Away3D Data format (AWD) v2.1 Alpha

Last updated: May 7th 2013

This is a work-in-progress draft of the file format specification for the binary (second-generation) Away3D data format (AWD). Any community feedback is welcome at http://groups.google.com/group/away3d-dev or to r@richardolsson.se.

Please note that this is an incomplete document, in an early pre-release state. Also note that the tools that are available at the Google Code project page for AWD are under development and should not be expected to conform fully to this specification at any point before it's final release.FS

All feedback is welcome, but these points are particularly interesting at this time:

- Namespaces: General thoughts on the need for namespaces?
- Material architecture: Any community feedback or wishes for features are welcome.
- Animation architecture: Any community feedback or wishes for features are welcome.
- Command blocks: Any community feedback is welcome.
- Limitations (see Part III): Any part of the format where limits need to be lifted?
- Paths, number of steps per segment? Should it be defined per instance or per path data?

Captions in red mark things that are very likely to change.

Table of Contents

1.

Part I: Introduction

This document defines the Away3D data format (AWD) created by the Away3D team for use as an exchange format between any conforming exporter on one end and mainly the Away3D engine (but also any other conforming parser) on the other.

An AWD file generated according to the rules and structure defined in this document is guaranteed to be successfully parsed by the Away3D runtime AWD importer and any other conforming parser.

1. What is AWD?

AWD is a binary file format for 3D scenes, objects and related data. It's main use is with the Away3D engine. The file format specification (this document) and a set of tools for working with AWD files are maintained by the Away3D development team.

1. What is Away3D?

Away3D is a real-time 3D engine designed for ActionScript 3 and the Adobe Flash Platform. It is free and open-source and can be downloaded from www.away3d.com. See the website for more information.

2. AWD Design intentions

AWD was created as a transfer format with the goal of having a one-to-one relationship between file format features and the engine features of Away3D. This is to be compared with some file formats that do not support all the features that Away3D supplies, and other formats which are much too verbose for the purpose of using them with Away3D.

In addition to this goal, AWD has been designed to be web and Flash friendly, meaning it needs to meet tough file size requirements, while not being too expensive to parse. Focus is on parsing performance over generation performance.

The main goals of AWD and reasons for creating it are:

- Support all the features of Away3D so that an entire Away3D scene can be transferred using AWD, but don't jump through hoops trying to be more than that.
- Provide a web-friendly format keeping in mind that AWD files will need to be transferred over the Internet as part of a user experience, meaning they should be as small as possible while at the same time minimizing the impact on parsing performance and time.
- Be extendable. Away3D is a constantly evolving engine, and the supporting file format needs to be able to keep up the pace. Furthermore, it should be possible for AWD users to extend the format without breaking conformity with the format specification.
- Be backwards (and forwards) compatible by defining a solid base format, and allowing for extension by well defined rules. A parser that does not understand a particular file format feature

should be able to ignore it and an evolution of the file format should not have to change the base format in such a way that older parsers can no longer follow the rules they know.

3. What about old-school AWD?

The previous incarnation of AWD, which was a plaintext format, is being deprecated with the introduction of this binary revision. The requirements of file formats used in Flash 3D going forward promote the use of compressed binary formats over ASCII-based files, which was a big factor in the decision to create this binary format.

What's described in this specification should be regarded as the current-generation AWD format, and used whenever possible.

2. How to read this document

This document has been divided into three sections, with the first section being an introduction to the file format, the Away3D engine and this document. The second section contains a detailed specification of all elements in the AWD file format. The third section describes how to use the AWD format, e.g. to create an encoder or a decoder or extend the file format for a particular use-case.

1. For encoder implementers

Encoder implementers should read the entire document paying extra attention to the structure tables for all blocks and primitive data types in Part II. The section called "Official AWD tools" might also contain information that is relevant to encoder implementers, including information about the official AWD SDK which can ideally be used to create exporters.

Whenever the expressions SHOULD or SHOULD NOT are used in a context that describes the structure of an AWD file or the behavior of an encoder, the encoder is encouraged to comply but not required to do so.

Whenever the expressions MUST or MUST NOT are used in a context that describes the structure of an AWD file or the behavior of an encoder, the encoder is required to comply or it can not be regarded as conforming to the AWD specification.

2. For parser implementers

Parser implementers should read the entire document paying extra attention to the structure tables for all blocks and primitive data types in Part II, as well as the section "Parsing an AWD document" in Part III. The section called "Official AWD tools" might also contain information that is relevant to parser implementers, including information about the official AWD SDK which can ideally be used to aid parsing of AWD files.

Whenever the expressions SHOULD or SHOULD NOT are used in a context that describes the structure of an AWD file or the behavior of a parser, the parser is encouraged to comply but not required to do so.

Whenever the expressions MUST or MUST NOT are used in a context that describes the structure of an AWD file or the behavior of a parser, the parser is required to comply or it cannot be regarded as conforming to the AWD specification.

3. For those looking to extend the format

A general understanding of AWD is necessary and can be obtained by reading Part I of this document and skim through Part II, paying extra attention to the sections called "The Top-level structure of an AWD document" and "The data block concept".

After that, the main focus should be reading the Part III section called "Extending AWD" and sections referenced therein.

4. Terminology

1. AWD Document

Used as an ambiguous term to refer to either an AWD stream (e.g. over a network) or a logical AWD file (e.g. on a hard drive.) Essentially any data beginning with an AWD header is considered an AWD document.

2. Block

Blocks are the top-level containers of data. They occur in a sequential list in the file data body. See the section "The Data Block concept" for more information.

3. Field

Fields are the smallest data elements defined by the AWD specification. They are a single logical chunk of data, such as an integer, float or a non-POD element such as a matrix or string as defined by this document.

4. Element

The term “element” is used to refer to any logical part of the file, be it a block or a field or a structured (and often recurring) sequence of fields.

5. Vector

The term "vector" when used in an AWD context refers to a vector according to the mathematical definition, e.g. a position in N-dimensional space (where N is the length of the vector.) Arrays (which are sometimes called "vectors" in other contexts) are called lists in the AWD context.

2.

Part II: File format specification

1. Top-level structure of an AWD document

The AWD document always begins with an uncompressed file header with meta-data, which is followed by an optionally compressed file body, containing the actual data.

Offset	Size	Type	Description
0	11	*File header*	See section File header
11	Variable	List of *blocks*	See section File body

Table 1: AWD top-level structure

1. File header

The header defines which version of the AWD format specification a file conforms to, as well as the algorithm used to compress the data body, if any, and configuration flags. Using a “Magic string”, it identifies itself as an AWD file.

Offset	Size	Type	Description
0	3	*ConstString*	Magic string, “AWD”
3	1	uint8	Version number (major version.)
4	1	uint8	Revision number (minor version.)
5	2	uint16	Flags (two-byte bitflag, see separate table.)
7	1	uint8	Compression type. 0: Uncompressed 1: File-level ZLIB 2: File-level LZMA
8	4	uint32	Compressed body length

			in bytes. Used for integrity check. Ignored if streaming.

Table 2: AWD header structure

The flags field is a two-byte bitflag field where each bit is a boolean (0=false, 1=true) that defines a configuration parameter for the entire file. The meaning of each bit is defined and described in the table below.

Bit	Value	Name	Description
00	0x0001	Streaming	Defines whether this file should be treated as a streaming file. If this bit is set, it means that the f is streaming and that more blocks can be expecte even after the file appears to end.
01 - 15	0x0008 - 0x8000	Unused	Unused in this version of the format.

Table 3: AWD header flag bits

2. File body

The file body data is a sequence of any number of data blocks, where each data block has the same top-level format, with fields defining block type and size (See the section called "The data block concept" for details).

The file body in it's entirety can optionally be compressed using one of the supported compression algorithms. Body blocks can also be added to an existing AWD document over time, via a mutable media like a network socket. This is referred to as streaming.

3. Streaming AWD

The AWD file format has been designed with the possibility of streaming and progressive loading in mind. Block references are required to always point backwards (to a previously declared block) which means that a parser can be certain when encountering a reference that it is not being made to a currently unknown block even though the entire file has not yet been downloaded.

To alert a parser that a document is streaming (and hence that the parser should continue reading data when available until the stream is closed by the environment) the “streaming” flag bit must be set in the flags field in the header.

Streaming AWD files *do not support compression* since parsing needs to be possible even before the entire document has finished loading (if it ever does.) Future versions of the format may support per-block compression.

4. Compression

The body part of an AWD file can optionally be compressed using one of the two supported compression algorithms, ZLIB (deflate) and LZMA. Which algorithm is used is defined by the compression type field in the document header.

If a parser does not recognize the algorithm defined in the compression type header field, the file can not be read by that parser, which should exit with an error status.

1. ZLIB/Deflate compression

The ZLIB/Deflate compression algorithm is used by the very popular ZIP and GZIP compression file formats and provides a fairly efficient compression at low decompression performance costs. Deflate compression is natively supported by many environments, including Flash Player. This means that in many cases (one of which is indeed the Flash Player) ZLIB compression is an excellent trade-off between file size and decompression speed. In Flash Player particularly, native decompression of ZLIB/Deflate can be several orders of magnitude faster than decoding other formats using implementations in ActionScript 3.

