can be constructed from another string.

6.10.2 Operators

The `string` type supports the following operators:

`=` `==` `!=`

6.11 Color

The MDL `color` type represents values across a continuous spectrum of wavelengths. The type is an abstraction of the conventional RGB representation of colors that enables materials to work with renderers that use a more accurate representation and rendering of colors.

A value of the `color` type maps wavelengths to `float` values. Wavelengths are given as `float` values in nanometers [*nm*] that are in the range [`limits::WAVELENGTH_MIN`, `limits::WAVELENGTH_MAX`], where the range bounds are constants in the `stdlimits` module (see Section 17) that depend on, and are provided by, the renderer implementation.

The exact implementation of the `color` type is not subject of the MDL specification. An implementation does not have to represent a spectrum exactly, it may choose to approximate spectra. For example, a simple implementation may actually represent colors as conventional RGB triples.

Unless otherwise specified, operations will be performed using the vacuum wavelength $\lambda_0 = \lambda \cdot \eta$, where λ is the local wavelength in a medium with an index of refraction of η.

6.11.1 Constructors

`color()`
: The default constructor creates a black color with zero amplitude everywhere.

`color(float value)`
: The amplitude of the color value is everywhere initialized to the scalar `value`, creating a gray color of corresponding magnitude.

`color(color value)`
: The copy constructor creates a copy of the color `value`.

`color(float r, float g, float b)`
`color(float3 rgb)`
: A color can be constructed from three `float` values or a single `float3` value, where the `'x'`, `'y'`, and `'z'` components are interpreted as red, green, and blue color component values, respectively, in the linear sRGB color model.

  Converting a `float3` value to a `color` value and back to a `float3` value shall result in the same value up to some numerical precision inaccuracies.

`color(float[<N>] wavelengths, float[N] amplitudes)`
: A color can be constructed from two `float` arrays that define samples for a smooth spectrum representation. The first array contains the wavelengths in increasing order and the second array, which must be of equal size, contains the amplitude values at these wavelengths. The construction may choose an approximation to represent the spectrum.

The following example creates a color from a function `intensity(float lambda)`, which returns the value at wavelength `lambda`. It samples the function at the wavelengths recommended for color construction by the state function `wavelength_base()` and passes both arrays to the color constructor.

```
color create_color() {
    float wavelengths[state::WAVELENGTH_BASE_MAX] = state::wavelength_base();
    float values[state::WAVELENGTH_BASE_MAX];
    for ( int i = 0; i < state::WAVELENGTH_BASE_MAX; ++i) {
        values[i] = intensity( wavelengths[i]);
    }
    return color( wavelengths, value);
}
```

Note: The construction of `color` values and the conversion between RGB values and color values potentially come at significant runtime costs depending on the implementation-defined color representation of the `color` type.

6.11.2 Conversion

There are no implicit conversions from and to values of type `color`.

6.11.3 Operators

The `color` type supports the following math operators which work on the color values:

```
=       /       /=      +       +=      -
-=      *       *=      ==      !=
```

All operators are defined for the `color` type as parameter on both sides. The `+`, `-`, and `*` operators are additionally defined for all combinations of a `color` type added, subtracted, and multiplied with a `float`. The `/` operator is additionally defined for the division of a `color` type divided by a `float`.

The equality and inequality comparison operators, `==` and `!=`, return a `bool` type. All other operators return a `color` type.

6.12 Textures

The MDL texture types represent references to texture data, associated sampler settings and lookup functions in the standard library. There are four texture types for four different texture shapes:

- `texture_2d` references texel data stored in a two-dimensional uniform grid.
- `texture_3d` references texel data stored in a three-dimensional uniform grid.
- `texture_cube` references texel data stored in a collection of six two-dimensional uniform grids, one for each direction (`+x`, `-x`, `+y`, `-y`, `+z`, and `-z`).
- `texture_ptex` references texel data stored in the PTEX format for two-dimensional surfaces.

Texel data consists conceptually of a single floating-point value, a vector of floating-point values, or a color value consistently over the whole texture.

The use of the texture types is restricted to uniform function and uniform material parameters. They cannot be used with variable definitions, except for variables in let-expressions, nor in structure or array type definitions.

A parameter of texture type can have an invalid reference value if no texture data was provided.

For texture data with an inherent frame of reference in space, MDL makes the following recommendations on texture space placement and orientation.

The origin of a `texture2D` type is in the lower left corner of the texture data. The x-axis extends to the right and the y-axis extends to the top.

The origin of a `texture3D` type is in the lower left back corner of the texture data. The x-axis extends to the right, the y-axis extends to the top, and the z-axis extends to the front. This convention implies that the respective unit basis vectors, e_x, e_y, and e_z, form a right-handed coordinate system with $e_x \times e_y = e_z$.

6.12.1 Constructors

```
texture_2d()
texture_3d()
texture_cube()
texture_ptex()
```

The default constructor creates an invalid reference value.

```
texture_2d( uniform texture_2d value)
texture_3d( uniform texture_3d value)
texture_cube( uniform texture_cube value)
texture_ptex( uniform texture_ptex value)
```

A texture can be created from another texture value of equal texture type.

```
texture_2d(   uniform string name, uniform tex::gamma_mode gamma = tex::gamma_default)
texture_3d(   uniform string name, uniform tex::gamma_mode gamma = tex::gamma_default)
texture_cube( uniform string name, uniform tex::gamma_mode gamma = tex::gamma_default)
texture_ptex( uniform string name, uniform tex::gamma_mode gamma = tex::gamma_default)
```

A texture can be created from a file path, defined in Section 2.2, given as literal argument to the `name` parameter of type `string`. The file path needs to name a file in one of the eligible texture file formats defined in Section 2.3. It is an error if the texture file does not exist.

Note: The gamma mode determines if the integration needs to apply an inverse-gamma correction or not before using the texture. This enumeration type is defined as part of the standard module `tex` in Section 20.3.

6.13 Light_profile

The MDL `light_profile` type is used to represent a reference to light profile data, which is typically provided by vendors of real-world physical lights to describe precisely how much light is emitted from a point light source in a particular direction.

The use of `light_profile` is restricted to uniform function and uniform material parameters. It cannot be used with variable definitions, except for variables in let-expressions, nor in structure or array type definitions.

A parameter of type `light_profile` can have an invalid reference value if no light profile data was provided.

6.13.1 Constructors

`light_profile()`

The default constructor creates an invalid reference value.

`light_profile( uniform light_profile value)`

A light profile can be created from another light profile value.

`light_profile( uniform string name)`

A light profile can be created from a file path, defined in Section 2.2, given as literal argument to the `name` parameter of type `string`. The file path needs to name a file in one of the eligible light profile file formats defined in Section 2.4. It is an error if the light-profile file does not exist.

6.13.2 Members

The `light_profile` type has no members.

6.14 Bsdf_measurement

The MDL `bsdf_measurement` type is used to represent a measured bidirectional scattering distribution function (BSDF) that can be used with the corresponding elemental distribution function `df::measured_bsdf` in Section 21.1.4 to render the measurement.

The use of `bsdf_measurement` is restricted to uniform function and uniform material parameters. It cannot be used with variable definitions, except for variables in let-expressions, nor in structure or array type definitions.

A parameter of type `bsdf_measurement` can have an invalid reference value if no measurement data was provided.

6.14.1 Constructors

`bsdf_measurement()`

The default constructor creates an invalid reference value, which renders a black opaque material if used.

`bsdf_measurement( uniform bsdf_measurement value)`

A BSDF measurement can be created from another BSDF measurement value.

`bsdf_measurement( uniform string name)`

A BSDF measurement can be created from a file path, defined in Section 2.2, given as literal argument to the `name` parameter of type `string`. The file path needs to name a file in one of the eligible BSDF measurement file formats defined in Section 2.5. It is an error if the file does not exist.

6.14.2 Members

The `bsdf_measurement` type has no members.

7 Arrays

array_type : *simple_type* [[[*conditional_expression* | < *simple_name* >]]]

Array types in MDL are similar in form to the typical array types of other programming languages, defining a sequence of data values of the same *element type*. Arrays are one-dimensional and of non-negative size, that is, arrays of zero size are allowed. Multi-dimensional arrays are not available in MDL.

MDL provides two kinds of array types: the *size-immediate array types* and the *size-deferred array types*. Both array types behave the same unless noted otherwise.

The size-immediate array type is the conventional array type whose size is immediately specified as a constant expression (see Section 6.5 for constant expressions). A size-immediate array type consists of a constant non-negative integer expression for the size within square brackets (`[]`) that follows the type identifier for the elements of the array. Two size-immediate arrays have the same type only if both the type of array element and their sizes are the same.

The following example shows a few variables of size-immediate array type:

```
int[4] channels;
float[3] weights;
bool[7] layer_active;
```

The size of a size-deferred array type is represented by a symbolic *size identifier*. The actual size is not defined immediately with the array type but deferred to the point where the array value is initialized, which can—for function and material parameters—be even outside of the MDL source files. Two size-deferred arrays have the same type only if both the type of array element and their size identifier are the same.

The size identifier, when used the first time, must be enclosed in angle-brackets (`<>`). This is the point of declaration of the size identifier. It is only allowed in function or material parameter lists, for example:

```
float sum_array( float[<count>] values);
```

The size identifier obeys normal scoping rules. It can also be used more than once to define further size-deferred arrays. All uses of a declared size identifier are then without the angle-brackets, which are reserved for declaring new size identifiers. The following example requires that both arguments need to be of the same type, and here in particular of the same array size, when calling this function:

```
float inner_product( float[<n>] a, float[n] b);
```

The value of the size identifier is a non-negative integer value of type `int`. It is not a l-value. However, although it cannot change its value, it is not a constant value in the sense of constant expressions of Section 6.5.

The full details about the use of size-deferred array types in function and material parameters are explained in Section 12.5.

7.0.3 Constructors of size-immediate arrays

Array constructors support only positional arguments and not named arguments, which are explained in Section 12.

type`[`*n*`]()`

The default constructor creates a size-immediate array value of the given element type *type* and given size *n*, where all elements are default constructed.

type`[`*n*`](` *type*`[`*n*`] value)`

The copy constructor creates a size-immediate array value of the given element type *type* and given size *n*, where all elements are copy-constructed from the corresponding elements in `value`.

type`[`*n*`](` *type* `value0, ...,` *type* $\texttt{value}_{n-1}$`)`
type`[](` *type* `value0, ...,` *type* $\texttt{value}_{n-1}$`)`

A constructor creating a size-immediate array value of the given element type *type* and of size *n*, where the elements are initialized with the parameter values. In the second variant of this constructor the array size is deduced from the number of arguments provided to the constructor.

Examples of size-immediate array constructors:

```
int[4] channels( 0, 1, 2, 3 );
int[4] channels2 = int[]( 0, 1, 2, 3); // identical to channels

float[3] weights  = float[3](); // zero initialized
float[3] weights2;              // identical to weights
```

7.0.4 Constructors of size-deferred arrays

Array constructors support only positional arguments and not named arguments, which are explained in Section 12.

type`[`*identifier*`]()`

The default constructor creates a size-deferred array value of the given element type *type* and given symbolic size *identifier*, where all elements are default constructed.

type`[`*identifier*`](` *type*`[`*identifier*`] value)`

The copy constructor creates a size-deferred array value of the given element type *type* and given symbolic size *identifier*.

Examples of size-deferred array constructors:


```
float array_examples( float[<count>] values) {
    float sum = sum_array( float[count]());   // default c'tor with all zeros
    return sum_array( float[count]( values)); // explicit copy c'tor
}
```

7.0.5 Conversion

For the purpose of function calls and overload resolution, see Section 12.4, a size-immediate array type can be implicitly converted to a size-deferred type of the same element type and compatible size identifier, and a dependent size-deferred array type can be implicitly converted to a defining size-deferred array type of the same element type. See Section 12.5 for all details on the use of size-deferred array types as function parameters.

7.0.6 Operators

Array elements are accessed by a non-negative integer expression of type `int` in square brackets following the array expression. These *array indices* are zero-origin; the index of the first element is zero, the index of the n^{th} element is $n - 1$.

```
float[3] weights;
bool[7] layer_active;
// ...
float red_factor = weights[0];
layer_active[6] = false;
```

Out-of-bounds accesses may not be checked in MDL and have undefined behavior.

8 Structures

struct_type_declaration : `struct` *simple_name* [*annotation_block*]
`{` {*struct_field_declarator*} `}` `;`

struct_field_declarator : *type simple_name* [`=` *expression*]
[*annotation_block*] `;`

MDL supports the definition of user-defined structures. A structure is a collection of named variables, possibly of different types. The declaration of a structure defines a new type name.

A structure is declared using the keyword `struct` followed by the name of the structure and the declaration of member variables, the structure's *fields*, enclosed in curly braces. Fields are declared with the same syntax as local variable declarations and may have an initializer expression. A field name can be any legal MDL identifier, though it cannot have the same name as the structure type.

Fields can have any built-in type or another user-defined structure type to produce a nested structure. A field can also have a size-immediate array type. Size-deferred array types are not allowed as field types.

Fields can have the uniform or varying modifier on their field types. In addition, the structure type can, when used, have another uniform or varying modifier for the whole structure type, which applies then to all field types. In this case, it is allowed if a field has already the same modifier on its type, but it is an error if it has a different modifier on its type.

For example, the following structure definition defines the type `color_pair`:

```
struct color_pair {
    color dark;
    color bright;
};
```

Once defined, a structure type can be used in the same manner as MDL's built-in types; to declare variables, function parameters and material parameters.

```
color_pair checkerboard;
```

A field may define an initialization value in the structure definition. The value can be any expression of that field's type including references to previous fields and function calls. The value is preceded by an equals sign (`=`).

```
struct color_pair {
    color dark   = color(0.2, 0.2, 0.2);
    color bright = color(1.0, 0.2, 0.1);
};
```

A structure can contain both uninitialized and initialized fields. All uninitialized fields, if any, must precede the first initialized field. (This restriction follows the requirements for default parameter values in function definitions; see Section 12.)

8.0.7 Constructors

A structure declaration looks generically like

```
struct structure_type {
    T1    name1;
    ...
    Ti-1  namei-1;
    Ti    namei = initializeri;
    ...
    Tn    namen = initializern;
};
```

where fields with initializer follow fields without initializer, and either or both can be omitted. With this notation, the following constructors are defined for each structure type:

---

structure_type`()`

The default constructor initializes all fields without initializer with their respective default constructor and all fields with initializer with the respective initializer value.

---

structure_type`(` *structure_type* `value)`

The copy constructor creates a field-wise copy of `value`.

---

structure_type`(` T_1 $name_1$`, ...,` T_{i-1} $name_{i-1}$`,` T_i $name_i$ `=` $initializer_i$`, ...,` T_n $name_n$ `=` $initializer_n$`)`

A constructor with a parameter for each field, which allows to define a value for each field at definition. Passing arguments to constructors is identical to passing arguments to functions described in Section 12. In particular, all fields without initializer require an explicit argument while all fields with initializer can be optionally left out, which initializes them to the value of their initializer expression. Note that, in addition, the default constructors mentioned above creates a value of a structure type without providing any argument to any field.