In a ZLIB compressed AWD file, the body is the exact output from ZLIB including heading and trailing meta-data (checksum), which means it can be handed straight to a ZLIB decoding (inflation) machine.

2. LZMA compression

LZMA is an extremely efficient compression algorithm and is part of the popular Windows 7zip compression utility. However, because native implementations are rare, and none exists for Flash Player, decompression often means more work and is often slower than when using ZLIB.

The LZMA compression is very configurable, and hence requires some meta-data to be stored for the compressed body data to be correctly decompressed. The first nine bytes of the body data in an LZMA-compressed AWD file define the size of the decompressed body as an unsigned 32-bit integer, followed by the LZMA properties encoded as per the LZMA standard. The below table describes the body structure of an LZMA-compressed AWD file.

Offset	Size	Type	Description
0	4	*uint32*	Length of decompressed body.
4	5	*ByteArray*	LZMA properties encoded as defined in the LZMA SDK.
9	Variable	*ByteArray*	Compressed body data.

Table 4: Structure of an LZMA-compressed AWD body

This structure allows for easy decompression using the LZMA SDK LzmaDecode() function, without the need of dynamic buffer allocation and chunk-for-chunk decompression of the stream.

2. Field types and special values

In this specification are recurring references to a number of both POD and complex (aggregate) data types. This section details the format of these data types and how they are parsed.

1. Field type identification

In contexts where types can vary (e.g. user attributes, see below) the data type is identified by an 8bit integer ID. This 8bit field, when referred to in structure tables, is simply called *type*.

ID	Type	Category
1	int8	Numeric
2	int16	
3	int32	
4	uint8	
5	uint16	
6	uint32	
11	float32	
12	float64	
21	bool	Derived numeric
22	color	
23	*BlockAddr*	
31	*ConstString*	Array types
32	*ByteArray*	
41	*Vector2x1*	Math types
42	*Vector3x1*	
43	*Vector4x1*	
51	*Matrix3x2*	
52	*Matrix3x3*	
53	*Matrix4x3*	
54	*Matrix4x4*	

Table 5: Field data type identifiers.

2. Numbers

1. Endianness

All numeric values in AWD are *little-endian*. Numeric values should never be encoded as big-endian in AWD. This means that to read a multi-byte numeric field the parser has to wait until the entire field has been loaded (since the MSB is the last one.) However, it also has performance gains on most modern platforms which are natively little-endian, and thus able to read entire streams of little-endian numeric data in a single operation. On these systems the same streams can then be sent of to the GPU without any marshaling, which constitutes a big optimizations particularly in high-level languages like ActionScript (Flash) and JavaScript (WebGL).

2. Integers

All fields that contain integers are defined as either int or uint (for unsigned integers) regardless of the size of their C representation. They are never referred to as "long", "short", "word" or any of the typical platform names. Instead, to remain platform-agnostic, a numeric suffix defines the width in bits. The following integer types can be used in AWD:

- int8 and uint8
- int16 and uint16
- int32 and uint32

3. Floating-point numbers

Non-integer numeric fields are referred to as floats, floating point numbers. Like with integers, a numeric suffix explicitly defines the precision. What in C are usually referred to as doubles are simply referred to as floats with a greater bit-width:

- float32
- float64

Float values must always be encoded as IEEE-754 compliant floating point numbers.

3. Booleans and true/false values

Booleans are encoded as a single byte where any non-zero value indicates a true state. False must hence be encoded as 0 (all eight bits equal zero) and any other value will be interpreted as true.

Offset	Size	Type	Description
0	1	uint8	Boolean encoded as an 8 bit integer. Any non-zero value indicates true.

4. Byte arrays

Whenever a byte array is mentioned, this is a reference to a sequence of arbitrarily formatted bytes. The context defines what the exact format of the content is and it's length, but usually the exact structure is not relevant to AWD as a format (e.g. with embedded images) and should be treated by a separate module (e.g. a JPEG decoder.)

5. Strings

Two types of character strings are used in AWD, ConstString and VarString. Both comprise an array of UTF-8 characters without BOM, the only difference being that whereas the length of a ConstString is always defined by the context, a VarString can have variable length in any given context.

1. ConstString

Offset	Size	Type	Description
0	Context-sensitive	Byte array	String content as UTF-8 without BOM.

2. VarString

A VarString can have any length between 0 and 65536 bytes. The length is defined by the first two bytes in the VarString field, which are to be interpreted as a 16 bit unsigned integer.

Offset	Size	Type	Description
0	2	uint16	String length
2	Variable	Byte array	String content as UTF-8 without BOM.

6. Lists

1. ConstList

TBD

2. VarList

TBD

7. Vectors and matrices

Vectors and matrices are serialized as a one-dimensional list of floating point numbers. The context defines the size of the vector or matrix. The precision (32 or 64 bits per float) for vectors and matrices is defined per-block in the block header flag field.

1. N-dimensional vector

N-dimensional vectors are encoded as a 1xN matrix.

2. MxN matrix

Matrices are encoded as a column-major (col0, col1, ... colN) serial list of 64-bit floating point numbers. For a matrix with N columns and M rows, the total size of resulting byte array is MxNxP bytes, where P is either 4 or 8 depending on the precision used.

8. Addresses

Fields referred to as *BlockAddr* fields are numeric block addresses, usually to a previous block in the file (and sometimes to the block itself, but never to a later occurring block). These are always 32-bit unsigned integers, where a null value is allowed (and means no block is referenced.)

9. Colors

Whenever a color is stored in an AWD file, it is represented by four 8-bit values, defining the red, green, blue and alpha channels respectively. This means that a color is always 32 bits long in total, and that every channel can have 256 possible values.

10. Attributes

The AWD format is designed to be extendable, both by future versions of AWD and by user applications. Blocks in an AWD file can have attributes that can either be user-defined (e.g. for use in a game or physics engine) or defined by the AWD format specification.

There are two types of attributes, differentiated and referenced by their key/name types:

- Numeric attributes (sometimes called "properties") are used mainly by the file format itself to maintain forward compatibility. The key is a 16-bit unsigned integer IDs, which makes it very compact while allowing for 65535 values. It's however not human-readable so the meaning of a key ID needs to be established in a contract between encoder and parser, e.g. this document.
- Text attributes (sometimes called "user attributes") are suitable for generator transparency (e.g. letting the end-user define them straight into the file.) Keys are VarStrings which means that these attributes are human-readable and more easily human-writable.

1. Attribute lists

Attributes are organized in a flat list, so that parsers that ignore attributes can skip the entire list in one seek operation. The list is a very simple structure consisting of a 32 bit integer defining it's length, followed by the serialized list of attributes.

Offset	Size	Type	Description
0	4	uint	Attribute list length in bytes.
4	Variable	*Attribute stream*	List of attributes

2. Numeric attributes (“properties”)

Numeric attributes are key/value pairs where the semantics of the key needs to be derived from a mutual understanding between encoder and parser. Attributes like these are used throughout the AWD format as a way of defining peripheral values for an element, like the number of segments or dimensions of a cube or material properties.

Offset	Size	Type	Description
0	2	*uint16*	Attribute ID (key)
2	4	*uint32*	Value length
6	Variable	Variable	Attribute value

The value length field defines the length of the value data portion of the attribute. The type of the value is defined by the context and attribute ID. For instance, a Primitive's width attribute is always a float64.

3. Text attributes (“user attributes”)

Attributes themselves are key/value pairs with a type field defining the data type of the value. The key (name) of the attribute is a VarString, as defined above in the section called Strings.

Offset	Size	Type	Description
0	1	*uint8*	Namespace ID. See "Extending AWD" for more information.
1	Variable	*VarString*	Attribute name (key) as string.
Variable	1	*type*	Attribute type (data type of value). See section "Field type identification".
Variable	4	*uint32*	Value length
Variable	Variable	Variable	Attribute value

At first glance the value length field can seem redundant since the attribute type field implicitly defines the size of the value. However, the length field allows attributes to store arrays of values (though always expressed in bytes, not number of elements.) An attribute value of type int32 with length 12 bytes contains three integer elements. Furthermore, for a parser that does not recognize a particular attribute type, the length field allows the entire value to be skipped.

Because of the length field, string values do not need to be defined as VarStrings with their own length field, but can instead be encoded as ConstStrings, the length of which is defined by the attribute value length field.

4. Method elements

Currently there are 4 types of Methods in AWD. This Methods are: ShadingMethods, ShadpwMethods, ShadowMapperMethods, and Lens (Lenses are no real method, but in AWD they should be handled as such).

All Methods are stored in a similar structure.