For example, given the following definition of `polygon`

```
struct polygon {
    int sides;
    color fill_color = color(1,1,1);
    color edge_color = color(0,0,0);
};
```

a variable of type `polygon` can be created with its constructor:

```
polygon triangle( 3, color(1, 0, 0), color(0.5, 0.5, 0.5));
```

Another variable of type `polygon` can be created with its default constructor:

```
polygon nogon;
```

The value of `nogon` is equal to `polygon(0)`, and also `polygon(0, color(1,1,1), color(0,0,0))`, which makes all default constructed values and initializers explicit.

8.0.8 Operators

A value of a structure type can be assigned to a l-value of the same type with the assignment operator (`=`).

Fields are referenced with the usual dot notation of the member selection operator, with a structure variable name followed by a dot (`.`) and the field name:

```
checkerboard.dark   = color(0.2, 0.2, 0.2);
checkerboard.bright = color(1.0, 0.2, 0.1);
// ...
color tile_color = checkerboard.dark;
```


9 Enumerations

enum_type_declaration : `enum` *simple_name* [*annotation_block*] `{`
enum_value_declarator { `,` *enum_value_declarator*}
`} ;`

enum_value_declarator : *simple_name* [`=` *assignment_expression*] [*annotation_block*]

MDL provides the capability to define an enumeration as a convenient way to represent a set of named integer constants.

An enumeration is declared using the keyword `enum` followed by a comma separated list of identifiers, the *enumerators*, enclosed in curly braces. An enumerator cannot have the same name as the enumeration type.

For example:

```
enum detail { low, medium, high };
```

The enumerators can be explicitly assigned literal values as well.

For example:

```
enum detail {
    low    = 1,
    medium = 2,
    high   = 3
};
```

This example defines a new type called `detail` with possible values of `low`, `medium` and `high`. Enumeration type values can be implicitly converted to integers which results in an integer with the explicitly assigned value. If explicit values are not specified, each element is assigned the value of its predecessor plus one. The first element is assigned the value zero.

The values associated with an `enum` are not required to be unique within the type. For example, the following are both legal `enum` declarations:

```
enum bool_states {
    on     = 1,
    yes    = 1,
    off    = 0,
    no     = 0
};

enum foo {
    a = 2,
    b = 1,
    c       // c will implicitly be assigned the value 2
};
```

The declaration of an enum defines a new type name and anonymous enums are not supported. The enum elements themselves define a symbol within the scope where the enum is declared.

The enum default constructor has the value of the first enumeration value.

10 Typedef

alias_type_declaration : `typedef` *type simple_name* `;`

The `typedef` specifier allows a new type name to be introduced which acts as a synonym for an existing type name. To declare a new type name, the `typedef` specifier is followed by the name of an existing type, an optional array specifier, and after that the new name, which thereafter acts as a synonym for the existing type.

For example:

```
typedef int number;
```

This defines a new type name `number` which is syntactically synonymous with `int`.

Declarations that use `typedef` to define a type name can appear at the file scope level and local scope levels in functions. On file scope level the new type name is valid from the location at which it is declared to the end of the file. On local scope levels the new type name is valid from the location at which it is declared to the end of the local scope.

Once a type name is defined it cannot be redefined in the same scope to name a different type.

The size in a type name for a size-deferred array is bound at the first use of this type in a scope.

Note that `typedef` does not introduce a new type, but instead only an additional name for an existing type. This means that for the purpose of differentiating parameters to match overloaded functions, the type name is not enough for two parameters to be considered different.

For example the following is not legal:

```
typedef int number;

int overload_example( number x) {}
int overload_example( int    x) {} // error -- redefinition of my_function
```

11 Control flow

compound_statement : `{` {*statement*} `}`

statement :
- *compound_statement*
- *type_declaration*
- *constant_declaration*
- *variable_declaration*
- *expression_statement*
- *if_statement*
- *switch_statement*
- *while_statement*
- *do_statement*
- *for_statement*
- *break_statement*
- *continue_statement*
- *return_statement*

type_declaration :
- *alias_type_declaration*
- *struct_type_declaration*
- *enum_type_declaration*

MDL supports the familiar programming constructs that control the flow of a function's execution. Specifically these are:

- The loop statements `for`, `while`, and `do-while`. The keywords `continue` and `break` are supported to control execution of the loop.
- The branch statements `if` with optional `else` clauses and `switch` statements with cases and optional defaults.
- A `return` statement to terminate a function and return a value.

11.1 Loops

for_statement : `for (` (*variable_declaration* | *expression_statement*) [*expression*] `;` [*expression*] `)` *statement*

while_statement : `while (` *expression* `)` *statement*

do_statement : `do` *statement* `while (` *expression* `) ;`

A for loop is declared using the keyword `for` followed by three expressions separated by semicolons and enclosed in parenthesis. The first expression is evaluated once before the loop begins. The second expression is evaluated once at the beginning of each iteration of the loop and will terminate the loop if it evaluates to `false`. The third expression is evaluated once at the end of each iteration of the loop. The loop body follows the `for` statement.

For example:


```
for (int i=0; i<10; i++) {
    // ...
}
```

Note that the first expression can declare variables which are visible in scope within the second and third expressions as well as the loop body.

The `continue` statement can be used in the body of the `for` loop to jump to the end of the current loop iteration.

The `break` statement will terminate the loop without further evaluation of the expressions in the `for` statement.

The `do` and `while` loop constructs are similar to each other. A `while` loop begins with the keyword `while` followed by an expression enclosed in parenthesis followed by the loop body. The body of the `while` loop will be executed as long as that expression evaluates to `true`.

When used in `do` or `while` loops, the `continue` statement will jump control to the end of the loop. The `break` statement will terminate the loop.

A `do` loop begins with the keyword `do` followed by the loop body and a while statement. The body will execute until a `break` statement is encountered or the expression in the `while` statement evaluates to `false`. The `while` test will be performed at the end of each iteration through the loop instead of at the beginning.

For example:

```
 while (x<n) {        // a while loop
   if (x==0) break;
   // ...
}

do {                 // a do loop
    // ...
} while (x<n);
```

11.2 Branches

if_statement : `if (` *expression* `)` *statement* [`else` *statement*]

switch_statement : `switch (` *expression* `) {` {*switch_case*} `}`

switch_case : `case` *expression* `:` {*statement*} | `default :` {*statement*}

An `if` statement is declared using the keyword `if` followed by an expression contained in parenthesis followed by a statement that will be executed if the expression evaluates to `true`.

The `else` keyword can optionally follow indicating a statement that will be executed if the expression evaluates to `false`. In nested conditional statements, an else statement matches the closest unmatched if

statement.

For example:

```
if (x<n) {
    // statements that execute if x<n
} else {
    // statements that execute if not x<n
}
```

Switch statements allow control to jump to a selected body of statements based on an integral value. A switch statement is declared using the keyword `switch` followed by an expression enclosed by parenthesis. The expression must evaluate to an `int` valued expression. Following this is a series of `case` blocks enclosed in curly braces.

The case blocks are declared using the keyword `case` followed by a colon, then a constant integral value, and finally the statements of the case block. If the value of the expression in the `switch` statement equals the case value, that case block will be executed. Execution will fall through to the subsequent case block unless control is terminated with a `break` statement.

An optional default case block can be declared using the `default` keyword followed by a colon and then the statements to be executed if no other case statement matches the expression value.

A case block has its own scope that extends to the next case block or the end of the switch block if it is the last case block. In consequence, variables declared in a case block are not accessible in any other case block.

For example:

```
switch (x) {
    case 0:
        // statements that will be executed if x==0
        break;
    case 1:
        // statements that will be executed if x==1
        break;
    case 2:
        // statements that will be executed if x==2
        break;
    default:
        // statements that will be executed if x!= 0, 1, or 2
        break;
}
```

11.3 Jumps

break_statement : `break ;`

continue_statement : `continue ;`

return_statement : `return` *expression* `;`

The return statement terminates execution of the current function and returns control to the calling function or completes execution of the function if the current function has been called by the renderer.

A return statement is declared using the `return` keyword followed by a return value, whose type is equal to or has an implicit conversion to the return type of the function.

The `break` statement is used to terminate a loop or `switch` statement. The `continue` statement is used to terminate an iteration of a loop and proceed to the next iteration of the loop.

The `return`, `break`, and `continue` statements all cause the flow of control to jump to a new location in the code.

12 Functions

function_declaration : *type* [*annotation_block*] *simple_name*
parameter_list [*frequency_qualifier*] [*annotation_block*]
(; | *compound_statement*)
| ...

parameter_list : ([*parameter* { , *parameter*}])

parameter : *type simple_name* [= *assignment_expression*]
[*annotation_block*]

argument_list : (
[*named_argument* { , *named_argument*}
| *positional_argument* { , *positional_argument*}
{ , *named_argument*}
]
)

named_argument : *simple_name* : *assignment_expression*

positional_argument : *assignment_expression*

MDL can define global, pure functions outside of material declarations that can be called from other functions or to provide values to material input parameters.

A function is declared and defined in the main scope of a source file. The declaration consists of the return type, followed by the function name, followed by a list of function parameters surrounded by parenthesis. The parameter list is a comma-separated list of parameter declarations, each comprised of a type name followed by the parameter name and an optional default initializer. Parameters with default initializer must follow parameters without initializer. Default initializer can refer to parameters before them. For example:

```
float function_example( float a, float b = 0.0, float c = b);
```

Functions must have a declaration that appears before any reference to the function is made. The function body can be included as part of the declaration or a separate function definition can occur later in the source file, after the function declaration. It is not an error to have the same function declaration more than once in a MDL file if only the first declaration uses default initializers. The function definition must be unique.

The function body opens a new scope for variable definitions. In addition, function parameters are part of the function body scope. It is thus not possible to shadow function parameters immediately in the function body.

Unused function parameters may create a warning message at compilation time. Such a warning can be suppressed with the `unused()` annotation described in Section 14.

Global functions can be called from material parameter initializers or from other functions. A function call consists of the name of the function followed by a list of function arguments surrounded by parenthesis.

The argument list is a comma-separated list of expressions. The argument list may be empty. A function call is an expression which has the type of the return type of the function declaration and is then subject to implicit conversion if the return value is used. For example, above declaration can be called as follows with its return value assigned to a variable:

```
double d = function_example( 5.0);
```

Function arguments are evaluated and assigned in the order from left to right to the corresponding parameters. Parameters without an argument must have a default initializer. Default initializers are also evaluated in the order from left to right.

For example, above function call is equivalent to:

```
double d = function_example( 5.0, 0.0, 0.0);
```

In addition to such *positional arguments*, where the position in the argument list decides to which parameter name an argument binds, MDL supports *named arguments*, where the parameter name is given explicitly with a *parameter selector* and separated with a colon from the argument value. This simplifies the use of functions if only a few of its parameters get values and the others remain default initialized.

For example, above function call is also equivalent to the following three variants:

```
double d1 = function_example( a: 5.0);
double d2 = function_example( b: 0.0, a: 5.0);
double d3 = function_example( c: 0.0, a: 5.0);
```

Note: Named arguments imply that the name of function parameters are part of the function signature. The name is not an implementation decision for a function and cannot be changed without considering all function calls.

Positional and named arguments can be used in a single function call. Positional arguments must be provided first and followed by named arguments.

All parameters without default initializer must have a value provided in the call, either through a positional or named parameter. Positional arguments can also be used on parameters with default initializers. No parameter is allowed to have more than one argument setting its value in a function call. This can occur with multiple named arguments, which is always a function call error. It can also occur with a combination of a positional and named argument, which eliminates this function declaration from the overload set (see overload resolution in Section 12.4) while other function declarations might still match.

12.1 Parameter passing

parameter : *type simple_name* [= *assignment_expression*]
[*annotation_block*]

Function arguments are semantically passed by value to function parameters. The argument's value is copied into the function being called, but the parameter's value is not copied back out. Results are only

passed back to the callee through the functions return value. Multiple results need to be packaged into a suitable type, such as a structure type or array type.

A function call is *matching* a function declaration if it has the same function name and its arguments match the parameter types. An argument matches a parameter type if the type of the argument expression is equal to the parameter type or an implicit conversion exists from the type of the argument expression to the parameter type including an implicit promotion from a uniform type to a varying type.

The copying behavior described above specifies the semantics of function parameter passing. A particular MDL implementation may choose another implementation, for example, for better performance, as long as the described copy-semantic behavior is guaranteed.

A function return value must be initialized with a return statement.

12.2 Uniform and varying parameters and functions

Function parameter types can be explicitly declared uniform or varying. Otherwise they are auto-typed and have the same property as the call arguments.

Note: For larger parameter lists auto-typing may lead to many possible combinations of which some may not be legal for type reasons, such as a varying value that cannot be assigned to a uniform parameter or variable. These illegal combinations must be detected at compilation time with an error diagnostic, but, otherwise, these combinations may not necessarily require different implementations.

A function return type can be explicitly declared uniform or varying. Otherwise it is auto-typed. In case the return type is auto-typed, there can only be two sources that can make the return type varying, otherwise it will be uniform: varying function parameters and function calls in the function body that return varying values. To simplify the static analysis of functions, an auto-typed return type is defined to be varying if and only if one or both of the following two conditions apply:

1. At least one of the function parameters is varying. This is independent of whether the actual varying value would influence the return value or not.
2. The function body contains calls of other functions that are varying or have varying return types. This is independent of whether the actual execution of the function body would call these functions and whether the function results would influence the return value or not.

A compiler can analyze the second condition. However, MDL allows a programmer to declare explicitly if a function is varying or uniform — if the second condition is met or not — by placing the `varying` or `uniform` keyword after the closing parenthesis of the parameter list.

For example:

```
float3 uniform_function_example( float3 param) uniform;  // function declaration

float3 varying_function_example( float3 param) varying { // function definition
    return param + state::normal();
}
```

For a varying function, an auto-typed return type is always varying. More interesting is a uniform function: The compiler can guarantee that the implementation does not call varying functions and when calling it, an

auto-typed return type is varying if and only if one of the function arguments is varying, and it is uniform otherwise.

This allows the natural and concise implementation of functions that can be used in a context of varying values as well as uniform values, such as standard library functions.

For example:

```
float sin( float a) uniform;
```

Calling this `sin` function with a varying argument results in a varying result value and calling it with an uniform argument results in an uniform result value.

12.3 Function and operator overloading

Multiple functions with the same name can be defined, called *function overloading*, provided they differ at least by the number or types of parameters. It is not sufficient for overloaded functions to only differ by return type, by name of parameters, or by their uniform, varying, or auto-typed property. For the purpose of being a different parameter type in this definition, it is also not sufficient if two types differ just by their uniform, varying, or auto-typed property.

A module (Section 15) always imports and exports all overloads of a function together. In addition, a module can add additional overloads to a function that it imports unqualified and may re-export this set of overloads again.

Note: The rules on imports prevent that function definitions from different modules are imported unqualified in the same module at the same time. The set of overloaded functions is thus always declared in a single module.

Various operators in MDL are overloaded for different parameter types, but additional user-defined overloads are not allowed.

12.4 Function overload resolution

Scope and overload resolution is the process that decides which function definition or operator is called at a particular function or operator call location. Everything said in this section for functions applies for operators as well unless specifically said otherwise. The handling of operators is simplified to the extent that all operators are defined in global scope and no additional user-defined overloads exist.