Offset	Size	Type	Description
0	2	uint16	Method type, see separate table.
2	Variable	*NumAttrList*	Method properties (see separate table.)
Variable	Variable	*UserAttrList*	User properties for method

ID	Name	Type
1	Color	*color*
2	OutlineColor	*color*
3	alpha	*float32*
100	texture	*BlockAddr*
101	cubeTexture	*BlockAddr*
102	textureProjector	*BlockAddr*
103	Second Normal map	*blockaddr*
104	gradient-texture	*Blockaddr*
105	lightMap-texture	*blockaddr*
201	DedicatedMesh	*bool*
202	ShowInnerLines	*bool*
203	UseSecondaryUV	*bool*
204	BasedOnSurface	*bool*
1001	matrix	*ConstList of float64*
1101	alphaMultiplier	*float32*
1102	redMultiplier	*float32*
1103	greenMultiplier	*float32*
1104	blueMultiplier	*float32*
1105	colorOffset	*color*
1106	power	*float64*
1107	Strength	*float64*
1108	World width	*float64*
1109	World height	*float64*
1110	World depth	*float64*
1111	DispersionR	*float64*
1112	power	*float64*
1113	**Reflectance**	*float64*
1114	Epsilon	*float64*
1115	cut-off	*float64*
1117	smoothness	*float64*
1118	warp factor	*float64*

1119	Range	*float64*
1120	CoverageRatio	*float64*
1121	OutlineSize	*float64*
1122	Distance	*float64*
1123	levels	*uint8*
1124	blendMode	*uint8*
1125	DepthMapSize	*uint8*
1127	Samples	*uint32*
1128	Number of Cascades	*uint32*
1129	RefractionIndex	*int32*
1130	baseMethod	*MethodBlock (BlockAddr for SharedMeth or null for not SharedMeth)*
1140	`Scattering`	*float32*
1141	`Translucency`	*float32*
1142	`Scatter Color`	*color*
1143	DispersionG	*float32*
1144	DispersionB	*float32*
1145	MaxDistance	*float32*
1146	mask	*BlockAddr*
1501	FOV	*float32*
1502	ProjectionHeight	*float32*
1503	x	*float32*
1504	y	*float32*

Table: List of all Properties used by ShaderMethods, ShadowMethods, ShadowMapper, and Lens.

11. References and null values

In cases where null values can exist (such as references to other blocks) Null is represented by zero, which also means it can be interpreted as false if evaluated as a boolean. This also means that in cases where Null needs to be treated as a special case (i.e. block references, namespace handles) zero is not a valid value but will be interpreted as Null.

3. The Data Block concept

The uncompressed file body is a flat sequence of "data blocks" that adhere to a pre-defined structure. The first fields in a block are required to be the same for any type of block, and are referred to as the block's header. The type and length of a block is defined by these fields, allowing full forward-compatibility between an extended AWD file and an unaware parser, which can determine whether a block type is known and if not skip that block.

Blocks can reference other blocks using their numeric 32-bit IDs. References are required to always be made backwards, meaning that a block can not reference a target block that is defined later in the document than the referring block. This is to speed up parsing and prevent problems with streaming AWD documents and should not cause any troubles in most realistic use-cases.

1. Anatomy of a block

All data blocks share a couple of characteristics. First, they all begin with a block header which specifies the block ID, type, and length. This must be read by all parsers to decide whether a block can be parsed or should be skipped (by seeking forward the number of bytes specified in the length field.)

Offset	Size	Type	Description
0	4	*BlockAddr*	Block ID
4	1	*uint8*	Block namespace handle.
5	1	*uint8*	Block data type ID. See table in the “Blocks” section of this document
6	1	*uint8*	Flags (see separate table.)
7	4	*uint32*	Block data size in bytes.

Table 6: Block header

Jidder

Bit	Value	Name	Description
00	0x0001	High precision	Defines whether to use float32 (bit unset) or float (bit set) for matrix and vector fields within this block.
01 - 15	0x0002 - 0x8000	Unused	Unused in this version of the format.

Table 7: Block flags

The body of a block varies and can theoretically be anything. Generally, the top-level structure of a block usually resembles the following, including the already mentioned header:

- Block header: 11 bytes structured according to the table above, defining block ID, namespace, type and length.

- Basic required values: a static condensed list of value-only fields where the order and semantics are defined explicitly by the AWD file format specification and where every value is required.
- Optional properties: a dynamic list of key/value pairs defined by the AWD file format specification. See the section “Numeric Attributes” for more info.
- Sub-structure: Lists of logical elements that are hierarchically ordered below this block, but do not have their own blocks, like lists of joints in a skeleton or sub-meshes in a geometry block.
- User attributes: A dynamic list of arbitrary key/value pairs. These can be used for app-specific data or meta-data stored by an encoder user directly.

2. Block IDs and addressing

All blocks have a unique numeric ID, which is used to reference that block from other blocks. The zero block ID indicates that a block will never be referenced and that a parser hence is not required to keep it in memory after it is done with it. These blocks are said to be temporary. Blocks with ID greater than zero are said to be persistent.

Block IDs must be incremented a single step for each block that has an ID. The first persistent block must have ID 1, and the next persistent block have ID 2. Any temporary blocks (that do not have IDs) do not affect this sequence of IDs.

Block references *must always be made backwards*, meaning that the block with ID N can only reference blocks for which the ID is less than N.

3. Block namespaces

A block needs to exist within a namespace, which defines whether it's a standard AWD block (the Null namespace) or part of an AWD extension. Namespace handles in block headers and elsewhere are 8-bit numeric IDs which allows an AWD file to have 255 namespaces on top of the default Null namespace.

All blocks defined in this document belong in the Null namespace, as they are part of the AWD standard blocks. See the Part III section "Extending AWD" for information about how to use other namespaces when extending the file format.

4. Block types

The following list documents the native AWD block types and the type IDs. All native AWD blocks are required to be defined in the Null namespace.

NS	Type ID	Block type	Category
0	1	TriangleGeometry	Geometry/data
0	11	PrimitiveGeometry	Geometry/data
0	21	Scene	Scene objects
0	22	Container	Scene objects
0	23	MeshInstance	Scene objects
0	31	SkyBox	Scene objects

0	41	Light	Scene objects
0	42	Camera	Scene objects
0	43	TextureProjector	Scene objects
0	51	LightPicker	Light Objects
0	81	StandardMaterial	Materials
0	82	Texture	Materials
0	83	CubeTexture	Materials
0	91	SharedMethod	Method
0	101	Skeleton	Animation
0	102	SkeletonPose	Animation
0	103	SkeletonAnimation	Animation
0	111	MeshPose	Animation
0	112	VertexAnimation	Animation
0	121	UVAnimation	Animation
0	122	Animator	Animation
0	253	Command	Misc
0	254	Namespace	Misc
0	255	Meta-data	Misc

Table 8: Block types, type ID's (with namespaces) and categories

4. Geometry blocks

5. Geometry blocks

1. TriangleGeometry (ID 1)

TriangleGeometry blocks contain geometry data for common triangle meshes. They are split into sub-geometries which in turn contain a number of data streams that define the geometry. A triangle geometry block can be referenced by several mesh instances in a scene, so that the same geometry data is used to render several objects saving storage space on disk and in memory.

The top-level structure of a mesh data block is defined by this table:

Offset	Size	Type	Description
0	11	*BlockHeader*	As defined in Table 6.
11	Variable	*VarString*	Look-up name.

Variable	2	*uint16*	Number of sub-geometries.
Variable	Variable	*NumAttrList*	Geometry properties.
Variable	Variable	List of *SubMesh*	Sub-meshes
Variable	Variable	*UserAttrList*	User attributes.

Table 9: Top-level structure of a MeshData block

There are no numeric properties for TriangleGeometry blocks in this version of AWD.

1. Sub-geometry

A sub-geometry is a per-material division of a triangle geometry. Sub-geometries are also used by Away3D to split meshes into buffers that do not exceed platform buffer size limits. Sub-geometries define their geometry as data streams, condensed lists of numeric values distinguished by type, e.g. vertex positions, face indices and UV coordinates.

Offset	Size	Type	Description
0	4	*uint32*	Length of sub-mesh (total length of data streams in bytes)
4	Variable	*NumAttrList*	Sub-mesh properties
Variable	Variable	List of *DataStream*	Geometry data streams
Variable	Variable	*UserAttrList*	User attributes.

Table 10: Structure of a MeshData block sub-mesh

A sub-mesh can contain any number of data streams (as long as it does not exceed the size limit inherited by the 32-bit length field).

2. Data streams

A data stream is a condensed sequence of geometry data, such as vertex positions, UV coordinates or face indices. The below table defines the structure of a data stream.

Offset	Size	Type	Description
0	1	uint8	Stream type (see separate table.)
1	1	uint8	Content data type.
2	4	uint32	Length of data stream in bytes
5	Variable	List of float32/int16	Data stream contents

Table 11: MeshData data stream structure

ID	Stream type
1	Vertex positions
2	Face indices
3	UV coordinates
4	Vertex normals
5	Vertex tangents
6	Joint index
7	Joint weight

Table 12: Stream types, type IDs and content data types for sub-meshes.

The internal structure of the values in a data stream varies between stream types, and are described in the next sections.