The identifier used in a function call is searched from the innermost scope enclosing the call outwards until a scope is found in which this identifier is declared. This declaration must declare a function, otherwise the program is ill-formed. The set of all function declarations in that scope with the same identifier is now considered for overload resolution.

Qualified identifiers are searched for in the module named in the qualification of the identifier. A similar set of all function declarations in that module scope is now considered for overload resolution.

All function declarations for which the signature does not match the call are eliminated from the set. A function signature *S* does not match the call if:

- The type of a positional argument in the call is not equal to the type of the parameter in that position in *S* or does not have an implicit conversion from the argument type to the parameter type.
- The parameter name of a named argument in the call does not exist in *S*.
- The type of a named argument in the call is not equal to the type of the named parameter in *S* or does not have an implicit conversion from the argument type to the parameter type.
- A parameter is specified more than once in the call if applied to *S*.

If the set of matching function declarations is empty after the above eliminations, the program is ill-formed.

If the set contains two or more function declarations, these will be compared pairwise and the *less specific* declaration will be removed from the set. This process is iterated until no further eliminations are possible.

If the set contains only one function declaration after the above eliminations, the corresponding function is called. Otherwise the overload resolution is ambiguous and the program is ill-formed.

A function signature *T* is less specific than a function signature *U* if for each positional or named argument that is used in the function call the types of the corresponding parameters are identical or there exists an implicit conversion from the type of the parameter of *U* to the type of the parameter of *T*.

Note: Above rules imply in MDL that if exactly one function declaration exists in the overload set for which all parameters match exactly without implicit conversions, the corresponding function is called.

The following example shows a few cases of which function declaration is chosen for which function call. Note that the overload of the standard library function is not necessarily recommended in practice.

```
using std import max;             // (1) unqualified import of all overloads of max

int foo( float  x);               // (2)
int foo( double x);               // (3)

int bar( int    a, int    b);     // (4)
int bar( float  a, double b);     // (5)
int bar( double b, float  b);     // (6)

float max( int a, float b);       // (7) (not recommended to overload std library)

int overload_resolution_example() {
    foo( 1);             // calls (2) since (2) is more specific than (3)
    foo( 1.0f);          // calls (2) since (2) is an exact match
    foo( 1.0d);          // calls (3)

    bar( 1, 1);          // calls (4)
    bar( 1.0f, 1.0f);    // ! fails with (5) and (6) being the ambiguous matches

    max( 1, 1);          // calls max using (1), the max(int,int) from std::
    max( 1, 1.0f);       // calls (7) as an exact match
}
```

Note: Function overloads are more naturally used with positional than with named arguments. The following example illustrates some of the possible results with named arguments.


```
int foo( int   a = 0, float b = 0);  // (1)
int foo( float b = 0, float a = 0);  // (2)

int overload_resolution_example() {
    foo();               // ! fails with (1) and (2) being the ambiguous matches
    foo( 1);             // calls (1) since here (2) is less specific than (1)
    foo( 1.0f);          // calls (2)
    foo( a: 1);          // calls (1) since here (2) is less specific than (1)
    foo( 1, b: 1.0f);    // calls (1) because the call would set b twice for (2)
    foo( a: 1, b: 1.0f); // calls (1) since here (2) is less specific than (1)
}
```

12.5 Function parameters of size-deferred array type

A size-deferred array type, see Section 7, can be used for function parameters. When the function is called, the size identifier — as a read-only variable of type `int` — is added to the scope of the function. The value of this identifier is then initialized to the size of the array passed as the function argument. The size identifier can be used throughout the function scope as a read-only value as any other variable of type `int`.

For example, the following function has one parameter, a size-deferred array named `values`.

```
float sum_array( float[<count>] values) {
    for (int i = 0; i < count; ++i)
        result += values[i];
    return result;
}
```

The name of the size identifier of the `values` array is `count`. Within the body of the function, `count` can then be used to control the `for` loop.

Any size-immediate or size-deferred array value can be passed as a function argument of a size-deferred array parameter.

For example, in the following call of `sum_array`, the value of the `count` size identifier is 4. The initialized value of `total_weight` from the result of the function call is 2.4.

```
float[4] weights( .2, .3, .5, 1.4 );
float total_weight = sum_array(weights);
```

Size-deferred arrays using different identifiers inside the angle brackets are of different type. It is still possible to test the sizes at run-time and provide component-wise operations. For example:

```
float inner_product( float[<n>] a, float[<m>] b) {
    if (n != m)
        return 0.0;
    float result = 0.0;
    for (int i = 0; i < n; ++i)
        result += a[i] * b[i];
    return result;
}
```

In above example, the array sizes can be enforced to be equal by using the same size identifier for both arrays; enclosed in angle-brackets (`<>`) at the first occurrence and plain inside the square-brackets (`[]`) at all later occurrences. For example:

```
float inner_product( float[<n>] a, float[n] b) {
    float result = 0.0;
    for (int i = 0; i < n; ++i)
        result += a[i] * b[i];
    return result;
}
```

Assuming both versions of `inner_product` are offered as function overloads, which is possible, a function call would choose this overload of `inner_product` if both arguments are of the same type. If they are of different type, the first implementation is chosen, although the actual array sizes my be equal.

There could be further specializations of `inner_product`, for example:

```
float inner_product( float[2] a, float[2] b) {
    return a[0]*b[0] + a[1]*b[1];
}
```

This overload of `inner_product` would only be chosen if the argument arrays are of size-immediate array type `float[2]`. It would never be chosen if one or both arguments are of a size-deferred array type.

The return type of a function may be a size-deferred array type, but the identifier inside the square brackets must already have been declared as the array size of (at least) one of the functions parameters. For example:

```
int[n] id( int[<n>] a) {
    return a;
}
```

It is possible to declare local variables of size-deferred array type if the size identifier comes from a parameter declaration. For example:


```
int local_variable_example( int[<n>] a) {
    int[n] b = a; // copy constructed array of same value
    float[n] c;   // default constructed array of same size
    ...
}
```

Another example illustrates the use of a local variable of a dependent size and its use to accumulate a result returned through a size-deferred array return type:

```
float[n] scale( int[<n>] a, float s) {
    float[n] result;
    for (int i = 0; i < n; ++i)
        result[i] = s * a[i];
    return result;
}
```

Any array of a given element type can be passed as an argument for a parameter that is declared as a size-deferred array type of the same element type. For example:

```
float magic_sum( float[<n>] a) {
    float[3] b( 0, 1, 2);
    return sum_array(a) + sum_array(b);
}
```

Array parameters can have default initializers like other parameters:

```
float initializer_example( float[<n>] a = float[](1.0)) {
    return sum_array(a);
}
```

For size-deferred array parameters, a function call can nonetheless use an array argument of different size than used in the initializer.

```
float result = initializer_example( float[](3.0, 4.0, 5.0));
```


13 Materials

Materials are built around two concepts: the `material` type and *material definitions*. The `material` type has a few related types, among which the distribution functions are the central connection to the renderers to describe surface, emission and volume properties. The material definitions have a signature like a function returning a `material`, but their implementations differ fundamentally from function bodies.

Material definitions exist in two variants: the *encapsulating material definition* described in Section 13.5 and the *open material definition* described in Section 13.6.

Note: This section introduces the following keywords:

```
material   edf   material_surface   material_volume
bsdf       vdf   material_emission  material_geometry
```

types, their use is restricted to what is described in this section and the standard module in Section 21. In particular, user-defined functions shall not have parameters, return types, or local variables using these types, and user-defined materials shall not have parameters using these types with the exception of the `material` type as detailed in Section 13.7.

13.1 The material type

The `material` type is defined as a built-in structure-like type with the following definition:

```
struct material {
  uniform bool      thin_walled = false;
  material_surface  surface     = material_surface();
  material_surface  backface    = material_surface();
  uniform color     ior         = color(1.0);
  material_volume   volume      = material_volume();
  material_geometry geometry    = material_geometry();
};
```

Primary material definition.

`thin_walled` – If `false`, surfaces are treated as interface between two volumes and the `ior` and `geometry` fields describe properties of the enclosed volume. If `true`, surfaces are treated as double-sided and enclose an infinitely thin volume. In the thin-walled case, front and back side use both the front side material if the BSDF and EDF of the `backface` field are left at their default values. Otherwise the back side uses the description provided with the `backface` field. If different materials are used for the front face and the back face, both materials must have equal transmission interactions.

`surface` – Front surface reflection, refraction and emission characteristics. Also used for the back surface if `thin_walled` is `true` and the BSDF and EDF of the `backface` field are left at their default values.

`backface` – Back surface reflection, refraction and emission characteristics if `thin_walled` is `true` and if either the BSDF or the EDF contained in this field is set to a non-default value. Otherwise this field is ignored.

`ior` – Index of refraction used in front and back surface transmission interactions.

`volume` – Scattering and absorption of light within the participating medium in a volume.

`geometry` – Controls for geometric manipulations of the object surface.

The `material` type enables three categorial different materials with respect to how they interpret surfaces:

1. A surface is an interface between volumes.
2. A surface represents a thin wall enclosing conceptually an infinitely thin volume with identical materials on both sides, for example, window glass can be modeled this way.
3. A surface represents a thin wall with different materials on both sides, in which case both materials must have equal transmission interactions or otherwise this is an error.

All cases and the corresponding settings of the different fields in the material description are described in the following table:

field name	interface	thin wall with equal sides	thin wall with different sides
`thin_walled`	`false`	`true`	`true`
`surface`	used for both sides†	used for both sides†	used for front side
`backface`	ignored	must be set to defaults	used for backside
`ior`	used	used but no refraction	used but no refraction
`volume`	used for enclosed volume	ignored	ignored
`geometry`	used once with front-side orientation for both sides		

†**Note:** In the first two cases where `surface` is used for both sides, the EDF contained in `surface` is only used for the front side and not for the back side. There is no emission on the back-side unless an EDF is specified with the `backface` field and `thin_walled` is set to `true`.

The `material` type shares the properties of structure types to construct values using constructors and to access fields using the dot-operator. It can be used in other structure types, arrays, or type definitions, but those are then also subject to the restrictions that apply to the `material` type.

13.2 Distribution function types: bsdf, edf, and vdf

Light interaction with objects is divided into three categories to describe reflection and transmission by a surface, dispersal and absorption within a volume, and the emission of light by a surface. Collectively, these properties are defined by three data types in MDL:

`bsdf` – A *bidirectional scattering distribution function* (BSDF) that describes the interaction of light with the surface. The default constructor `bsdf()` creates a non-scattering, i.e., black BSDF.

`edf` – An *emission distribution function* (EDF) that describes the light-emitting properties of the surface. The default constructor `edf()` creates a non-emitting, i.e., black EDF.

`vdf` – A *volume distribution function* (VDF) that describes the scattering and absorption of light in a volume. The default constructor `vdf()` creates an isotropic VDF.

MDL defines a standard set of distribution functions based on these types for use in constructing the components of a material. These distribution functions are described in Section 21.

The distribution functions are defined in the `df` module of MDL. In the examples that follow in Section 13.4, distribution function names are prefixed by the `df::` name qualifier (for example, in `df::diffuse_reflection_bsdf`).

13.3 Compound types in the fields of the material

The four fundamental properties of the `material` type—for the surface, volume, emission, and geometric characteristics of objects — are represented by `struct` types. These structs contain fields that specify instances of the three distribution functions.

One field in the emission property uses the following built-in enumeration type:

```
enum intensity_mode {        This mode determines the measure used for the emission intensity.
    intensity_radiant_exitance,
    intensity_power
};
```

The four `struct` types for the fundamental properties of the `material` type are:

```
struct material_surface {
  bsdf              scattering = bsdf();
  material_emission emission   = material_emission();
};
```

Surface properties of the material model.

`scattering` – BSDF describing the reflective and/or transmissive scattering of incoming light.

`emission` – Light emission properties of the material defined by a struct that contains both the EDF and an intensity factor.

```
struct material_emission {
  edf            emission  = edf();
  color          intensity = color(0.0);
  intensity_mode mode      = intensity_radiant_exitance;
};
```

Emission properties of the material model.

`emission` – The emission distribution function.

`intensity` – Scaling factor for the result of the emission distribution function. For local EDFs, this is the radiant exitance if `mode` is set to `intensity_radiant_exitance` (the default), or power if `mode` is set to `intensity_power`.

`mode` – The measure used for the `emission` parameter.


```
struct material_volume {
  vdf   scattering             = vdf();
  color absorption_coefficient = color();
  color scattering_coefficient = color();
};
```

Volume properties of the material model.

`scattering` – VDF describing the scattering of light within the participating medium.

`absorption_coefficient` – The probability density (per meter in world space) of light being absorbed by the participating medium.

`scattering_coefficient` – The probability density (per meter in world space) of light being scattered from its current direction.

```
struct material_geometry {
  float3 displacement   = float3(0.0);
  float  cutout_opacity = 1.0;
  float3 normal         = state::normal();
};
```

Geometric properties of the material model.

`displacement` – Vector defining direction and distance of position modification of current surface position in internal space.

`cutout_opacity` – A value between `0.0` and `1.0` for a cutout mask, where for a value of `0.0` the object geometry is ignored and for a value of `1.0` the object geometry is there.

`normal` – Surface normal vector in internal space to use for all calculations at the current surface position.

The different material fields are in principle evaluated independently and at the renderer algorithms sole discretion. The evaluation of the fields in the `material_geometry` structure though have the potential to change the renderer state described in Section 19, for example, `state::normal`, and thus can influence the evaluation results of other fields that are evaluated later. The relevant evaluation orders are as follows: The geometry fields are evaluated before all surface fields. Within the geometry fields, `displacement` is evaluated first, `cutout_opacity` second and `normal` last.

The `float3` vectors of the `material_geometry` structure are defined in the internal space of the renderer. The state functions for coordinate space transformations, see Section 19.2, allow the use of object-space or world-space dependent values for those vectors. Note though that a world-space dependency can have negative performance and memory impact on renderers that support object instancing because multiple instances with a shared object might have different displacements in world coordinates and can no longer be shared.

13.4 Instantiating a material

postfix_expression : *primary_expression*
{...
| *argument_list*
}

argument_list : `(`
`[`*named_argument* `{ , }` *named_argument*`}`
`|` *positional_argument* `{ , }` *positional_argument*`}`
`{ , }` *named_argument*`}`
`]`
`)`

The `material` type can be used to define a material in a syntactic form that resembles a structure type constructor including positional and named arguments. For example, to create a diffuse green material, the `surface` field of the `material` struct is defined as a `material_surface` struct in which the BSDF is tinted green.

```
material(
    surface : material_surface(
                scattering : df::diffuse_reflection_bsdf(
                               tint : color( 0.5, 1.0, 0.5)
                             )
              )
)
```

In this example, the value of the `scattering` parameter to `material_surface` is a `bsdf` from the standard MDL module `df` (see Section 21.1.1).

Material instantiations are used in material definitions, which are explained in the next two sections.