1. Content structure of vertex positions, vertex normals, and vertex tangents data streams

All these streams consist of a flattened serialized list of number triplets, with the general form X_1 Y_1 Z_1 X_2 Y_2 Z_2 ... X_N Y_N Z_N. This means that the total number of value items in a stream of one of these types is always a multiple of three.

2. Content structure of face index data streams

Face index data streams define triangles as triplets of index integers. The indices refer to vertices defined in the vertex stream, where index zero refers to the vertex defined by the first triplet in the vertex stream. An index *i* in the face index data stream refers to the vertex defined by the three subsequent floating point numbers starting at index *3i* in the vertex stream.

Content structure of UV coordinate data streams

The UV coordinate stream is very similar to the vertex position stream except each item is a two-dimensional vector which defines a UV pair. This means that the total number of values in a UV coordinate data stream is always a multiple of two. The UV pair with index *i* comprises the two subsequent floating point numbers starting at *2i* in the UV stream, and defines the UV coordinate for the vertex defined by the three subsequent floating point numbers starting at index *3i* in the vertex stream.

In some applications, it makes sense to have more than one set of UV coordinates. This can be achieved simply by including several streams of the UV type in a sub-geometry.

3. Content structure of Joint index and weight streams

The joint index and joint weight streams are used to bind the vertices of a mesh to a joint in a skeleton. They must have the same number of elements.

The number of joints that affect a vertex needs to be constant throughout the mesh, which means that the number of values in these streams are even multiples of the number of vertices in the mesh. For the sake of example, lets define N as the number of joints per vertex. If N=2, that means that each vertex can be bound to two joints.

For each vertex in the vertex data stream, N joint weights are stored in the joint weight stream. At the corresponding index in the joint index stream is a reference to which joint this weight concerns, stored as an index into the list of joints in the skeleton.

Continuing the example of N=2, consider the following two streams:

Joint indices: 0, 1, 3, 0

Joint weights: 0.6, 0.4, 1.0, 0

The joint weights for the first vertex in the vertex stream are defined by the first two numbers in these streams (since N=2). The first joint to which this vertex is bound is the one with index 0, and the bind is weighted at 0.6. The same vertex is also bound to joint 1, for which the weight is 0.4.

The second pair in each stream defines the two bindings for the second vertex of this sub-mesh. In this example, this vertex is bound to joint 3 which is weighted at 1.0, and also to joint 0. Note however that this last binding has zero weight, which in practice means that it will be ignored.

The sum of all weights on a vertex must always be 1.0. In this example the first pair has a sum of 1.0 (0.6 and 0.4) and so does the second pair (1.0 and 0.0), and they are thus both correctly formatted.

Weights should also be ordered from largest to smallest, so that any zero weights are always at the end. This is because the rendering engine might discard any weights after the first zero weight as a performance optimization. Away3D does this.

Since AWD enforces a constant number of joints per vertex, there can often be cases where it is possible to bind more joints to a vertex than what is necessary for that vertex. In those cases the index can be set to any number, and the weight to zero as in the example above.

2. PrimitiveGeometry (ID 11)

Primitive blocks are a type of geometry meta-data block in that it doesn't actually contain any geometry data. Rather, the meta-data in the block defines the type of primitive and it's properties, and the actual geometry is re-constructed by the receiving end using these properties.

The primitive block defines a field for the primitive type followed by a numeric attribute list defining properties such as dimensions, geometric density et c. Like most other blocks, user attributes can be appended to this block through it's user attribute list.

Offset	Size	Type	Description
0	11	*BlockHeader*	As defined in Table 6.
11	1	*uint8*	Primitive type (see separate table.)
12	Variable	*NumAttrList*	Primitive properties.
Variable	Variable	*UserAttrList*	User attributes.

Table 13: Primitive block fields (in addition to common scene object fields).

The type field defines the primitive type according to the following table:

Type ID	Primitive type
1	Plane
2	Cube
3	Sphere
4	Cylinder
5	Cone
6	Capsule
7	Torus

Table 14: Primitive types.

The numeric properties are defined by the following table. Some properties are limited to a specific primitive or set of primitives. The "relevant for" column declares when this is the case, and any semantic differences for a property depending on primitive type is described in the description column.

ID	Name	Type	Relevant for	Description
999	WireStyle	uint8	all	0: no wires 1: wires only 2: wires and mesh
1	Width	float32	Plane, cube	Total width of the primitive (X axis).
2	Height	float32	Plane, cube, cylinder,Capsule	Total height of the primitive (Y axis).
3	Depth	float32	Cube	Total depth of the primitive (Z axis).
4	Radius (default)	float32	Sphere, cylinder,Capsule,Torus	Radius of a sphere or cylinder primitive. When used on a cylinder, defines the overal or "default" radius, which can be overridden by the more specific top/bottom radii. See note at 1.
5	Radius (top/tube)	float32	Cylinder,Torus	Top radius of a cylinder. Overrides default radius (property 4) if one exists.
6	Radius (bottom)	float32	Cylinder	Bottom radius of a cylinder. Overrides default radius (property 4) if one exists.
7	Segments (W/R)	uint16	Plane, cube, sphere, cylinder,Capsule,Torus	Number of segments on the width/X axis on plane and cube primitives, or number of sectors on sphere, cylinder and cone primitives.
8	Segments (H/T)	uint16	Plane, cube, sphere, cylinder,Capsule,Torus	Number of segments on the height/Y axis on all primitives.

9	Segments (D)	uint16	Cube	Number of segments on the depth/Z axis.
10	Top closed	bool	Cylinder	Defines whether the top end of a cylinder should be closed, forming a cone.
11	Bottom closed	bool	Cylinder	Defines whether the bottom end of a cylind should be closed, forming a cone.
12	Y up/tile6	bool	Plane, Sphere, Cylinder,Capsule,T orus, cube	Defines whether the primitive should be created such that the "up" axis of the mesh aligns with the Y axis. On planes and cylinders, the up axis is the "height" axis. O sphere, the up axis is defined as that which intersects the sphere's poles.

Table 15: Primitive block properties.

The top, bottom and default radius properties on a cylinder are related to each other such that if either top or bottom radius has been defined and the other is missing, the undefined radius must be assumed to be the same as the default radius if on has been defined. If no default radius has been defined, an undefined top or bottom radius must be assumed to have the same value as the other which has been defined. If neither of the three properties have been defined, the default value can be decided by the parser.

Jidder

6. Scene object blocks

Scene graph blocks are spatial objects that can be added to a scene graph. They share the same first three fields, defined below.

Offset	Size	Type	Description
0	4	*BlockAddr*	Parent ID (Numeric). Reference to a previously defined scene graph object.
4	128	*4x4 matrix*	Transform
132	Variable	*VarString*	Look-up name.

Table 16: Common fields for all scene blocks.

The subsequent sections of a scene graph block differ between the various block types.

1. Scene (ID 21)

The scene is a special case of the scene graph blocks. It's the top-level element in the scene graph, and can not have a parent. Hence, the parent field must always be null (zero) and should be ignored by parsers. In addition to the common fields that all scene object blocks share, scene blocks only have an empty numeric attribute list, and any number of user attributes.

Offset	Size	Type	Description
0	11	*BlockHeader*	As defined in Table 6.
11	Variable	*SceneHeader*	As defined in Table 16.
Variable	Variable	*NumAttrList*	Scene properties (unused in this version.)
Variable	Variable	*UserAttrList*	User attributes.

Table 17: Scene block fields (in addition to common scene object fields).

2. Container (ID 22)

Containers are objects to which other scene-graph objects can be parented, but that don't have any volume or visual appearance on their own. Containers like scenes only have an empty numeric attribute list and a user attribute list with optional content.

Offset	Size	Type	Description
0	11	*BlockHeader*	As defined in Table 6.
11	Variable	*SceneHeader*	As defined in Table 16.
0	Variable	*NumAttrList*	Container properties (unused in this version.)
Variable	Variable	*UserAttrList*	User attributes.

Table 18: Container block fields (in addition to common scene object fields).

SharedMethod (ID 91)

A Shared Method is a Block that contains a Method that can be shared between multiple materials.

In the AWD-Block-Structure it must appear before any materials that is using it.

Offset	Size	Type	Description
0	11	*BlockHeader*	As defined in Table 6.
11	Variable	*VarString*	Look-up name.
Variable	Variable	*NumAttrList*	Method
Variable	Variable	*UserAttrList*	User attributes.

Table 18: Container block fields (in addition to common scene object fields).

Command Block(ID 253)

A Command Block is used to execute a action defined by the block CommandID. In AWD2.1 the only available actions are: “PutObjectIntoSceneGraph” and “CopyObjectIntoSceneGraph”.

Offset	Size	Type	Description
0	11	*BlockHeader*	As defined in Table 6.
11	Variable	*SceneHeader*	As defined in Table 16.
0	Variable	*NumAttrList*	Command properties
Variable	Variable	*UserAttrList*	User attributes.

Table 18: Container block fields (in addition to common scene object fields).

3. LightPicker (ID 51)

In Away3d a LightPicker tells a material, by which scene-lights it should be lit.

A LightPicker-Block in AWD is made of a name, and a list of AWD-Block-IDs, pointing to Light-Blocks.

Offset	Size	Type	Description
0	11	*BlockHeader*	As defined in Table 6.
11	Variable	*VarString*	Look-up name.
Variable	2	*uint16*	Number lights
Variable	Variable	*List of BlockAddr*	LightIDs
Variable	Variable	*UserAttrList*	User attributes.

Table 19: MeshInstance fields (in addition to common scene object fields.)

4. Light (ID 41)

Light blocks represent light-sources in the scene that are used for lighting/shading of objects with compatible materials.

Light blocks have a type field that defines what kind of light source the block represents, and a numeric attribute list with lamp properties.

Lights are used by LightPicker-Blocks, so they have to be parsed before this LightPicker.

This restriction excludes the LightPicker-Block of beeing a valid Block for the Scene.graph.

Instead of the local-tranformation matrix, the AWD-exporter should write the global-tranformation-matrix into the Sceneheader of a light-object. This provides a way to get the light-object-position for a light thats not part of the (AWD) scene-graph.

The CommandBlock (with action = “PutIntoSceneGraph”) provides a way to put the Light at its correct place in the SceneGraph.

Offset	Size	Type	Description
0	11	*BlockHeader*	As defined in Table 6.
11	Variable	*SceneHeader*	As defined in Table 16.
Variable	1	uint8	Light source type.
Variable	1	*uint8*	Number of shadow methods.(including ShadowMapper)
Variable	Variable	*NumAttrList*	Light source properties (see separate table.
Variable	Variable	*List of* shadow*Method*	List of shadow method elements, the length of which is defined by the preceding integer field.
Variable	Variable	*UserAttrList*	User attributes.

Table 20: Lamp block fields (in addition to common scene block fields).

Light source types

Type ID	Material type
1	Point Light
2	Directional Light

The light source properties are defined as a list of numeric attributes, and can contain any of the attributes defined in this table. Some properties are only used for certain types of light sources. The description column declares when this is the case.

ID	Name	Type	point	directional	Description	Default
1	radius	float32	+	-	A radius at which the light intensit starts to decay.	90000
2	falloff	float32	+	-	The radius at which the light intensity reaches zero (objects further from the light-source won't be affected.)	100000
3	Color	color	+	+	Color of the light.	0xffffff

4	Specular	float32	+	+	Intensity of specular light.	1.0
5	Diffuse	float32	+	+	Intensity of diffuse light.	1.0
6	Casts shadows	bool	+	+	Defines whether this light can cast shadows onto surfaces with compatible materials.	false
7	Ambient-Color	color	+	+		0xffffff
8	Ambient-Level	float32	+	+		1.0

Table 21: Light block properties.

1. Shadow Mapper

defaults to null. Options for **Directional** Light are [DirectionalShadowMapper, CascadeShadowMapper, NearDirectionalShadowMapper]. Options for **Point** Light are [CubeMapShadowMapper]

Newly created options for **Shadow Mapper** set to CascadeShadowMapper are [CascadeShadowMapMethod]. Newly created options for **Shadow Mapper** set to NearDirectionalShadowMapper are [NearShadowMapMethod]. Otherwise, newly created

Type ID	Shadow Mapper type	Category
1501	DirectionalShadowMapper	Mapper for DirectionalLights
1502	NearDirectionalShadowMapper	
1503	CascadeShadowMapper	
1504	CubeMapShadowMapper	Mapper for PointLights

Jidder

DirectionalShadowMapper (methodID = 1501)

Jidder

ID	Name	Type	Description
1125	DepthMapSize	*uint8*	Defaults to 2048. Options are [256, 512, 2048]

Jidder

NearDirectionalShadowMapper (methodID = 1502)

Jidder

ID	Name	Type	Description
1125	DepthMapSize	*uint8*	Defaults to 2048. Options are [256, 512, 2048]

1120	CoverageRatio	*float32*	default = 0.5

Jidder

CascadeShadowMapper (methodID = 1503)

Jidder

ID	Name	Type	Description
1125	DepthMapSize	*uint8*	Defaults to 2048. Options are [256, 512, 2048]
1128	Number of Cascades	*uint32*	Defaults to 3. Options are [1, 2, 3, 4]

Jidder

CubeMapShadowMapper (methodID = 1601)

Jidder

ID	Name	Type	Description
1125	DepthMapSize	*uint8*	Defaults to 2048. Options are [256, 512, 2048]

Jidder

2. Shadow Methods

Type ID	Shadow method type	Category
1001	**CascadeShadowMapMethod**	Mapped-methods
1002	**NearShadowMapMethod**	
1101	**FilteredShadowMapMethod**	Methods
1102	**DitheredShadowMapMethod**	
1103	**SoftShadowMapMethod**	
1104	**HardShadowMapMethod**	

Jidder

CascadeShadowMapMethod (methodID = 1001)

Jidder

ID	Name	Type	Description

1130	baseMethod	*Method*	not needed

Jidder

1. NearShadowMapMethod (methodID = 1002)

Jidder

ID	Name	Type	Description
1130	baseMethod	*Method*	not needed

2. FilteredShadowMapMethod (methodID = 1101)

Jidder

ID	Name	Type	Description
3	Alpha	*float32*	1
1114	Epsilon	*float32*	0.002

3. DitheredShadowMapMethod (methodID = 1102)

Jidder

ID	Name	Type	Description
3	Alpha	*float32*	1
1114	Epsilon	*float32*	0.002
1127	Samples	*uint32*	5
1119	Range	*float32*	1

4. SoftShadowMapMethod (methodID = 1103)

Jidder

ID	Name	Type	Description
3	Alpha	*float32*	1
1114	Epsilon	*float32*	0.002
1127	Samples	*uint32*	5
1119	Range	*float32*	1

5. HardShadowMapMethod (methodID = 1104)

Jidder

ID	Name	Type	Description
3	Alpha	*float32*	1
1114	Epsilon	*float32*	0.002

Skybox (ID 31)

A SkyBox renders a 360° panorama arround your scene. The origin of this panaorama will always appear to be the viewers position.

The SkyBox makes use of a CubeTextureBlock, to access the 6 bitmaps it needs.

Offset	Size	Type	Description
0	11	*BlockHeader*	As defined in Table 6.
11	Variable	*VarString*	Look-up name.
Variable	4	*BlockAddr*	Block-ID of CubeTextureBlock
Variable	Variable	*NumAttrList*	SkyBox properties (none in this version.)
Variable	Variable	*UserAttrList*	SkyBox user attributes.

Table 22: Skybox fields

5. Camera (ID 42)

Camera blocks in AWD documents represent any type of camera supported by the format specification. The type of lens is the only required field in the block, and other properties (e.g. type-specific properties) are defined in a numeric attribute list.

Offset	Size	Type	Description
0	11	*BlockHeader*	As defined in Table 6.
11	Variable	*SceneHeader*	As defined in Table 16.
Variable	Variable	*Method*	LensMethod
Variable	Variable	*UserAttrList*	User attributes.

Table 23: Camera block fields (in addition to common scene object fields.)

Camera properties and their data types are defined in the table below. Some properties are limited to a specific type of camera or set of camera types. The description column declares when this is the case.

7. Lens-Methods

A LensMethod creates a Lens used by a Camera or TextureProjector.

ID	Name	Type	Description
5001	**PerspectiveLensMethod**		
5002	1. **OrthographicLensMethod**		
5003	2. **OrthographicOffCenterLensMethod**		

Table 24: LensMethods

3. PerspectiveLensMethod (methodID = 5001)

Jidder

ID	Name	Type	Description
1501	FOV	*float64*	

4. OrthographicLensMethod (methodID = 5002)

Jidder

ID	Name	Type	Description
1502	Projection Height	*float64*	

5. OrthographicOffCenterLensMethod (methodID = 5003)

Jidder

ID	Name	Type	Description
1503	x	*float64*	
1504	y	*float64*	

1. TextureProjector (ID 43)

Textureprojector is used together with the ProjectiveTextureMethod.

Offset	Size	Type	Description
0	11	*BlockHeader*	As defined in Table 6.

11	Variable	*SceneHeader*	As defined in Table 16.
Variable	1	*uint8*	Lens/perspective type. 1: Perspective lens 2: Orthographic
Variable	4	*BlockAddr*	The Texture used.
Variable	Variable	*UserAttrList*	User attributes.

Table: TextureProjector block fields (in addition to common scene object fields.)

2. MeshInstance (ID 23)

MeshInstance blocks define what is probably the most common item in a scene, mesh objects. The geometry is defined by a geometry block elsewhere in the file, and can hence be re-used by several mesh instances.

In addition to the common scene block fields, mesh instances define a reference to the mesh data block, as well as a numeric property list and user attributes.

Offset	Size	Type	Description
0	11	*BlockHeader*	As defined in Table 6.
11	Variable	*SceneHeader*	As defined in Table 16.
Variable	4	*BlockAddr*	ID of mesh data block.
Variable	2	*uint16*	Number of materials
Variable	Variable	*List of BlockAddr*	Material IDs
Variable	Variable	*NumAttrList*	Mesh instance properties
Variable	Variable	*UserAttrList*	Mesh instance user attributes.