13.5 Encapsulating material definitions

function_declaration : ...
| *type* `[`*annotation_block*`]` *simple_name*
parameter_list `[`*annotation_block*`]` `=` *expression* `;`

parameter_list : `(` `[`*parameter* `{ , }` *parameter*`}]` `)`

parameter : *type simple_name* `[=` *assignment_expression*`]`
`[`*annotation_block*`]`

A material definition gives a name to a specific material instance. In addition, a material definition can define input parameters that can be used within the material instantiation in expressions and function-call arguments to initialize properties of the material model or of other already existing materials. Parameterizing a material definition enables the encapsulation and customization of materials to create custom material libraries. This parameterization can be thought of as analogous to the definition of a function.

Technically, a material definition is a function definition with a, possibly empty, parameter list that returns a material. The parameter types are restricted to those allowed for function parameters plus the `material` type, see Section 13.7 for its use. In particular, material parameters shall not be of type `bsdf`, `edf`, `vdf`, `material_surface`, `material_volume`, `material_emission`, or `material_geometry`.

Deviating from function definitions is the implementation of the material definition. Instead of a function body, a material definition requires an assignment from an expression of type `material`. This can be

an instantiation of the `material` structure type itself, the result of another material definition, or a let-expression as explained in Section 13.8.

The following example uses the previous material instance for a diffuse green material and gives it a name without parameters:

```
material green_material()
  = material(
      surface : material_surface(
                  scattering : df::diffuse_reflection_bsdf(
                                 tint : color( 0.5, 1.0, 0.5)
                               )
                )
    )
```

The following example extends the previous example to a more generic diffuse material with a `tint` parameter in the signature. The `tint` parameter input is used to set the equally-named `tint` parameter of the `df::diffuse_reflection_bsdf`:

```
material diffuse_material( color tint = color( 1.0))
  = material(
      surface : material_surface(
                  scattering : df::diffuse_reflection_bsdf(
                                 tint : tint
                               )
                )
    );
```

Given this parameterization of `tint` in `diffuse_material`, the `green_material` definition—using `diffuse_material` in a syntactically identical manner to calling a function—can be defined instead as:

```
material green_material( )
  = diffuse_material( color( 0.5, 1.0, 0.5));
```

The `tint` parameter has been bound by the encapsulation, and is inaccessible by a user of `green_material`.

13.6 Open material definition

function_declaration : ...
| *type* [*annotation_block*] *simple_name*
(*) [*annotation_block*] = *expression* ;

An *open material definition* functions in the same manner as the encapsulation of the previous example. However, parameters are not encapsulated in this definition. The use of the wildcard ('*') hints at the idea that all parameters available for the material on the right-hand side of the assignment remain accessible by a user of the new material definition.

Open material definitions can be used to define a family of materials from one material definition using different default values for its parameters. The parameters remain accessible for later adjustments. Open material definitions are also called *material variants*.

For example, if the green material is defined as an open material definition as follows:

```
material green_material( * )
  = diffuse_material( color( 0.5, 1.0, 0.5) );
```

Then, a user of the green material can define another diffuse material based on the green material changing its tint, for example, to a light green material. This example uses again an open material definition, which leaves the `tint` parameter accessible:

```
material light_green_material( * )
  = green_material( color( 0.7, 1.0, 0.7) );
```

Such a refinement of an open material definition can also be done using an enclosing material definition.

13.7 Materials as material parameters

Material parameters can be of type `material`, which enables generic adaptor materials and the re-use of materials and material components in other materials. The components of a `material` parameter are accessed in the material definition using the dot notation for the fields of the material.

The following example provides a generic adaptor that takes a material and returns the equivalent thin-walled material with identical front-face and back-face material properties:

```
material thin_walled_material( material base)
  = material(
      thin_walled : true,
      surface     : base.surface,
      volume      : base.volume,
      geometry    : base.geometry
    );
```

The next example provides a generic material that adds a clear-coat layer to another material. Technically, this example adds a specular reflective layer with a Fresnel blend over the base material. The Fresnel blend is influenced through an additional `ior` parameter.

```
material add_clear_coat( color    ior = color(1.5),
                         material base )
  = material(
      volume   : base.volume,
      geometry : base.geometry,
      surface  : material_surface(
                   emission   : base.surface.emission,
                   scattering : fresnel_layer(
                                  layer : specular_bsdf(),
                                  base  : base.surface.scattering,
                                  ior   : ior
                                )
                 )
    );
```


13.8 Sharing in material definitions with let-expressions

let_expression : `let`
(*variable_declaration*
| `{` *variable_declaration* {*variable_declaration*} `}`
)
`in` *unary_expression*

Material definitions can become deeply nested as parameters take further material definitions as values and they may contain common subexpressions without the ability to share them. A *let-expression* can help to improve both aspects; more structural flexibility in describing material definitions and shared common subexpressions.

A let-expression consists of a block of variable declarations and a final expression of type `material`, which becomes the result of the let-expression. The scope of the variables extends to the end of the let-expression, such that the final expression can make use of the variable values. In general, variables can also be used in subsequent definitions of other variables in the let-expression. Variables are read-only and cannot be re-defined in the same let-expression.

Variables in let-expression can be of any MDL type and in particular any of the following types: `texture_2d`, `texture_3d`, `texture_cube`, `texture_ptex`, `light_profile`, `bsdf_measurement`, `material`, `bsdf`, `edf`, `vdf`, `material_surface`, `material_emission`, `material_volume`, or `material_geometry`.

For example, the `add_clear_coat` material defined in the previous section can be rewritten using variables in a let-expression to clarify the construction of the `surface` input to the material:

```
material add_clear_coat( color    ior = color(1.5),
                         material base)
  = let {
      bsdf coat = specular_bsdf();
      bsdf coated_scattering = fresnel_layer(
                                 layer : coat,
                                 base  : base.surface.scattering,
                                 ior   : ior
                               );
      material_surface coated_surface(
                         emission   : base.surface.emission,
                         scattering : coated_scattering
                       );
    } in
      material(
        volume   : base.volume,
        geometry : base.geometry,
        surface  : coated_surface
      );
```

Note that the `bsdf` variable `coat` is used as the value of the `layer` parameter in `fresnel_layer`.

13.9 Conditional expressions for materials

conditional_expression : *logical_or_expression* [`?` *expression* `:` *assignment_expression*]

Conditional expressions with the ternary conditional operator (‘`?`’) can be used in expressions of type `material`, `bsdf`, `edf`, or `vdf` under the restriction that the first operand is uniform.

Such conditional expressions can for example be used to implement a material switcher that, depending on a Boolean input parameter, selects one or the other material:

```
material switch_material( uniform bool condition,
                          material     m1,
                          material     m2)
  = condition ? m1 : m2;
```


14 Annotations

annotation_declaration : `annotation` *simple_name parameter_list* `;`

MDL defines a mechanism called annotations to attach metadata to various components of a material or function, as well as the material or function itself.

Annotations must be declared before they are used. An unknown or wrongly typed annotation shall issue a warning and will be suppressed.

An annotation is declared globally using the `annotation` keyword followed by the name of the annotation and optional annotation parameters surrounded in parenthesis. Annotation parameters can have default initializers with constant expressions as values.

For example:

```
annotation annotation_example( float f, string s = "");
```

This declares an annotation named `annotation_example` which accepts a `float` and a `string` parameter.

Annotations are a part of the module where they are defined and they share the name space with other MDL identifiers, such as functions.

An annotation can be declared multiple times, called *annotation overloading*, provided the overloads differ at least by the number or types of their parameters. This is similar to function overloading and overload resolution is defined for annotations analogously to function overload resolution in Section 12.4.

14.1 Annotation application

annotation : *qualified_name argument_list*

annotation_block : `[[` *annotation* { `,` *annotation* } (`]` `]` | `]` `]`)

Annotations can be applied to:

- Material definitions
- Function declarations
- Material and function parameters including function return values
- Structure types and their fields
- Enumeration types and their values
- Constants and variable declarations

Annotations are placed in blocks immediately after the declaration they are annotating, and before the opening curly brace for function definitions, structure type or enumeration type annotations. The comma-separated list of one or more annotations is enclosed in double brackets. An individual annotation looks

like a function call with no return value where the name of the function is the annotation and the arguments are values associated with the annotation. These values need to be constant expressions. The annotation syntax includes positional and named arguments explained for functions in Section 12. It is not an error to have several annotations of the same name.

An annotation block has the following form:

```
[[
    annotation_name(param1, param2, ...),
    annotation_name(param1, param2, ...),
    ...
]]
```

The `annotation_example` declared above can be used to annotate a material parameter as follows:

```
material material_example(
    color p(0)  [[ annotation_example( 1.5, s : "a string") ]]
)
  = material(...);
```

Annotations can be used to attach any type of metadata, though the common case is metadata to describe the user interface for a material or material parameter, see also the standard annotations in Section 18.

Note: The double brackets used for annotation blocks, `[[` and `]]`, are not tokens in MDL to avoid problems with a possible occurrence of `]]` for nested array accesses. Nonetheless, annotations require that the double brackets are not separated by any other characters including white spaces.

15 Modules

In MDL, all identifiers and all typenames live in modules with the exception of the types available as built-in reserved words. A module defines a namespace. Declarations inside a module need to be marked for export before they can be used outside of the module, and other modules need to import those declarations before they can be used.

A module corresponds one-to-one to a MDL source file. The name of the module is the name of the source file without the `.mdl` file extension. Note that this restricts the file names of modules that are referenced from other modules to legal MDL identifiers.

Modules can be used as such or they can be organized in packages. A package corresponds one-to-one to a directory containing module source files. Packages can be nested. The name of the package is the name of the directory. Note that this restricts the names of directories that are used as packages to legal MDL identifiers.

15.1 Import declarations

import : `import` *qualified_import* { `,` *qualified_import* } `;`
| [`export`] `using` *qualified_name*
`import` (`*` | *simple_name* { `,` *simple_name* }) `;`

qualified_import : [`::`] *simple_name* { `::` *simple_name* } [`::` `*`]

qualified_name : [`::`] *simple_name* { `::` *simple_name* }

simple_name : *IDENT*

A module can import individual or all declarations from other modules with import declarations, which must be specified after the version declaration (see Section 4.1) and before all other declarations.

Depending on the particular form of the import declarations used, the imported declarations can only be referred to by their qualified identifiers or by their unqualified identifiers. A qualified identifier is formed by the identifier of the imported module including its optional package name, the scope operator '`::`', and the unqualified identifier of the declaration. Declarations that can be referred to by their unqualified identifier can also be referred to by their qualified identifier.

The following table illustrates the four different forms of the import declaration. For the example code, it assumes a module `m` with three declarations `a`, `b`, and `c`, and a module `n` with a declaration `d`. For the sake of the illustration here, those declarations can be for example materials or functions.

Import declaration	Accessible declarations	Description
`import m::*, n::*;`	`m::a`, `n::b`, `m::c`, `n::d`	Qualified import of all declarations from the specified modules.
`import m::a, m::b, n::d;`	`m::a`, `m::b`, `n::d`	Qualified import of selected declarations only. Other declarations from the specified modules are inaccessible unless imported through an additional import declaration.
`using m import *;`	`a`, `b`, `c`, `m::a`, `m::b`, `m::c`	Unqualified import of all declarations from a module. Only one module can be specified in an unqualified import declaration.
`using m import a, b;`	`a`, `b`, `m::a`, `m::b`	Unqualified import of selected declarations only. Only one module can be specified in an unqualified import declaration.

It is not an error if the same declaration is imported more than once.

The following example illustrates how a qualified import of all declarations is combined with the unqualified import of selected declarations using the same module `m` from above:

```
import m::*;
using m import a, b;
// accessible declarations are: a, b, m::a, m::b, m::c
```

It is an error if a declaration of the same name is imported in the unqualified form from two different modules.

A module must not import itself and the import-graph of all modules in a system must not contain any cycles.

The module name can be optionally preceded by a sequence of package names separated by the scope operator '`::`', which forms a *relative package path*. Adding the scope operator '`::`' to the beginning of a relative package path changes it to an *absolute package path*.

Qualified identifiers from modules in packages require the qualification with all packages names and the module name as given in the import declaration, all separated by the scope operator '`::`'.

Package and module names live in a separate namespace. In case of ambiguities other declarations are preferred over a module.

Relative and absolute package paths are translated to relative and absolute file paths, respectively, by changing the scope operator '`::`' to the forward slash '`/`' and appending the `.mdl` file extension.

Modules are located in the file system using the file path search on their respective file path as defined in Section 2.2.

15.2 Export declarations

mdl : *mdl_version* {*import*} {[`export`] *global_declaration*}

import : `import` *qualified_import* { `,` *qualified_import* } `;`
| [`export`] `using` *qualified_name*
`import` (`*` | *simple_name* { `,` *simple_name* }) `;`

All declarations of a module are private to that module unless they are prefixed with the `export` qualifier.

For example:

```
int private_function_example() {    // not visible outside of module
    return 1;
}

export int function_export_example() {  // visible outside of module
    return private_function_example();
}
```

For function overloads, either all overloaded versions have an `export` qualifier or none.

Parameter types and identifiers in default initializers of parameters of exported functions must be exported as well. This ensures their accessibility in modules importing such functions. However, a module importing such functions may not necessarily need to import all types, unless it uses those parameters and does not rely on their defaults, and it may not need to import identifiers of default initializers, unless it uses them explicitly. Furthermore, it is sufficient if such types and identifiers are exported by some module, which must not be the same module as the one exporting the function. This is necessary for identifiers that are imported in their qualified form from another module and cannot be re-exported.

Return types of exported functions must also be exported by some module (or be one of the builtin types). Modules importing a function also need to import its return type if they call the function or use the type otherwise.

Imported declarations can be exported again if they are imported in their unqualified form. The unqualified imported declaration is part of the importing modules namespace. Exporting it makes it only visible from this namespace and not from its originating modules namespace.

For example a module `p` exports two declarations from the module `m`, which is defined as above:

```
// module p
export using m import a, b;
```

Another module can now import `p::a` and `p::b` without knowing about module `m`. For example:

```
// module example
using p import a;
import p::b;
// accessible declarations are: a, p::a, p::b
// while m::a and m::b remain inaccessible
```

A module can import the same declaration from different modules without problems:

```
// module example
import m::a, p::a;
// accessible declarations are: m::a, p::a
```

An export of an imported declaration does not create a new declaration. This is relevant for types and their use to distinguish among function overloads, where an export of an imported type does not constitute a new type. In continuation of the previous example with the assumption that `m::a` is a type, the following function overloads will cause an error because `m::a` is not a different type than `p::a`:

```
int function_overload( m::a parameter);
int function_overload( p::a parameter); // ! error
```

It further remains an error to import the same declaration in its unqualified form from two different modules even though one is only an export of the other. The following example illustrates this error:

```
// module example with error
using m import a;
using p import a; // ! error because a is imported a second time in
                  //   unqualified form from a different module
```

15.3 Interaction with function overloads

If a module imports an identifier referencing a function, all overloads of that function are imported. If the identifier is imported in its unqualified form, the module may add further overloads of that function. If the identifier is exported from the importing module, all overloads from the imported module as well as the added overloads must be exported.

Adding overloads to an imported function never alters the behavior of the imported overloads, or any other imported function. For example, if an imported function `a` calls another function `b`, the behavior of `a` cannot be changed by adding additional overloads to `b`.