Table 25: MeshInstance fields (in addition to common scene object fields.)

ID	Name	Type	Description	Default
1	CastShadows	*bool*	If the mesh Instance cast a shadow	false

Table 26: MeshInstance block properties.

8. Material blocks

1. BitmapTexture block (ID 82)

The bitmap texture block defines a bitmap, either as an external file or as embedded image data, that can be referenced and used by other blocks.

Offset	Size	Type	Description
0	11	*BlockHeader*	As defined in Table 6.
11	Variable	*VarString*	Look-up name
Variable	1	uint8	Image type 0: External 1: Embedded
Variable	4	uint32	Data length
Variable	Variable	*ByteArray* or *ConstString*	Image data (JPEG/PNG/ATF (?) file stream) or URL to external file.
Variable	Variable	*NumAttrList*	Texture properties (none in this version.)
Variable	Variable	*UserAttrList*	Texture user attributes.

Table 27: Texture block fields

Jidder

2. CubeTexture block (ID 83)

Jidder

Offset	Size	Type	Description
0	11	*BlockHeader*	As defined in Table 6.
11	Variable	*VarString*	Look-up name
Variable	4	*BlockAddr*	Texture to be used in the positive X direction.
Variable	4	*BlockAddr*	Texture to be used in the negative X direction.
Variable	4	*BlockAddr*	Texture to be used in the positive Y direction.
Variable	4	*BlockAddr*	Texture to be used in the negative Y direction.
Variable	4	*BlockAddr*	Texture to be used in the

			positive Z direction.
Variable	4	*BlockAddr*	Texture to be used in the negative Z direction.
Variable	Variable	*NumAttrList*	CubeTexture properties (none in this version.)
Variable	Variable	*UserAttrList*	CubeTexture user attributes.

Table 28: CubeTexture block fields

Jidder

3. SimpleMaterial block (ID 81)

Simple material blocks represent materials that exist in Away3D already. This means that mere property values are enough to configure the material (as opposed to custom shader materials which require shader code to be embedded within the AWD file.)

Offset	Size	Type	Description
0	11	*BlockHeader*	As defined in Table 6.
11	Variable	*VarString*	Look-up name
Variable	1	uint8	Material type (see separate table)
Variable	1	uint8	Number of shading methods.
Variable	Variable	*NumAttrList*	Material properties (see separate table.)
Variable	Variable	*List of ShaderMethod*	List of shader method elements, the length of which is defined by the preceding integer field.
Variable	Variable	*UserAttrList*	Material user attributes.

Material types

Type ID	Material type
1	Color material
2	Texture material

Material properties

ID	Name	Type	Color Material	MP	Texture Material	MP	Description	Default
1	Color	color	+	+	-	-	Color (used only by color materials)	0xffffff
2	Texture	*BlockAddr*	-	-	+	+	Reference to texture block (used only by bitmap materials)	null
3	NormalTexture	*BlockAddr*	+	+	+	+	NormalTexture	null
4	spezialID	*uint8*	-	-	-	-	0: SinglePass 1:MultiPass 2:SkyBox	0
5	smooth	*bool*	+	+	+	+	Default to true	true
6	mipmap	*bool*	+	+	+	+	Default to true	true
7	bothSides	*bool*	+	+	+	+	Default to false	false
8	Pre-multiplied	*bool*	+	+	+	+	Default to false	false
9	BlendMode	uint8	+	+	+	+	[0: NORMAL,1:ADD, 2: ALPHA, 8: LAYER, 10: MULTIPLY]	0
10	Alpha		+	-	+	-	Overall alpha of material.	1.0
11	Alpha blending	*bool*	+	-	+	-	Defines whether alpha blending (semi-transparency) should be enabled for this material.	false
12	Binary alpha threshold	float32	+	+	+	+	Defines a cut-off threshold for the alpha channel when not using alpha blending. Pixels with alpha over this value wil	0.0

								be fully opaque, and all other pixel will be completely transparent.	
13	Repeat	*bool*	+	+	+	+	Defines whether to repeat this material over the surface of meshes for which the UV coordinates are outside the 0-1 span. Default false	True
14	Diffuse Level	float32	-	-	-	-	may be needed in later versions	1.0
15	Ambient level	float32	+	+	+	+		1.0
16	Ambient Color	*color*	+	+	+	+		0xffffff
17	Ambient texture	*BlockAddr*	-	-	+	+		null
18	Specular Level	float32	+	+	+	+		1.0
19	Specular Gloss	float32	+	+	+	+		50
20	Specular Color	color	+	+	+	+		0xffffff
21	Specular Texture	*BlockAddr*	+	+	+	+		null
22	LightPicker	*BlockAddr*	+	+	+	+		null

Table 29: Dynamic properties for material blocks.

Shading Methods

A shading method defines a way that a material's surface is rendered, e.g. with regards to light. Some methods require special treatment, e.g. diffuse and specular shading methods, and for this reason shading methods are sorted into different categories.

The Base-Method of a Composite-Method must always be parsed before the Composite-Method.

1.

Type ID	Shading method type	Category
1	EnvMapAmbientMethod	Ambient
51	DepthDiffuseMethod (no properties)	Diffuse
52	GradientDiffuseMethod	Diffuse

53	WrapDiffuseMethod	Diffuse
54	LightMapDiffuseMethod	DiffuseComp
55	CellDiffuseMethod	DiffuseComp
56	SubSurfaceScatteringMethod	DiffuseComp
101	AnisotropicSpecularMethod (no properties)	Specular
102	PhongSpecularMethod (no properties)	Specular
103	CellSpecularMethod	SpecularComp
104	FresnelSpecularMethod	SpecularComp
151	HeightMapNormalMethod	Normal
152	SimpleWaterNormalMethod	Normal
401	ColorMatrix	EffektShader
402	ColorTransform	EffektShader
403	EnvMap	EffektShader
404	LightMapMethod	EffektShader
405	ProjectiveTextureMethod	EffektShader
406	RimLightMethod	EffektShader
407	AlphaMaskMethod	EffektShader
408	RefractionEnvMapMethod	EffektShader
409	OutlineMethod	EffektShader
410	FresnelEnvMapMethod	EffektShader
411	FogMethod	EffektShader

Jidder

GradientDiffuseMethod (methodID = 52)

Jidder

ID	Name	Type		Description
104	gradient-texture	*Blockaddr*		

Jidder

1. WrapDiffuseMethod (methodID = 53)

Jidder

ID	Name	Type		Description

1118	warp factor	*float64*		

2. **LightMapDiffuseMethod (methodID = 54)**

Jidder

ID	Name	Type		Description
1124	blendMode	*uint8*	*MULTIPLY*	options: add / multiply
105	lightMap-texture	*blockaddr*	*default_texture*	
1130	baseMethod	*Method*	*BasicDiffuse*	

Jidder

3. **CellDiffuseMethod (methodID = 55)**

Jidder

ID	Name	Type		Description
1123	levels	*uint8*	*3*	
1117	smoothness	*float32*	*0.1*	
1130	baseMethod	*Method*	*BasicDiffuse*	

4.

5. **SubsurfaceScatteringDiffuseMethod (methodID = 56)**

Jidder

ID	Name	Type		Description
1140	`Scattering`	*float32*	*0.2*	
1141	`Translucency`	*float32*	*1*	
1142	`Scatter Color`	*color*	*0xFFFFFF*	
1130	baseMethod	*Method*		

6.

7.

8. **CellSpecularMethod (methodID = 103)**

Jidder

ID	Name	Type	Description
1115	cut-off	*float32*	

1117	smoothness	*float32*	
1130	baseMethod	*Method*	

9.

10. FresnelSpecularMethod (methodID = 104)

Jidder

ID	Name	Type	Description
204	BasedOnSurface	*bool*	
1106	power	*float32*	
1113	**Reflectance**	*float32*	
1130	baseMethod	*Method*	not need to be saved

11. HeightMapNormalMethod (methodID = 151)

Jidder

ID	Name	Type		Description
1108	World width	*float32*		5
1109	World height	*float32*		5
1110	World depth	*float32*		5

12. SimpleWaterNormalMethod (methodID = 152)

Jidder

ID	Name	Type	Description
103	Second Normal map	*blockaddr*	

13.