15.4 Interoperability of modules of different language versions

Modules that are implemented using a different language version of MDL can nonetheless be freely mixed with the following restrictions:

1. If a module imports declarations from another module, both modules need to use a language version with the same major version number.
2. If a module imports declarations from another module, those declarations must be legal in the language version of the importing module. In particular, the name of the declared entity, its signature, parameter names and default initializer must all be legal.

The second restriction enables forward and backwards compatibility to the extent possible. As an example, the standard modules of MDL 1.1 contain extensions, like a new distribution function, that cannot be imported into a MDL 1.0 module while the other declarations that came from MDL 1.0 continue to be available for MDL 1.0 module imports.

15.5 Package structure with lead module

As an example, the module system with its package search allows the packaging of materials and functions in a library of implementation modules underneath a package directory and a lead module of the same name as the package at the same level as the package directory. The lead module can import all publicly relevant declarations from the implementation modules and export them.

The directory and module structure for a module and package `m` with implementation modules `i1`, `i2`, and `i3` would look like this:

```
<search-path-root>
  +--- m.mdl
  +--- m
       +--- i1.mdl
       +--- i2.mdl
       +--- i3.mdl
```

An example for the lead module `m` in the file `m.mdl` would be:

```
// lead module m
export using m::i1 import public_material_1;
export using m::i2 import public_material_2;
export using m::i3 import public_material_3;
```


16 Standard modules

MDL requires the following set of standard modules to be available. An import declarative with a standard module name shall always refer to these standard modules. In the case of modules with the same name, standard modules shall always be found before user-defined modules.

`std`	imports all other standard modules and re-exports all declarations.	
`limits`	standard global constants for MDL limits.	cf. Section 17
`anno`	standard annotations.	cf. Section 18
`state`	standard renderer state functions.	cf. Section 19
`math`	standard library math functions and constants.	cf. Section 20.1
`tex`	standard library texture functions.	cf. Section 20.3
`df`	standard elemental distribution functions, modifiers, and combiners.	cf. Section 21

The standard `std` module allows to import all standard modules at once and use their declarations with in the `std` scope. For example:

```
import std::*;

float stdlib_example() {
    return std::sin( std::HALF_PI);
}
```


17 Standard limits

The following standard global constants for some limits in MDL are made available through the standard `limits` module (Section 16). For example, the `limits` module can be imported as any other module and all the regular scoping and shadowing rules apply:

```
import limits::*;
```

`const float FLOAT_MIN`	The smallest positive normalized `float` value supported by the current platform.
`const float FLOAT_MAX`	The largest `float` value supported by the current platform.
`const double DOUBLE_MIN`	The smallest positive normalized `double` value supported by the current platform.
`const double DOUBLE_MAX`	The largest `double` value supported by the current platform.
`const int INT_MIN`	The smallest `int` value supported by the current platform.
`const int INT_MAX`	The largest `int` value supported by the current platform.
`const float WAVELENGTH_MIN`	The smallest `float` value that the current platform allows for representing wavelengths for the `color` type and its related functions.
`const float WAVELENGTH_MAX`	The largest `float` value that the current platform allows for representing wavelengths for the `color` type and its related functions.

18 Standard annotations

MDL defines a set of standard annotations to attach metadata to various components of a material or function, as well as the material or function itself.

Annotations need to be declared before they can be used, see Section 14. The standard annotations are declared in the standard `anno` module (Section 16). For example, the `anno` module can be imported as any other module and all the regular scoping and shadowing rules apply:

```
import anno::*;
```

The following are the provided standard annotations to represent common metadata:

`soft_range(int  min, int  max)`
`soft_range(int2 min, int2 max)`
`soft_range(int3 min, int3 max)`
`soft_range(int4 min, int4 max)`
`soft_range(float  min, float  max)`
`soft_range(float2 min, float2 max)`
`soft_range(float3 min, float3 max)`
`soft_range(float4 min, float4 max)`
`soft_range(double  min, double  max)`
`soft_range(double2 min, double2 max)`
`soft_range(double3 min, double3 max)`
`soft_range(double4 min, double4 max)`
`soft_range(color min, color max)`
	Specifies a range of useful values for the parameter, however the parameter value can exceed this range. Vector values are compared component-wise.
	
	`min` – The minimum value of the range.
	
	`max` – The maximum value of the range.

`hard_range(int  min, int  max)`
`hard_range(int2 min, int2 max)`
`hard_range(int3 min, int3 max)`
`hard_range(int4 min, int4 max)`
`hard_range(float  min, float  max)`
`hard_range(float2 min, float2 max)`
`hard_range(float3 min, float3 max)`
`hard_range(float4 min, float4 max)`
`hard_range(double  min, double  max)`
`hard_range(double2 min, double2 max)`
`hard_range(double3 min, double3 max)`
`hard_range(double4 min, double4 max)`
`hard_range(color min, color max)`
	Specifies bounds for the parameter that cannot be exceeded. Vector values are compared component-wise.
	
	`min` – The minimum value of the range.
	
	`max` – The maximum value of the range.

`display_name( string name)`	Specifies a name to use when the element is displayed in a user interface.

`in_group(`
`  string group)`
`in_group(`
`  string group,`
`  string subgroup)`
`in_group(`
`  string group,`
`  string subgroup,`
`  string subsubgroup)`
	Specifies that a material, function, or parameter is in a group or nested subgroup for presentational purposes in a user interface.

`hidden()`	Specifies that the element should not be visible in an application's user interface.
`description(` `  string description)`	Specifies a description of the element. An application providing a user interface for the element can use this text to provide a more detailed description beyond the display name.
`author(` `  string name)`	Specifies the name of the material or function author. Multiple authors are specified with multiple annotations.
`contributor(` `  string name)`	Specifies the name of a contributing material or function author. Multiple contributors are specified with multiple annotations.
`copyright_notice(` `  string copyright)`	Specifies copyright information.

`created(`
`  int year,`
`  int month,`
`  int day,`
`  string notes)`
	Specifies the date on which a material or function was created along with a string to hold creation notes.

`modified(`
`  int year,`
`  int month,`
`  int day,`
`  string notes)`
	Specifies a date on which a material or function was modified along with a string to hold notes related to the modification.

`version_number(`
`  int major,`
`  int minor = 0,`
`  int branch = 0,`
`  int build = 0)`
	Specifies a hierarchical nested version number of a material or function. The parameter names recommend a conventional use of up to four numbers, not all of which are needed in all cases. Higher numbers shall indicate later versions given all numbers left of it are equal.

`key_words(` `  string[] words)`	Specifies a list of keywords related to a material or function. These keywords can be used to search libraries of materials or functions.

`unused()`
`unused(`
`  string description)`
	Specifies that the annotated element is not used in the implementation. This can be used to suppress compiler warnings about unused elements. The variant with the `description` text can be used to give an additional explanation why the element is not used, such as `"deprecated"` or `"reserved for future use"`, which may be shown in user interfaces.

19 Renderer state

Materials and functions need access to some common values available in the renderer, such as some varying aspect of the current fragment being shaded or a uniform property of the render pass. MDL gives access to these values through a set of global functions, the *renderer state functions*, or *state functions* for short.

Note: The renderer state is thus immutable and cannot be changed in a function or material. However, some material properties influence the renderer state of later stages, see Section 13 for details.

Although the renderer state cannot be defined in MDL itself, it is made available through one of the standard MDL modules, see Section 16. The `state` module can be imported as any other module, for example with:

```
import state::*;
```

All the regular scoping and shadowing rules apply. For example, this allows to access the `normal` state function using its fully qualified name in case it has been shadowed as in the following example:

```
material normal_parameter_example( float3 normal = state::normal())
  = material( ...);
```

MDL materials and functions are used in the renderer in different contexts, for example, when rendering a surface or when rendering a volume. Not all state functions are well defined in all contexts. Yet, in order to enable generic functions, for example, for texture lookups on surfaces as well as for environments, all state functions are always defined and return sensible defaults in such occasions. The defaults are documented in detail below.

Most state functions are varying and only a few are uniform such as `transform(...)`. All state functions are documented with their explicit uniform or varying property below.

19.1 Basic renderer state values

The following is a list of all basic renderer state values. Note that all spatial values are in internal space, see 19.2 for coordinate space transformation functions.

`float3 position() varying`	The intersection point on the surface or the sample point in the volume. The default value is zero for contexts without a position, such as the environment.
`float3 normal() varying`	The shading surface normal as a unit-length vector. The value is a unit-length vector facing to the origin if the context is the environment. The default value is zero for contexts without a normal, such as volumes.
`float3 geometry_normal() varying`	The true geometric surface normal for the current geometry as a unit-length vector. The value is a unit-length vector facing to the origin if the context is the environment. The default value is zero for contexts without a normal, such as volumes.
`float3 motion() varying`	Tangential motion vector.

`int texture_space_max() varying`	The maximal number of texture spaces available.

`float3 texture_coordinate(`
`  int index`
`) varying`
	The texture space at the given index where $0 \leq$ *index* $<$ *texture_space_max*(). An index outside this range yields undefined behavior.

`float3 texture_tangent_u(`
`  int index`
`) varying`
	The array of tangent vectors for each texture space. The tangent vector is a unit length vector in the plane defined by the surface normal, which points in the direction of the projection of the tangent to the positive u axis of the corresponding texture space onto the plane defined by the original surface normal.

`float3 texture_tangent_v(`
`  int index`
`) varying`
	The array of bitangent vectors for each texture space. The bitangent vector is a unit length vector in the plane defined by the surface normal, which points in the general direction of the positive `v` axis of the corresponding texture space, but is orthogonal to both the original surface normal and the tangent of the corresponding texture space.

`float3x3 tangent_space(`
`  int index`
`) varying`
	The array of tangent space matrices for each texture space. These matrices are available as a convenience and are constructed from the `texture_tangent_u`, `texture_tangent_v`, and surface normal as the `x`, `y`, and `z` axis of the coordinate system, respectively.

`float3 geometry_tangent_u(`
`  int index`
`) varying`
	Array of geometry tangents. Together with `geometry_normal` and `geometry_tangent_v`, this forms a number of orthonormal bases. The orientation of each basis around the normal is the same as that of `tangent_space`.

`float3 geometry_tangent_v(`
`  int index`
`) varying`
	Array of geometry bitangents. Together with `geometry_normal` and `geometry_tangent_u`, this forms a number of orthonormal bases.

`int object_id() uniform`	Returns the object ID provided in a scene, and zero if none was given or for the environment.
`float3 direction() varying`	Lookup direction in the context of an environment lookup and `float3(0.0)` in all other contexts.
`float animation_time() varying`	The time of the current sample in seconds, including the time within a shutter interval.
`const int WAVELENGTH_BASE_MAX`	The number of wavelengths returned in the result of `wavelength_base()`.
`float[WAVELENGTH_BASE_MAX]` `wavelength_base() uniform`	Array of wavelengths, in increasing order, that are recommended when constructing spectra to achieve best approximation results (see Section 6.11.1). Wavelengths are given in nanometers [*nm*]. Each wavelength λ is between the shortest and longest wavelengths considered for spectra, defined by `limits::WAVELENGTH_MIN` $\leq \lambda \leq$ `limits::WAVELENGTH_MAX`.

19.2 Coordinate space transformations

The state provides functions to transform spatial values—namely scales, vectors, points and normals—between the following coordinate spaces:

- internal space
- object space
- world space

Spatial values returned by state functions are always provided in internal space unless noted otherwise. Internal space is implementation dependent and can vary across different platforms. If a material or function can perform calculations independently of the coordinate system then it can operate with those values directly, otherwise it will need to transform them into a known space.

```
enum coordinate_space {
    coordinate_internal,
    coordinate_object,
    coordinate_world
};
```

The coordinate space determines in which space coordinates and other spatial values are represented and in which space their representation is transformed into.

```
float4x4 transform(
    coordinate_space from,
    coordinate_space to
) uniform
```

Returns a transformation matrix to transform from one space to another. The `from` and `to` parameters can be any of the `coordinate_space` enumeration values.

The matrix returned from `transform()` assumes that vectors are considered to be column vectors and multiplied on the right-hand side of the matrix. In other words, the translation components T_x, T_y, and T_z of a translation-only transformation are located in the following positions of the returned matrix:

$$\begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

See also Section 6.9 for the other relevant conventions on matrices.

```
float3 transform_point(
    coordinate_space from,
    coordinate_space to,
    float3 point
) uniform
```

Transforms a point from one coordinate system to another. The `from` and `to` parameters can be any of the `coordinate_space` enumeration values.

```
float3 transform_vector(
    coordinate_space from,
    coordinate_space to,
    float3 vector
) uniform
```

Transforms a vector from one coordinate system to another. The translation component of the coordinate systems is ignored when transforming the vector. The `from` and `to` parameters can be any of the `coordinate_space` enumeration values.

`float3 transform_normal(`
`  coordinate_space from,`
`  coordinate_space to,`
`  float3 normal`
`) uniform`
	Transform a surface normal from one coordinate system to another. As with vector transformations, the translation component of the coordinate systems is ignored. Additionally the transpose of the inverse transformation matrix is used to properly transform the surface normal. The `from` and `to` parameters can be any of the `coordinate_space` enumeration values.

`float transform_scale(`
`  coordinate_space from,`
`  coordinate_space to,`
`  float scale`
`) uniform`
	Transforms a scale measure from one coordinate system to another. Only the uniform scale factor of the coordinate system transformation is used. If the transformation is not a similarity transform, the average of the non-uniform scale factors is used. The `from` and `to` parameters can be any of the `coordinate_space` enumeration values.

Note that for all transform functions, if the `from` argument is `coordinate_internal` then the coordinated to transform should be a return value of a state function or a value that his been previously transformed into internal space. Otherwise, the result is undefined and may differ across implementations.

For example the result of:

```
transform_point( coordinate_internal,
                 coordinate_world,
                 float3(1,0,0))      // ! undefined behavior
```

is undefined since the definition of the coordinates (1,0,0) in internal space is implementation dependent.

The world space is in *scene units*, which can vary from integration to integration and scene to scene. The following state functions offer conversion factors between scene units and meters:

`float meters_per_scene_unit() uniform`	Returns the distance of one scene unit in meters.
`float scene_units_per_meter() uniform`	Returns the distance of one meter in scene units.

19.3 Rounded corners state function

```
float3 rounded_corner_normal(
    uniform float radius          = 0.0,
    uniform bool  across_materials = false
) varying
```

Returns a modified shading normal `state::normal` that is changed near surface mesh edges to blend smoothly into the shading normal of neighboring facets. The result of this function can be used to set the `normal` field of the `material_geometry` structure, which results in a perceived look of smooth edges instead of a facetted look with hard edges. It can as well be combined with normal perturbation schemes.

`radius` – influence radius around the edges within which the normal is modified. The radius is specified in meters in world space. A radius of `0.0` disables the rounded corners and this function just returns `state::normal`.

`across_materials` – normal smoothing happens only between facets of equal material, unless this parameter is set `true`, in which case the smoothing happens between facets irrespective of their materials.

20 Standard library functions

MDL defines a standard library of built-in functions and a few constants. Many of these functions are overloaded to support different MDL scalar and vector data types. A list of overloaded versions of each standard library function is included with each description. The following generic types are used in these function signatures to abbreviate common overloads of vector types and floating-point types.

Generic type	Represents one of these types
boolN	`bool`, `bool2`, `bool3`, `bool4`
intN	`int`, `int2`, `int3`, `int4`
float	`float`, `double`
floatN	`float`, `float2`, `float3`, `float4`, `double`, `double2`, `double3`, `double4`

For any specific function signature, if more than one generic type is used to represent a vector type, all of those vector types need to have the same number of fields, and if more than one generic type is used to represent a floating-point type, all of those types need to be of single precision or double precision.

The vector versions of the standard library functions operate component-wise unless noted otherwise. The description is per component.

The standard library functions and constants are made available through the standard MDL modules, see Section 16 and the individual sections below. For example, the `math` module can be imported as any other module:

```
import math::*;
```

All the regular scoping and shadowing rules apply. For example, this allows to access the `abs` standard math function using its fully qualified name in case it has been in shadowed as in the following example:

```
float stdlib_shadowing_example( float max) {
    return math::max( 1.0, max);
}
```

20.1 Math constants

The following standard math constants are made available through the standard `math` module (Section 16):

```
import math::*;
```

`const float PI`	`= 3.14159265358979323846f`
`const float TWO_PI`	`= 6.28318530717958647692f`
`const float HALF_PI`	`= 1.57079632679489661923f`

20.2 Math functions

The following standard math functions, including some geometric and color related functions, are made available through the standard `math` module (Section 16):

```
import math::*;
```

All standard math functions are uniform and not documented as such explicitly below.

intN `abs(` *intN* `a)` *floatN* `abs(` *floatN* `a)` `color abs( color a)`	Returns the absolute value.
floatN `acos(` *floatN* `a)`	Returns the arc cosine.
`bool all(` *boolN* `a)`	Returns `true` if all components are `true` and `false` otherwise.
`bool any(` *boolN* `a)`	Returns `true` if any component is `true` and `false` otherwise.
floatN `asin(` *floatN* `a)`	Returns the arc sine.
floatN `atan(` *floatN* `a)`	Returns the arc tangent.
floatN `atan2(` *floatN* `y,` *floatN* `x)`	Returns the arc tangent of `y/x`. The signs of `y` and `x` are used to determine the quadrant of the result.
float `average(` *floatN* `a)` `float average( color a)`	Returns the average of the vector elements or the color.
`color blackbody( float temperature)`	Returns the color for a blackbody radiator at the given temperature in Kelvin.
floatN `ceil(` *floatN* `a)`	Returns the nearest integer that is greater than or equal to the value.

intN `clamp(` *intN* `a,` *intN* `min,` *intN* `max)`
intN `clamp(` *intN* `a, int min,` *intN* `max)`
intN `clamp(` *intN* `a,` *intN* `min, int max)`
intN `clamp(` *intN* `a, int min, int max)`
floatN `clamp(` *floatN* `a,` *floatN* `min,` *floatN* `max)`
floatN `clamp(` *floatN* `a,` *float* `min,` *floatN* `max)`
floatN `clamp(` *floatN* `a,` *floatN* `min,` *float* `max)`
floatN `clamp(` *floatN* `a,` *float* `min,` *float* `max)`
`color clamp( color a, color min, color max)`
`color clamp( color a, float min, color max)`
`color clamp( color a, color min, float max)`
`color clamp( color a, float min, float max)`
	Returns `min` if `a` is less than `min`, `max` if `a` is greater than `max`, and `a` otherwise.

floatN `cos(` *floatN* `a)`	Returns the cosine. Angles specified by `a` are in radians.
`float3 cross( float3 a, float3 b)`	Returns the cross product of `a` and `b`.
floatN `degrees(` *floatN* `a)`	Converts the value from radians to degrees, i.e, returns $180 \cdot a/\pi$.
float `distance(` *floatN* `a,` *floatN* `b)`	Returns the Euclidean distance between `a` and `b`.

float `dot(` *floatN* `a,` *floatN* `b)`	Returns the dot product of `a` and `b`.

`float eval_at_wavelength(`
`  color a,`
`  float wavelength)`
	Evaluates and returns the value of `a` at the given wavelength, where the wavelength argument is given in nanometers [*nm*] and `limits::WAVELENGTH_MIN` $\leq$ `wavelength` $\leq$ `limits::WAVELENGTH_MAX`.

floatN `exp(` *floatN* `a)` `color exp( color a)`	Returns the constant *e* raised to the power `a`.
floatN `exp2(` *floatN* `a)` `color exp2( color a)`	Returns the value two raised to the power `a`.
floatN `floor(` *floatN* `a)`	Returns the nearest integer that is less than or equal to the value.
floatN `fmod(` *floatN* `a,` *floatN* `b)` *floatN* `fmod(` *floatN* `a, float b)`	Returns `a` modulo `b`, in other words, the remainder of `a/b`. The result has the same sign as `a`.
floatN `frac(` *floatN* `a)`	Returns the positive fractional part.
boolN `isnan(` *floatN* `a)`	Returns `true` if the value does not represent a valid number and `false` otherwise. This can occur as the result of an invalid operation such as taking the square root of a negative number.
boolN `isfinite(` *floatN* `a)`	Returns `true` if the value represents a valid and finite number, and `false` otherwise.
float `length(` *floatN* `a)`	Returns the length of `a`.

floatN `lerp(` *floatN* `a,` *floatN* `b,` *floatN* `l)`
floatN `lerp(` *floatN* `a,` *floatN* `b,` *float* `l)`
`color lerp( color a, color b, color l)`
`color lerp( color a, color b, float l)`
	Returns the linear interpolation between `a` and `b` based on the value of `l`, such that the result is $\mathtt{a} \cdot (1 - \mathtt{l}) + \mathtt{b} \cdot \mathtt{l}$.

floatN `log(` *floatN* `a)` `color log( color x)`	Computes the natural logarithm.
floatN `log2(` *floatN* `a)` `color log2( color x)`	Computes the base two logarithm.
`float luminance( float3 a)` `float luminance( color a)`	Returns the Y channel (luminance) of `a` when interpreting `a` in the CIE XYZ color space. The color space of `a` is implementation dependent if `a` is of type `color` and assumes the linear sRGB color space if `a` is of type `float3`. In the latter case, the luminance is then equal to $0.212671 \cdot \mathtt{a.x} + 0.715160 \cdot \mathtt{a.y} + 0.072169 \cdot \mathtt{a.z}$.

intN max(*intN* a, *intN* b) *intN* max(int a, *intN* b) *intN* max(*intN* a, int b) *floatN* max(*floatN* a, *floatN* b) *floatN* max(*float* a, *floatN* b) *floatN* max(*floatN* a, *float* b) color max(color a, color b) color max(*float* a, color b) color max(color a, *float* b)	Returns the maximum of a and b.
float max_value(*floatN* a) float max_value(color a)	Returns the largest value of a.
float max_value_wavelength(color a)	Returns the smallest wavelength in [*nm*] at which the largest value lies.
intN min(*intN* a, *intN* b) *intN* min(int a, *intN* b) *intN* min(*intN* a, int b) *floatN* min(*floatN* a, *floatN* b) *floatN* min(*float* a, *floatN* b) *floatN* min(*floatN* a, *float* b) color min(color a, color b) color min(*float* a, color b) color min(color a, *float* b)	Returns the minimum of a and b.
float min_value(*floatN* a) float min_value(color a)	Returns the smallest value of a.
float min_value_wavelength(color a)	Returns the smallest wavelength in [*nm*] at which the smallest value lies.
floatN[2] modf(*floatN* a)	Returns an array with the integral part of a as first element and the fractional part of a as second element. Both the fractional and integer parts have the same sign as a.
floatN normalize(*floatN* a)	Scales a by the reciprocal of its length to give it a length of 1. If the length of a is zero the result of normalize(a) is undefined.
intN pow(*intN* a, *intN* b) *intN* pow(*intN* a, int b) *floatN* pow(*floatN* a, *floatN* b) *floatN* pow(*floatN* a, *float* b) color pow(color a, color b) color pow(color a, float b)	Returns a raised to the power b. Floating-point value a must not be negative, while for the overloaded functions with integer exponent b, the integer value a may be negative.
floatN radians(*floatN* a)	Converts the value from degrees to radians, i.e, returns $\pi \cdot a/180$.
floatN round(*floatN* a)	Returns the nearest integer value for a.
floatN rsqrt(*floatN* a) color rsqrt(color a)	Returns the reciprocal of the square root of a.

floatN `saturate(` *floatN* `a)` `color saturate( color a)`	Clamps `a` so that $0.0 \leq a \leq 1.0$.
intN `sign(` *intN* `a)` *floatN* `sign(` *floatN* `a)`	Returns 1 if `a` is greater than 0, -1 if `a` is less than 0, and 0 otherwise.
floatN `sin(` *floatN* `a)`	Returns the sine of `a`. Angles specified by `a` are in radians.
floatN`[2] sincos(` *floatN* `a)`	Returns an array with the sine of `a` as first element and the cosine of `a` as second element. Angles specified by `a` are in radians.
floatN `smoothstep(` *floatN* `a,` *floatN* `b,` *floatN* `l)` *floatN* `smoothstep(` *floatN* `a,` *floatN* `b,` *float* `l)`	Returns 0 if `l` is less than `a` and 1 if `l` is greater than `b`. A smooth curve is applied in-between so that the return value varies continuously from 0 to 1 as `l` varies from `a` to `b`.
floatN `sqrt(` *floatN* `a)` `color sqrt( color a)`	Returns the square root of `a`
floatN `step(` *floatN* `a,` *floatN* `b)`	Returns 0 if `b` is less than `a` and 1 otherwise.
floatN `tan(` *floatN* `a)`	Returns the tangent of `a`. Angles specified by `a` are in radians.

`float2x2 transpose( float2x2 a)`
`float2x3 transpose( float3x2 a)`
`float3x2 transpose( float2x3 a)`
`float3x3 transpose( float3x3 a)`
`float4x2 transpose( float2x4 a)`
`float2x4 transpose( float4x2 a)`
`float3x4 transpose( float4x3 a)`
`float4x3 transpose( float3x4 a)`
`float4x4 transpose( float4x4 a)`
`double2x2 transpose( double2x2 a)`
`double2x3 transpose( double3x2 a)`
`double3x2 transpose( double2x3 a)`
`double3x3 transpose( double3x3 a)`
`double4x2 transpose( double2x4 a)`
`double2x4 transpose( double4x2 a)`
`double3x4 transpose( double4x3 a)`
`double4x3 transpose( double3x4 a)`
`double4x4 transpose( double4x4 a)`
	Computes the transpose of the matrix `a`.

20.3 Texture

The following standard texture functions are made available through the standard `tex` module (Section 16):

```
import tex::*;
```

The uniform or varying modifier for the standard texture functions is documented explicitly.

`int width( uniform texture_2d tex) uniform`
`int width( uniform texture_3d tex) uniform`
`int width( uniform texture_cube tex) uniform`
	The width of the texture `tex` in pixels along the u-direction in texture space. The width is zero for an invalid texture reference.

`int height( uniform texture_2d tex) uniform`
`int height( uniform texture_3d tex) uniform`
`int height( uniform texture_cube tex) uniform`
	The height of the texture `tex` in pixels along the v-direction in texture space. The height is zero for an invalid texture reference.

`int depth( uniform texture_3d tex) uniform`	The depth of the texture `tex` in pixels along the w-direction in texture space. The depth is zero for an invalid texture reference.

`bool texture_isvalid( uniform texture_2d tex) uniform`
`bool texture_isvalid( uniform texture_3d tex) uniform`
`bool texture_isvalid( uniform texture_cube tex) uniform`
`bool texture_isvalid( uniform texture_ptex tex) uniform`
	Returns `true` if the value of `tex` is a valid texture reference and `false` otherwise.

The standard texture lookup functions are provided in different variants, which differ only in the lookup type *ltype*, which is used as part of the function name and which determines the return type of the lookup function. The generic lookup type *ltype* can be any of the following types:

Generic type	Represents one of these types
ltype	`float`, `float2`, `float3`, `float4`, `color`

The standard texture lookup function for the `texture_ptex` is varying and the other standard texture functions are uniform. Note that their result is usually varying because of the use of a varying state value for their lookup coordinate parameter. All functions are documented accordingly with their explicit uniform or varying property below.

`enum gamma_mode {`
`  gamma_default,`
`  gamma_linear,`
`  gamma_srgb`
`};`
	The gamma mode determines whether a texture can be used as is in the linear workflow of the renderer or if it needs to be inverse-gamma corrected. The value `gamma_default` leaves this decision up to the texture itself. The value `gamma_linear` defines that the texture can be used as is. The value `gamma_srgb` defines that the texture uses the gamma compression of the sRGB standard, which is close to a gamma factor of 2.2, and the texture needs to be inverse-gamma corrected by the integration before use.
	
	Note: This mode is only used in the texture constructors that create a texture from a file path as defined in Section 6.12.1.

`enum wrap_mode {`
`  wrap_clamp,`
`  wrap_repeat,`
`  wrap_mirrored_repeat,`
`  wrap_clip`
`};`
	The wrap mode determines the texture lookup behavior if a lookup coordinate is exceeding the normalized half-open texture space range of $[0, 1)$: `wrap_clamp` clamps the lookup coordinate to the range, `wrap_repeat` takes the fractional part of the lookup coordinate effectively repeating the texture along this coordinate axis, `wrap_mirrored_repeat` is like `wrap_repeat` but takes one minus the fractional part every other interval to mirror every second instance of the texture, and `wrap_clip` makes the texture lookup return zero for texture coordinates outside of the range.

ltype `lookup_`*ltype*`(`
`  uniform texture_2d tex,`
`  float2 coord,`
`  uniform wrap_mode wrap_u = wrap_repeat,`
`  uniform wrap_mode wrap_v = wrap_repeat,`
`  uniform float2 crop_u = float2(0.0, 1.0),`
`  uniform float2 crop_v = float2(0.0, 1.0)`
`) uniform`
	Returns the sampled texture value for the two-dimensional coordinates `coord` given in normalized texture space in the range $[0,1)^2$, where the wrap modes define the behavior for coordinates outside of that range. The crop parameters further define a sub-range on the texture that is actually used and that defines the normalized texture space in the range $[0,1)^2$. The crop parameter defaults `float2(0.0, 1.0)` corresponds to the whole texture in the corresponding axis. A lookup on an invalid texture reference returns zero.

ltype `lookup_`*ltype*`(`
`  uniform texture_3d tex,`
`  float3 coord,`
`  uniform wrap_mode wrap_u = wrap_repeat,`
`  uniform wrap_mode wrap_v = wrap_repeat,`
`  uniform wrap_mode wrap_w = wrap_repeat,`
`  uniform float2 crop_u = float2(0.0, 1.0),`
`  uniform float2 crop_v = float2(0.0, 1.0),`
`  uniform float2 crop_w = float2(0.0, 1.0)`
`) uniform`
	Returns the sampled texture value for the three-dimensional coordinates `coord` given in normalized texture space in the range $[0,1)^3$, where the wrap modes define the behavior for coordinates outside of that range. The crop parameters further define a sub-range on the texture that is actually used and that defines the normalized texture space in the range $[0,1)^3$. The crop parameter defaults `float2(0.0, 1.0)` corresponds to the whole texture in the corresponding axis. A lookup on an invalid texture reference returns zero.

ltype `lookup_`*ltype*`(`
`  uniform texture_cube tex`
`  float3 coord`
`) uniform`
	Returns the sampled texture value for a cube texture lookup in the direction given by the three-dimensional vector `coord`. A lookup on an invalid texture reference returns zero.

ltype `lookup_`*ltype*`(`
`  uniform texture_ptex tex,`
`  int channel = 0`
`) varying`
	Returns the sampled PTEX texture value for the current surface position starting at the channel provided with the `channel` parameter. A lookup on an invalid texture reference or beyond available channels returns zero.

ltype `texel_`*ltype*`(`
`  uniform texture_2d tex,`
`  int2 coord`
`) uniform`
	Returns the raw texture value for the two-dimensional coordinates `coord` in the valid range $[0, \texttt{width(tex)}-1] \times [0, \texttt{height(tex)}-1]$. A lookup on an invalid texture reference or outside the valid range returns zero.