14. EnvMapAmbientMethod (methodID = 1)

Jidder

ID	Name	Type	Description
101	cubeTexture	*BlockAddr*	

Jidder

EffectMethods

EffectMethods are stored in SharedMethod-Blocks

15. ColorMatrixMethod (methodID = 401)

Jidder

ID	Name	Type	Description
1001	matrix	ConstList of *float32*	default = identity matrix

Jidder

16. ColorTransformMethod (methodID = 402)

Jidder

ID	Name	Type	Description
1101	alphaMultiplier	float32	1
1102	redMultiplier	float32	1
1103	greenMultiplier	float32	1
1104	blueMultiplier	float32	1
1105	colorOffset	color	0x00000000

Jidder

17. EnvMapMethod (methodID = 403)

Jidder

ID	Name	Type	Description
101	cubeTexture	*BlockAddr*	DefaultCubeTexture
3	alpha	float32	1
1146	mask	BlockAddr	Default-Texture

Jidder

LightMapMethod (methodID = 404)

Jidder

ID	Name	Type	Description
1124	blendMode	*uint8*	default MULTIPLY(10) / other option= ADD(1)
100	texture	BlockAddr	Default_Texture

ProjectiveTextureMethod **(methodID = 405)**
Jidder

ID	Name	Type	Description
1124	mode	*uint8*	Default=MULTIPLY options: ADD MIX
102	textureProjector	BlockAddr	default = null

RimLightMethod (methodID = 406)

Jidder

ID	Name	Type	Description
1	color	*color*	0xffffff
1107	Strength	float32	0.4
1106	power	float32	2

AlphaMaskMethod (methodID = 407)

Jidder

ID	Name	Type	Description
203	UseSecondaryUV	*bool*	false
100	texture	BlockAddr	Default_texture

RefractionEnvMapMethod (methodID = 408)

Jidder

ID	Name	Type	Description
101	envMap(CubeTexture)	BlockAddr	Default_Cube_texture
1129	RefractionIndex	int32	0.1
1111	Dispersion R	float32	0.01
1143	Dispersion G	float32	0.01
1144	Dispersion B	float32	0.01
3	Alpha	float32	1

OutlineMethod (methodID = 409)

Jidder

ID	Name	Type	Description
2	OutlineColor	color	0x00000000
1121	OutlineSize	float32	1
202	ShowInnerLines	bool	true
201	DedicatedMesh	bool	false

FresnelEnvMapMethod (methodID = 410)

Jidder

ID	Name	Type	Description
101	envMap(CubeTexture)	BlockAddr	Default_Cube_Texture
3	alpha	float32	1

FogMethod (methodID = 411)

Jidder

ID	Name	Type	Description
1122	Min-Distance	float32	0
1145	Max-Distance	float32	1000
1	Color	color	0x808080

9.

10. Animation blocks

1. Skeleton (ID 101)

The Skeleton block defines a skeletal hierarchy of joints that can be bound to by any mesh (see the MeshData block for details on how to bind a mesh to a skeleton.)

Offset	Size	Type	Description
0	Variable	VarString	Look-up name
Variable	2	uint16	Number of joints
Variable	Variable	*NumAttrList*	Skeleton properties (none in this version.)
Variable	Variable	List of *SkeletonJoint*	A list of joints
Variable	Variable	*UserAttrList*	Skeleton user attributes.

Table 30: Structure of a skeleton block.

The skeleton block contains a list of joints which are the "bones" in the skeleton. The internal structure of a joint in AWD is defined by the next section. There can be virtually any number of joints (limited only by the 16-bit integer defining the number) in a skeleton block, but the receiving engine might have harder restrictions on how many joints it can handle.

1. Skeleton joint

The skeleton joint defines a deformable and bindable "bone" in the skeletal hierarchy. The data inside a joint structure defines it's name and parent-child relationships, as well as the "bind" transform, which describes the transformational state that the joint should be in when vertices are bound to it.

Offset	Size	Type	Description
0	2	uint16	Joint ID
4	2	uint16	Parent joint ID
Variable	Variable	VarString	Look-up name
Variable	Variable	*Matrix4x3*	Bind pose transform
Variable	Variable	*NumAttrList*	Joint properties (none in this version.)
Variable	Variable	*UserAttrList*	Joint user attributes.

Table 31: Structure of joints inside a skeleton block

Each joint has an ID, which among other things is used by other joints to define the parent joint. A parent joint ID of -1 means there is no parent, i.e. that the defining joint is the root. The joint ID is also used to reference a particular joint from the joint index data stream in geometry blocks for binding.

2. SkeletonPose (ID 102)

The skeleton pose block defines the transformations for all bones in a skeleton such that the skeleton assumes a particular static pose. These can then be used to position a mesh (e.g. a character) statically, or in a frame-by-frame animation using a SkeletonAnimation block (see the next section.)

A pose comprises a list of joint transform structures, which in turn define the transformation for each of the joints in the skeleton. It also defines the length of said list, and a name.

There is no hard binding between a skeleton pose and a particular skeleton. It's up to the logic of the parsing party (e.g. a game engine) to apply the pose to a compatible skeleton. This allows for the same poses (and hence animations) to be used for several different skeletons, as long as they have the same structure.

The below table defines the structure of the SkeletonPose block.

Offset	Size	Type	Description
0	Variable	VarString	Look-up name
Variable	2	uint16	Number of joint transformations
Variable	Variable	*NumAttrList*	Pose properties (none in this version.)
Variable	Variable	List of *JointTransform*	A list of joint transformations
Variable	Variable	*UserAttrList*	Pose user attributes.

Table 32: Structure of SkeletonPose block.

1. Skeleton pose joint transform

The joint transform element, a list of which is contained within the skeleton pose block, comprises a transform matrix for a single joint. The order of the joint transform elements within the pose block should be the same as the order of the joint elements in the skeleton block, that is depth-first incrementally recursive.

The first field of the joint transform element is a boolean which indicates whether there is a transformation defined for the joint represented by this element. If true, the next field is a 4x4 matrix. If false, there is no second field and the next piece of data will be the next joint transform (if any.)

Offset	Size	Type	Description
0	1	bool	Defines whether joint ha transformation
1	Variable	Matrix4x3	Transformation for joint (if any)

Table 33: Structure of a joint transform element inside a SkeletonPose block.

3. SkeletonAnimation (ID 103)

Skeleton animation blocks define actual animation of a skeleton as frame-by-frame poses. The term "frame" is used for a point in time at which an exact pose is defined in the file format. It does not necessarily coincide with a refresh in the playback engine. Instead, these frames should be regarded as keyframes, and the actual output during playback be calculated by interpolating two subsequent keyframes.

Offset	Size	Type	Description
0	Variable	VarString	Look-up name

Variable	2	uint16	Number of frames
Variable	Variable	*NumAttrList*	Animation properties (none in this version.)
Variable	Variable	List of *SkelAnimFrame*	Frames as a list of skeleton animation fram structures (see below.)
Variable	Variable	*UserAttrList*	Animation user attributes.

Table 34: Structure of SkeletonAnimation blocks.

1. Skeleton animation frames

The frames in a skeleton animation are defined in a list of skeleton animation frame structures, which are explained in the structure table below. They consist of a reference to a skeleton pose block, as well as a duration in milliseconds for that frame. This allows for variable-duration frames, which would typically be interpolated between by the animation engine.

Offset	Size	Type	Description
0	4	*BlockAddr*	ID of skeleton pose block.
4	2	uint16	Duration in milliseconds

Table 35: Structure of SkelAnimFrame elements.

4. UVAnimation (ID 121)

Jidder

Offset	Size	Type	Description
0	Variable	*VarString*	Look-up name
Variable	2	uint16	Number of frames
Variable	Variable	*NumAttrList*	Animation properties
Variable	Variable	List of *UVAnimFrame*	UV animation frames
Variable	Variable	*UserAttrList*	Animation user attribute

Jidder

Mekk

Offset	Size	Type	Description
0	24	*Matrix3x2*	Two-dimensional UV transformation.
24	1	uint16	Duration in milliseconds

11. Miscellaneous blocks

1. Namespace blocks (ID 254)

The namespace block is a block that must exist in any AWD file with user extensions. A namespace block couples a short numeric "namespace handle" which is unique within the file, with a namespace string identifier which should be globally unique. To make sure that namespace string identifiers are unique, good practice is using a URI with a domain that is controlled by the defining party, e.g. http://www.away3d.com/prefab/awpns for the Prefab3D AWP project format.

The block body itself consists of an 8-bit integer for the numeric ID, and a variable string for the URI/string identifier, as defined by the below table.

Offset	Size	Type	Description
0	1	*uint8*	Namespace handle (Zero is reserved for nul namespaces.)
1	Variable	*VarString*	Namespace URI/string identifier.

Table 36: Structure of a single namespace definition, multiple of which can occur in a namespace list block.

> NOTE: Zero must not be used as a numeric namespace ID. It is reserved for use as a null reference when a block or user attribute does not have a namespace.

See the section on "Extending AWD" for more information about how to use namespaces.

2. Meta-data blocks (ID 255)

The meta-data block is something that encoders can optionally include in an AWD document to define meta-data such as creation date, name and version of encoder et c. Only one meta-data block should exist in an AWD document, and it should occur at the very start of the block list.

The structure is very simple, comprising only a numeric property list.

Offset	Size	Type	Description
0	Variable	*NumAttrList*	Meta-data properties.

The properties of a meta-data blocks are defined by the following table.