21 Standard distribution functions

MDL defines a standard set of elemental distribution functions, modifiers, and combiners that are used in defining materials in Section 13. The distribution functions define light interaction at a boundary (reflection and transmission), light scattering in participating media, and the emission of light from a surface.

The distributions, modifiers, and combiners are made available in MDL programs by loading the `df` module, in the same manner as the modules described in Section 15:

```
import df::*;
```

The regular scoping and shadowing rules apply to all symbols imported from the `df` module. However, unlike the other modules, the `df` module is not written in MDL; it requires a tight coupling with the rendering system that is outside the scope of MDL itself.

21.1 Bidirectional scattering distribution functions

This section documents the elemental *bidirectional scattering distribution function* (BSDF) models defined in MDL and their input parameters. Defining an instance of these BSDF models uses the syntax of a function returning a `bsdf`, which is not allowed elsewhere in MDL.

Inputs have to meet some requirements to ensure energy conservation. Unless noted otherwise, the main criterion is for color inputs to be in $[0, 1]$. In some cases models have to enforce additional restrictions. Two principal approaches are possible when normalizing color inputs. First, colors may be clamped, that is, values outside of $[0, 1]$ are set to the boundary values. Second, colors may be divided by their largest component if it is larger than one. The former approach is slightly faster, while the latter avoids color shifts.

Two BSDFs (`specular_bsdf` and `simple_glossy_bsdf`) require the definition of their *scattering mode*, represented by an enum parameter of the BSDF:

```
enum scatter_mode {
    scatter_reflect,
    scatter_transmit,
    scatter_reflect_transmit
};
```

Specular modes for reflection, transmission, or both.

21.1.1 Diffuse interaction

```
bsdf diffuse_reflection_bsdf (
  color          tint      = color(1.0),
  float          roughness = 0.0,
  uniform string handle    = ""
);
```

Lambertian reflection extended by the Oren-Nayar microfacet model.

`tint` – Scaling factor, defined as a color, multiplied by the result of the distribution function.

`roughness` – Oren-Nayar roughness coefficient, simulating view-dependent diffuse reflection. Range: $[0, 1]$, with `0` specifying complete view independence.

`handle` – Name to provide access to this component for use in an MDL integration.

```
bsdf diffuse_transmission_bsdf(
  color          tint   = color(1.0),
  uniform string handle = ""
);
```

Pure diffuse transmission of light through a surface.

`tint` – Scaling factor, defined as a color, multiplied by the result of the distribution function.

`handle` – Name to provide access to this component for use in an MDL integration.

21.1.2 Specular interaction

Specular reflections and transmissions implement an idealized surface in which light is reflected in the mirror direction or is transmitted based on the index of refraction of the boundary using Snell's law.

```
bsdf specular_bsdf(
  color                tint   = color(1.0),
  uniform scatter_mode mode   = scatter_reflect,
  uniform string       handle = ""
);
```

Pure specular reflection and/or transmission. Uses the reflection color in transmission mode if the ior values indicate total interior reflection.

`tint` – Scaling factor, defined as a color, multiplied by the result of the distribution function.

`mode` – One of three values: `reflect`, `transmit`, or (for both) `reflect_transmit`.

`handle` – Name to provide access to this component for use in an MDL integration.

21.1.3 Glossy interaction

Glossy reflections can be anisotropic and are therefore dependent upon the orientation of the surface specified by its local coordinate system. The following two BSDFs can override this coordinate system by

specifying the tangent in the u direction.

```
bsdf simple_glossy_bsdf(
  float                roughness_u,
  float                roughness_v  = roughness_u,
  color                tint         = color(1.0),
  float3               tangent_u    = state::texture_tangent_u(0),
  uniform scatter_mode mode         = scatter_reflect,
  uniform string       handle       = ""
);
```

Glossy reflection and transmission based on a microfacet model. Becomes black in transmission mode if the ior values indicate total interior reflection.

`roughness_u` – Roughness coefficient in the u direction. Range: $[0, \infty)$, with `0` specifying pure specular reflection.

`roughness_v` – Roughness coefficient in the v direction. Range: $[0, \infty)$, with `0` specifying pure specular reflection.

`tint` – Scaling factor, defined as a color, multiplied by the result of the distribution function.

`tangent_u` – The tangent in the u direction, in internal space.

`mode` – One of three values: `reflect`, `transmit`, or (for both) `reflect_transmit`.

`handle` – Name to provide access to this component for use in an MDL integration.

```
bsdf backscattering_glossy_reflection_bsdf(
  float          roughness_u,
  float          roughness_v  = roughness_u,
  color          tint         = color(1.0),
  float3         tangent_u    = state::texture_tangent_u(0),
  uniform string handle       = ""
);
```

Backscattering glossy reflection.

`roughness_u` – Roughness coefficient in the u direction. Range: $[0, \infty)$, with `0` specifying pure specular reflection.

`roughness_v` – Roughness coefficient in the v direction. Range: $[0, \infty)$, with `0` specifying pure specular reflection.

`tint` – Scaling factor, defined as a color, multiplied by the result of the distribution function.

`tangent_u` – The tangent in the u direction, in internal space.

`handle` – Name to provide access to this component for use in an MDL integration.

21.1.4 Measured interaction

Measured BSDF data can contain the measured information for the surface reflection behavior, the surface transmission behavior, or both. The following BSDF selects those depending on the additional `mode` parameter. It is an error if the `mode` parameter selects a mode for which no data is provided in the measurement.

`bool bsdf_measurement_isvalid(`
`  uniform bsdf_measurement measurement`
`) uniform`
	Returns `true` if the value of `measurement` is a valid BSDF measurement reference and `false` otherwise.

```
bsdf measured_bsdf(
  uniform bsdf_measurement measurement,
  uniform float            multiplier   = 1.0,
  uniform scatter_mode     mode         = scatter_reflect,
  uniform string           handle       = ""
);
```

General isotropic reflection and transmission based on measured data.

`measurement` – Measured BSDF data.

`multiplier` – Factor to scale the measurement. Scaling is limited to the maximum scale where an albedo of one is reached for a particular direction and larger factors will not scale the measurement any further.

`mode` – One of three values: `reflect`, `transmit`, or (for both) `reflect_transmit`.

`handle` – Name to provide access to this component for use in an MDL integration.

21.2 Light emission

This section documents the elemental *emission distribution function* (EDF) models defined in MDL and their input parameters.

Additionally, two supportive functions on light profiles are available in the `df` module.

`float light_profile_power(`
`  uniform light_profile profile`
`) uniform`
	Returns the power emitted by this light profile. A lookup on an invalid light profile reference returns zero.

`float light_profile_maximum(`
`  uniform light_profile profile`
`) uniform`
	Returns the maximum intensity in this light profile. A lookup on an invalid light profile reference returns zero.

`bool light_profile_isvalid(`
`  uniform light_profile profile`
`) uniform`
	Returns `true` if the value of `profile` is a valid light profile reference and `false` otherwise.

```
edf diffuse_edf(
  uniform string handle = ""
);
```

Uniform light emission in all directions.

`handle` – Name to provide access to this component for use in an MDL integration.

```
edf spot_edf(
  uniform float    exponent,
  uniform float    spread              = math::PI,
  uniform bool     global_distribution = true,
  uniform float3x3 global_frame        = float3x3(1.0),
  uniform string   handle              = ""
);
```

Exponentiated cosine weighting for spotlight. The spot light is oriented along the positive z-axis.

`exponent` – Exponent for cosine of angle between axis and sample point.

`spread` – angle of the cone to which the the cosine distribution is restricted. The hemispherical domain for the distribution is rescaled to this cone.

`global_distribution` – If `true`, the global coordinate system defines the orientation of light distribution. Otherwise, it is oriented along the local tangent space.

`global_frame` – Orthonormal coordinate system that defines the orientation of the light distribution with respect to the object space. In other words, multiplying a direction in the coordinate frame of the light distribution with this matrix transforms it into object space.

`handle` – Name to provide access to this component for use in an MDL integration.

```
edf measured_edf(
  uniform light_profile profile,
  uniform float         multiplier          = 1.0,
  uniform bool          global_distribution = true,
  uniform float3x3      global_frame        = float3x3(1.0),
  float3                tangent_u           = state::texture_tangent_u(0),
  uniform string        handle              = ""
);
```

Light distribution defined by a profile.

`profile` – Definition of light distribution.

`multiplier` – Factor to scale the light distribution intensity.

`global_distribution` – If `true`, the global coordinate system defines the orientation of light distribution. Otherwise, it is oriented along the local tangent space.

`global_frame` – Orthonormal coordinate system that defines the orientation of the light distribution with respect to the object space. In other words, multiplying a direction in the coordinate frame of the light distribution with this matrix transforms it into object space.

`tangent_u` – The tangent in the u direction, in internal space.

`handle` – Name to provide access to this component for use in an MDL integration.

21.3 Volume scattering

This section documents the elemental *volume distribution function* (VDF) models defined in MDL and their input parameters.

```
vdf anisotropic_vdf(
  float          directional_bias = 0.0,
  uniform string handle           = ""
);
```

Volume light distribution with directional bias.

`directional_bias` – Influence of light direction on scattering. Range: $[-1, 1]$, with `0` specifying isotropic behavior, 1 forward scattering, and -1 back scattering.

`handle` – Name to provide access to this component for use in an MDL integration.

21.4 Distribution function modifiers

Distribution function modifiers accept another distribution function as input parameter and change their behavior, such as changing its overall color, adding a directional-dependent thin-film effect, or attenuating them in a directional dependent way, to form a new distribution function.

```
bsdf tint(
   color         tint,
   bsdf          base
);
edf tint(
   uniform color tint,
   edf           base
);
```

Tint the result of an elemental or compound distribution function with an additional color.

`tint` – Scaling factor, defined as a color, multiplied by the result of the distribution function.

`base` – Input distribution function.

```
bsdf thin_film(
  float thickness,
  color ior,
  bsdf  base
);
```

Add reflective thin-film interference color to an elemental or compound BSDF.

`thickness` – Thickness of thin-film layer in nanometer [nm].

`ior` – Index of refraction.

`base` – Base BSDF.

```
bsdf directional_factor(
  color normal_tint  = color(1.0),
  color grazing_tint = color(1.0),
  float exponent     = 5.0,
  bsdf  base         = bsdf()
);
```

Directional modifier.

`normal_tint` – Color scaling factor at the normal.

`grazing_tint` – Color scaling factor at the grazing angle.

`exponent` – Exponent for directional factor. Default value (5.0) is from Schlick's approximation.

`base` – Base BSDF to be modified by directional factor.

```
bsdf measured_curve_factor(
  color[<N>] curve_values,
  bsdf       base          = bsdf()
);
```

Modifier weighting a base BSDF based on a measured reflection curve.

`curve_values` – Measured data for the reflection behavior. A 1-d function measured in the pre-image range from zero to Pi/2 with equally spaced measured reflectance values.

`base` – Base BSDF to be modified by the measured reflectance curve.

21.5 Distribution function combiners

Distribution function combiners accept one or more distribution functions and combine them in a weighted, possibly directional dependent way to form a new combined distribution function.

21.5.1 Mixing distribution functions

Mixers combine distribution functions as a weighted sum to form a new distribution function. The sum of the weights should not exceed one and in case it does, the weights are either normalized or clamped, depending on the kind of mixer chosen.

The weights are combined with the affected distribution functions in values of the following structure types.

```
struct bsdf_component {
   float         weight    = 0.0;
   bsdf          component = bsdf();
};
struct edf_component {
   uniform float weight    = 0.0;
   edf           component = edf();
};
struct vdf_component {
   float         weight    = 0.0;
   vdf           component = vdf();
};
```

Component in a mixing operation.

`weight` – Scaling factor for the effect of the component in the mixing operation. Range: $[0, 1]$.

`component` – Distribution function defining the operation of the component.

```
bsdf normalized_mix(
   bsdf_component[<N>] components
);
edf normalized_mix(
   edf_component[<N>] components
);
vdf normalized_mix(
   vdf_component[<N>] components
);
```

Mix `N` elemental or compound distribution functions based on the weights defined in the components. If the sum of the weights exceeds 1.0, they are normalized.

`components` – Array of distribution function components combined by the mix.

```
bsdf clamped_mix(
   bsdf_component[<N>] components
);
edf clamped_mix(
   edf_component[<N>] components
);
vdf clamped_mix(
   vdf_component[<N>] components
);
```

Mix `N` elemental or compound distribution functions based on the weights defined in the components. Distribution functions and weights are summed in the order they are given. Once a component weight would cause the sum to exceed 1.0, it is replaced with the result of subtracting the sum from 1.0. All subsequent weights are then set to 0.0.

`components` – Array of distribution function components combined by the mix.

21.5.2 Layering distribution functions

Layerers combine distribution functions by logically layering one distribution function over another. A weight controls the contribution of the top layer versus the base layer, which is weighted with one minus the weight. The weight can be a directional dependent weight, depending on the specific kind of layerer chosen.

```
bsdf weighted_layer(
  float  weight
  bsdf   layer
  bsdf   base   = bsdf(),
  float3 normal = state::normal()
);
```

Add an elemental or compound BSDF as a layer on top of another elemental or compound BSDF according to `weight`. The base is weighted with `1-weight`.

`weight` – Factor for `layer`. Range: [0, 1].
`layer` – Layer to add to the `base` BSDF.
`base` – Base BSDF.
`normal` – Surface normal vector, in internal space, applied to top layer.

```
bsdf fresnel_layer(
  color  ior,
  float  weight = 1.0,
  bsdf   layer  = bsdf(),
  bsdf   base   = bsdf(),
  float3 normal = state::normal()
);
```

Add an elemental or compound BSDF as a layer on top of another elemental or compound BSDF according to `weight` and a Fresnel term using a dedicated index of refraction for the layer. The base is weighted with `1-(weight*fresnel(ior))`.

`ior` – Index of refraction.
`weight` – Factor for `layer`. Range: [0, 1].
`layer` – Layer to add to the `base` BSDF.
`base` – Base BSDF.
`normal` – Surface normal vector, in internal space, applied to top layer.


```
bsdf custom_curve_layer(
  float  normal_reflectivity,
  float  grazing_reflectivity = 1.0,
  float  exponent             = 5.0,
  float  weight               = 1.0,
  bsdf   layer                = bsdf(),
  bsdf   base                 = bsdf(),
  float3 normal               = state::normal()
);
```

BSDF as a layer on top of another elemental or compound BSDF according to `weight` and a Schlick-style directional-dependent curve function. The base is weighted with `1-(weight*curve())`.