ID	Name	Type	Description
1	Timestamp	awd_uint32	Generation date and time defined as seconds since the Epoch (00:00, 1/1 1970.)
2	Encoder name	*ConstString*	Name of encoder (i.e. the library or tool used to

			encode the AWD file, e.g. libawd.)
3	Encoder version	*ConstString*	Encoder version.
4	Generator name	*ConstString*	Name of generator (i.e. the tool used to create the content, e.g. Maya.)
5	Generator version	*ConstString*	Generator version.

Jidder

3.

Part III: Using AWD

1. Official AWD tools

The official AWD tool-chain is a constantly growing set of tools, importers/exporters and programming language libraries and extensions. This section details the tools available at the time of writing. Visit the AWD project page on Google Code (http://code.google.com/p/awd) for the latest information about available AWD tooling.

1. AWD SDK

The AWD SDK has been created to aid and greatly simplify the creation of AWD importers and exporters. Furthermore, it exists to help prevent discrepancies between encoders and decoders. An AWD file that was encoded using the AWD SDK should be expected to structurally conform to the AWD specification, and the AWD SDK can be used to reliably decode a conforming AWD file.

Whether the content is logically conforming (e.g. whether the contents of a data stream has been sequentially ordered in accordance with what the specification dictates) is still up to the encoder programmer. Below are the modules of the AWD SDK.

For more information on the AWD SDK, how to build it and use it, see the AWD Google Code page. Below is a list of the programming languages supported in the AWD SDK.

1. The libawd library for C++

The main reference implementation of AWD is the libawd C++ library.

2. PyAWD - AWD for Python

PyAWD is a Python module for working with AWD files. It's available as a Python binding and object-oriented wrapper for the C libawd library, or as a standalone python library that does not require libawd to run (but does not perform as well as the libawd wrapper.)

2. Official importer/exporter implementations

These are importers and exporters that are being officially developed and maintained as a part of the AWD project, and that are either in a usable state or planned at the time of releasing this document. Please visit the AWD project page on Google Code for the latest set of importers/exporters, and for links to any known community implementations.

Vendor/Application	Export	Import	Real-time
Prefab3D	Yes	Yes	No
Blender	Yes	No	No
Autodesk Maya	Yes	No	No

2. Extending the AWD format

AWD is user-extendable by the means of user attributes and user blocks. Attributes are a versatile way to augment an already existing block type, such as a mesh instance or a material, with custom properties. User blocks on the other hand can be used to add top-level data types to the format, like player spawn points in a game, force fields in a physics simulator, or a list of configuration settings in an editor.

1. User attributes

User attributes are key/value pairs with plaintext keys that can be appended to most AWD block types. These can be utilized by user applications to augment AWD blocks with application-specific properties, such as physics properties or game settings.

See the Part II section called "Attributes" for more information on how to add user attributes to a block.

2. User blocks

The term "user block" refers to a block type that is not defined by the AWD file format specification, but rather by an extending entity (i.e. a file format user.) User blocks share the same block header as any other block, but must be defined in a non-null namespace to distinguish them from AWD blocks.

3. Namespaces in AWD

When extending AWD, there is a need to mark those blocks that do not belong to standard AWD as belonging to some other context, a "namespace". That way the same numeric block type identifier can be used for both a standard AWD block and a block defined by the user application.

There is also the rare case where a single AWD file has been influenced by several separate encoders in which two different user attributes have the same key/name. To prevent files like these from being incorrectly parsed by user-extended parsers, any encoder that extends AWD must use namespaces with user attributes.

Namespaces serve the purpose of coupling a user attribute or user block with an identifier that is guaranteed to be unique, such as a URI. The AWD namespaces are inspired by those in XML, where a namespace URI is defined once in a document and any element belonging to that namespace subsequently identifies the namespace using a shortened ID.

In AWD, the shortened ID is an 8-bit unsigned integer that is defined in a namespace block, and then referenced in every user attribute and user block.

1. Using the namespace block

An encoder that extends AWD must insert a namespace block before any user block or block with user attributes appears in the document. It's good practice to put the namespace block first in the file. See the section "Namespace block" for the exact structure of this block.

2. Picking a namespace identifier

A namespace identifier can be any string that fits in a VarString. The main requirement is that it is unique within the file where it's used, but it lies in the interest of the user application that it is also consistent and globally unique, so that a user parser can identify blocks belonging to it's namespace. A user application hosted at example.com could use "http://example.com/awdns" as it's AWD namespace identifier, which can be assumed to be unique not only in a particular file, but also consistent and universal so that it can be hard-coded into the custom parser.

If an encoder intends to create a namespace in a file, any existing namespace definitions must be inspected by the encoder so that an ambiguous identifier is not added (e.g. if the original file was encoded by the same encoder and already contains user blocks or attributes in the relevant namespace.) Two namespaces within a document must not have the same numeric handle or string identifier.

3. Parsing an AWD document

AWD is designed for linear parsing, or even streaming and "block-wise" parsing of such a stream. It should never be necessary to seek backwards in a file, and unless a particular type of content has not been implemented in the parser and thus is skipped, even forward seeking is rare.

1. Handling block references

Because internal block references are always made backwards, node B can only refer to node A if B occurs after A in the document.

As a block is read, the parser must determine whether to store a reference to it's internal representation of that block depending on it's ID. If the block ID is zero, this is a promise from the encoder that there will be no references to this block in the document, so it is not necessary for the parser to hold on to it. However, if the block ID is greater than zero, the parser should store a reference to it's internal representation of that block in a lookup table by ID. When any reference is encountered, this lookup table can be used for random access to the correct block representation by ID.

Using Away3D as an example, when the parser encounters a TriangleGeometry block, it will create a Geometry instance and store it in a vector with numeric indices. When a MeshInstance occurs with a reference to this geometry block, an instance of the Away3D Mesh class will be created and it's geometry property assigned to the previously created Geometry instance. The latter can easily be retrieved from the lookup vector using the reference ID in the MeshInstance block.

2. Handling unrecognized elements

Because AWD can be extended both by users and future versions of the format, a conforming parser needs to be able to deal with blocks that it does not recognize.

If a block is encountered that uses an unknown block ID or namespace, the entire block should be skipped using the size field that is always defined in the common block header. Blocks in unknown namespaces can always be skipped unless a parser is expecting some kind of user-defined block. A parser library should delegate user blocks (blocks in unknown namespaces) to the application code so that it's up to the application logic to decide whether they need to be parsed.

3. Parsing extended AWD documents

User-defined blocks will always contain a reference to a namespace other than the default Null namespace (see section on User-defined blocks and Extending AWD). That way a particular user block can be analyzed and a decision can be made whether it's in a namespace that the parser is expecting, or whether it should be skipped.

Namespaces must be defined early (usually first) in the file through the use of namespace blocks. From the content of such blocks, a parser can create a look-up table for namespace identifier strings, and when a namespace reference occurs determine from the look-up table whether the user block or attribute is in a namespace which it expects and understands.

Another common way for extension are the user attributes that can be attached to almost any block. As with user blocks, attributes have a length field, allowing them to be skipped if the attribute key, value type, or the namespace in which the attribute is defined, is unrecognized. Optionally, if user attributes are concluded to never be relevant, a parser can skip all attributes belonging to a block using the length field of the attribute list.

See the section Attributes fore more information about attributes.

4. AWD Limitations

There are some limitations inherent with the way AWD is designed. The following is a list of such limitations (per file unless otherwise stated.)

1. General limitations

Feature	Limit	Reason
AWD file size (min)	11 bytes	Size of header.
AWD file size (max)	4 GB	Limited to max value of the body length header field (32 bit unsigned integer.) Not applicable to streams.
Number of blocks	>4 billion	32-bit block address.
Number of namespaces	256 (incl. Null)	8-bit namespace handles.
Block types (per namespace)	256	8-bit block type fields.
Block types (total)	65535	16 bits total for namespace and block types.
Block data length	4GB	32-bit block length field.

2. AWD data type limitations

Feature	Limit	Reason
Length of VarStrings	65535 single-byte characters, les multi-byte characters.	16-bit length field.

Number of numeric attributes (pe list)	65535	16-bit ID field
Number of text attributes (per lis	>400 million	Assuming attribute names with three bytes/characters and a single-byte value (e.g. Boolean). Limited by attribute element length and 32-bit list length identifier.

3. Geometry limitations

Feature	Limit	Reason
Materials/sub-meshes per mesh or sub-paths per path.	65535	16 bit length field
Mesh vertices	>350 million	32 bit stream length field, 12 bytes per vertex (optimized for size.)
Mesh triangles	>350 million	32 bit stream length field, 12 bytes per triangle (optimized for size.)
Path quadratic segments	>119 million	32 bit stream length field, 12 bytes per point (optimized for size), 3 points per segment.
Path cubic segments	>89 million	32 bit stream length field, 12 bytes per point (optimized for size), 4 points per segment.

5. AWD Structure examples

Below are some examples of simple AWD files to illustrate the structure of an uncompressed file.

The *[N]* symbol illustrates a link (numeric reference) to another block with the ID N. Even though the indentation of the lists in these examples might imply a tree structure, such structure only exists logically, whereas the actual "physical" representation of data in the file is linear, as is the parsing.

Removed temporarily while the format is still in motion.