`normal_reflectivity` – Reflectivity for angle of incidence normal to the surface.
`grazing_reflectivity` – Reflectivity for angle of incidence at 90 degrees to surface normal.
`exponent` – Exponent for Schlick's approximation.
`weight` – Factor for `layer`. Range: [0, 1].
`layer` – Layer to add to the `base` BSDF.
`base` – Base BSDF.
`normal` – Surface normal vector, in internal space, applied to top layer.

```
bsdf measured_curve_layer(
  color[<N>] curve_values,
  float      weight        = 1.0,
  bsdf       layer         = bsdf(),
  bsdf       base          = bsdf(),
  float3     normal        = state::normal()
);
```

BSDF as a layer on top of another elemental or compound BSDF according to `weight` and a measured reflectance curve. The base is weighted with `1-(weight*curve())`.

`curve_values` – Measured data for the reflection behavior. A 1-d function measured in the pre-image range from zero to Pi/2 with equally spaced measured reflectance values.
`weight` – Factor for `layer`. Range: [0, 1].
`layer` – Layer to add to the `base` BSDF.
`base` – Base BSDF.
`normal` – Surface normal vector, in internal space, applied to top layer.

---

22 Appendix A – The syntax of MDL

This section describes the syntactic structure of MDL in a grammar using Wirth's extensions of Backus Normal Form. The left-hand side of a production is separated from the right hand side by a colon. Alternatives are separated by a vertical bar. Optional items are enclosed in square brackets. Curly braces indicate that the enclosed item may be repeated zero or more times.

Non-terminal and meta-symbols are given in *italic* font. Terminal symbols except identifiers, typenames, and literals are given in `teletype` font. The definition for the terminal symbols *identifier*, *typename*, *boolean_literal*, *integer_literal*, *float_literal*, and *string_literal* is given in Section 5 with the other parts of the lexical structure of MDL.

This grammar is an incomplete description of MDL, defining a superset of all legal MDL programs. Further restrictions required by legal MDL programs are included in corresponding chapters of this document. The page number of the section that describes the use of the syntax rule in MDL programs is listed to the right.

mdl	:	*mdl_version* { *import* } {[`export`] *global_declaration* }	9, 72
boolean_literal	:	`true` | `false`	15
enum_literal	:	`intensity_radiant_exitance` | `intensity_power`	15
integer_literal	:	*INTEGER_LITERAL*	15
floating_literal	:	*FLOATING_LITERAL*	15
string_literal	:	*STRING_LITERAL*	15
mdl_version	:	`mdl` *floating_literal* `;`	9
simple_name	:	*IDENT*	71
qualified_import	:	[`::`] *simple_name* { `::` *simple_name* } [`::` `*`]	71
qualified_name	:	[`::`] *simple_name* { `::` *simple_name* }	71
frequency_qualifier	:	`varying` | `uniform`	17

relative_type : `bool` | `bool2` | `bool3` | `bool4` 17
| `int` | `int2` | `int3` | `int4`
| `float` | `float2` | `float3` | `float4`
| `float2x2` | `float2x3` | `float2x4`
| `float3x2` | `float3x3` | `float3x4`
| `float4x2` | `float4x3` | `float4x4`
| `double` | `double2` | `double3` | `double4`
| `double2x2` | `double2x3` | `double2x4`
| `double3x2` | `double3x3` | `double3x4`
| `double4x2` | `double4x3` | `double4x4`
| `color` | `string` | `bsdf` | `edf` | `vdf`
| `light_profile` | `bsdf_measurement`
| `material` | `material_emission` | `material_geometry`
| `material_surface` | `material_volume` | `intensity_mode`
| `texture_2d` | `texture_3d` | `texture_cube` | `texture_ptex`
| *IDENT* [`::` *relative_type*]

simple_type : [`::`] *relative_type* 17

array_type : *simple_type* [`[` [*conditional_expression* | `<` *simple_name* `>`] `]`] 17, 39

type : [*frequency_qualifier*] *array_type* 17

parameter : *type simple_name* [`=` *assignment_expression*] 52, 53, 64
[*annotation_block*]

parameter_list : `(` [*parameter* { `,` *parameter*}] `)` 52, 64

positional_argument : *assignment_expression* 52

named_argument : *simple_name* `:` *assignment_expression* 52

argument_list : `(` 52
[*named_argument* { `,` *named_argument*}
| *positional_argument* { `,` *positional_argument*}
{ `,` *named_argument*}
]
`)`

import : `import` *qualified_import* { `,` *qualified_import*} `;` 9, 71, 73
| [`export`] `using` *qualified_name*
`import` (`*` | *simple_name* { `,` *simple_name*}) `;`

global_declaration : *annotation_declaration* 9
| *constant_declaration*
| *type_declaration*
| *function_declaration*

annotation_declaration : `annotation` *simple_name parameter_list* `;` 69

constant_declarator : *simple_name* (*argument_list* | `=` *conditional_expression*) 22
[*annotation_block*]

constant_declaration : `const` *array_type constant_declarator* { `,` *constant_declarator*} `;` 22

type_declaration :
- *alias_type_declaration* 48
- | *struct_type_declaration*
- | *enum_type_declaration*

alias_type_declaration : `typedef` *type simple_name* `;` 47

struct_field_declarator : *type simple_name* [`=` *expression*] [*annotation_block*] `;` 42

struct_type_declaration : `struct` *simple_name* [*annotation_block*] `{` {*struct_field_declarator*} `}` `;` 42

enum_value_declarator : *simple_name* [`=` *assignment_expression*] [*annotation_block*] 45

enum_type_declaration : `enum` *simple_name* [*annotation_block*] `{` *enum_value_declarator* { `,` *enum_value_declarator*} `}` `;` 45

variable_declarator : *simple_name* [*argument_list* | `=` *assignment_expression*] [*annotation_block*] 18

variable_declaration : *type variable_declarator* { `,` *variable_declarator*} `;` 18

function_declaration : *type* [*annotation_block*] *simple_name* 52, 64, 65
- (*parameter_list* [*frequency_qualifier*] [*annotation_block*]
  - (`;`
    - | *compound_statement*
    - | `=` *expression* `;`
    -)
  - | `(` `*` `)` [*annotation_block*] `=` *expression* `;`
  -)

annotation_block : `[[` *annotation* { `,` *annotation*} (`]` `]` | `]` `]`) 69

annotation : *qualified_name argument_list* 69

statement :
- *compound_statement* 48
- | *type_declaration*
- | *constant_declaration*
- | *variable_declaration*
- | *expression_statement*
- | *if_statement*
- | *switch_statement*
- | *while_statement*
- | *do_statement*
- | *for_statement*
- | *break_statement*
- | *continue_statement*
- | *return_statement*

compound_statement : `{` {*statement*} `}` 48

expression_statement : [*expression*] `;` 21

if_statement : `if (` *expression* `)` *statement* [`else` *statement*] 49

switch_statement : `switch (` *expression* `) {` {*switch_case*} `}` 49

switch_case : `case` *expression* `:` {*statement*} | `default :` {*statement*} 49

while_statement : `while (` *expression* `)` *statement* 48

do_statement : `do` *statement* `while (` *expression* `) ;` 48

for_statement : `for (` (*variable_declaration* | *expression_statement*)
[*expression*] `;` [*expression*] `)`
statement 48

break_statement : `break ;` 50

continue_statement : `continue ;` 50

return_statement : `return` *expression* `;` 51

literal_expression : *boolean_literal* 15
| *enum_literal*
| *integer_literal*
| *floating_literal*
| *string_literal* {*string_literal*}

primary_expression : *literal_expression* 21
| *simple_type* [`[ ]`]
| `(` *expression* `)`

postfix_expression : 20
primary_expression
{`++`
| `--`
| `.` *simple_name*
| *argument_list*
| `[` *expression* `]`
}

let_expression : `let` 67
(*variable_declaration*
| `{` *variable_declaration* {*variable_declaration*} `}`
)
`in` *unary_expression*

unary_expression : *postfix_expression* 20
| (`~` | `!` | `+` | `-` | `++` | `--`) *unary_expression*
| *let_expression*

multiplicative_expression	:	*unary_expression* {(* | / | %) *unary_expression*}	20
additive_expression	:	*multiplicative_expression* {(+ | -) *multiplicative_expression*}	20
shift_expression	:	*additive_expression* {(<< | >> | >>>) *additive_expression*}	20
relational_expression	:	*shift_expression* {(< | <= | >= | >) *shift_expression*}	20
equality_expression	:	*relational_expression* {(== | !=) *relational_expression*}	20
and_expression	:	*equality_expression* {& *equality_expression*}	20
exclusive_or_expression	:	*and_expression* {^ *and_expression*}	20
inclusive_or_expression	:	*exclusive_or_expression* {| *exclusive_or_expression*}	20
logical_and_expression	:	*inclusive_or_expression* {&& *inclusive_or_expression*}	20
logical_or_expression	:	*logical_and_expression* {|| *logical_and_expression*}	20
conditional_expression	:	*logical_or_expression* [? *expression* : *assignment_expression*]	20
assignment_operator	:	= | *= | /= | %= | += | -= | <<= | >>= | >>>= | &= | ^= | |=	20
assignment_expression	:	*logical_or_expression* [? *expression* : *assignment_expression* | *assignment_operator assignment_expression*]	20
expression	:	*assignment_expression* {, *assignment_expression*}	20

23 Appendix B – MBSDF file format

The MBSDF file format stores the data for a measurement of a bidirectional scattering distribution function (BSDF). A BSDF consist of two parts: a bidirectional reflection distribution function (BRDF) and a bidirectional transmission distribution function (BTDF). A measurement can contain either one or both.

This section documents version 1 of the MBSDF file format.

The filename extension is `.mbsdf`.

The file format starts with a file header and has one or two BSDF data blocks. The file header is readable ASCII text while the BSDF data blocks are binary data after a text identifier. The BRDF data comes before the BTDF data if both are present. The second block is directly appended to the first.

The basic entities to describe the format are the following types and their file storage size and description:

Type	Bytes	Storage
string	variable	string restricted to printable ASCII characters and terminated by '\n'.
uint	4	binary storage of a 32-bit unsigned integer value in little-endian ordering
float	4	single-precision floating-point number as defined in IEEE-754 with binary storage in little-endian ordering.

A type followed by an array-like brackets '[]' denote a sequence of zero or more occurrences of this type.

23.1 Header block

The header block starts with a magic identifier string including a version number followed by a sequence of strings for meta-data.

Type	Bytes	Value	Comment
string	20	`NVIDIA ARC MBSDF V1\n`	magic identifier including file format version number, terminated by newline.
string[]	variable	meta-data	sequence of zero or more strings, each representing a key-value pair in the format `<key>="<value>"\n`. The sequence `\n` encodes a newline character, `\"` encodes a double quote, and `\\` encodes a backslash in the value part. Other escape sequences are not allowed and ignored.

The meta-data can be used to document, for example, a name of the measured material, authorship, copyright, measurement device or date.

23.2 BSDF data block

A BRDF data block starts with an identifier to distinguish between BRDF and BTDF data, followed by the binary data. The identifier terminates with an equal sign before the newline character without a double-quote, which distinguishes it unambiguously from the sequence of meta-data in the header block.

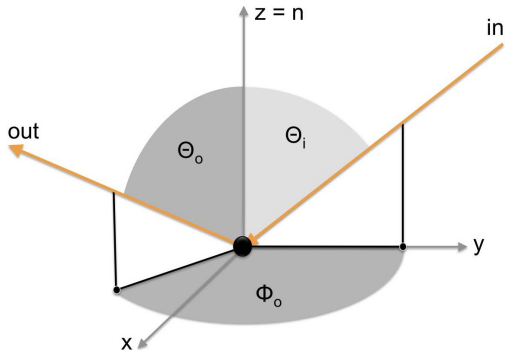

Figure 1: The geometry of the angles in a single BRDF measurement for the incoming light direction `in` and outgoing light direction `out`.

Type	Bytes	Value	Comment
string	variable	*IDENT*=\n	identifier to start the binary data block, where the *IDENT* can be `MBSDF_DATA_REFLECTION` or `MBSDF_DATA` for a BRDF block or `MBSDF_DATA_TRANSMISSION` for a BTDF block.
uint	4	T	enum value encoding the type of one measurement: $T = 0$: one float representing a scalar intensity measurement. $T = 1$: three float values representing a sRGB measurement.
uint	4	n_Θ	number of steps used for an equi-spaced angular resolution of Θ_i and Θ_o, greater than zero.
uint	4	n_Φ	number of steps used for an equi-spaced angular resolution of Φ, greater than zero.
float[]	$n_\Phi \cdot n_\Theta \cdot n_\Theta \cdot S$	measured data	S is the size of a single measurement in bytes, i.e., 4 for $T = 0$ and 12 for $T = 1$. The data is stored in the order defined by this indexing offset function: `offset(idx_theta_i, idx_theta_o, idx_phi) = idx_theta_i*`n_Φ`*`n_Θ `+ idx_theta_o*`n_Φ `+ idx_phi`.

The angular resolution is the same for Θ_i and Θ_o. Both angles are sampled in the range $[0, \pi/2]$, while Φ is sampled in the range $[0, \pi]$.

24 Bibliography

[1] NVIDIA Corporation, Santa Clara, California, United States. *NVIDIA Material Definition Language: Technical Introduction*, Version 1.0, 2014.

25 Changes to this document

Main changes for MDL 1.2 since the MDL 1.1 specification document version 1.1.3 from June 11, 2014.

25.1 Changes for version 1.2.2

- Clarified that the `uniform` or `varying` type modifiers are not allowed on the type of a global constant.
- Clarified that the `float3` constructor from `color` assumes the linear sRGB color space.
- Clarified that the `color` constructor from RGB values assumes sRGB values in linear space.
- Added terminology that open material definitions are also called *material variants*.
- Clarified that the resource types `texture_2d`, `texture_3d`, `texture_cube`, `texture_ptex`, `light_profile`, and `bsdf_measurement` can be used as types for variables in let-expressions.
- Clarified that the `luminance` function assumes linear sRGB color space for its `float3` parameter.

25.2 Changes for version 1.2.1

- Updated version to 1.2.
- Distances for the volume absorption coefficient and the volume scattering coefficient of the MDL material model in Section 13.3 are in meters in world space.
- Added new `anno::soft_range` and `anno::hard_range` annotation overloads for vector and color parameter types.
- Added a new `state::object_id()` state function returning an integer object ID or zero if none is provided in the scene.
- The `::state::meters_per_scene_unit()` and `::state::scene_unit_per_meters()` state functions in Section 19.2 offer the conversion between the world space in scene units and meters.
- The radius of the rounded corner state function in Section 19.3 is in meters in world space.
- The standard library functions that return width, height, and depth of textures have been extended to handle also invalid texture references and return zero in this case.
- Added `texture_isvalid`, `light_profile_isvalid`, and `bsdf_measurement_isvalid` functions to test resources for valid references.
- Added a new `wrap_clip` mode for texture lookups that returns zero for out-of-range lookups.
- Added a `tangent_u` parameter to the `df::measured_edf` emission distribution function to control the orientation in the case of anisotropic light profiles.