Away3D Data format (AWD) v2.1 Alpha

Last updated: May 12th 2013

This is a work-in-progress draft of the file format specification for the binary (second-generation) Away3D data format (AWD).

This AWD2.1 Specification is an extension of the AWD2.0 Specification

Any community feedback is welcome at:

- https://github.com/awaytools
- http://www.away3d.com/forum
- r@richardolsson.se
- 80prozent@differentdesign.de

Please note that this is an incomplete document, in an early pre-release state.

Also note that the tools that are available at https://github.com/awaytools are under development and should not be expected to conform fully to this specification at any point before it's final release.

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 are things that are not yet supported by the lates AWD2parser yet.

Inhaltsverzeichnis

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.

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.

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.

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.

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.

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.

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.

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.

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.

Terminology

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.

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.

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.

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.

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.

Part II: File format specification

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

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

Flags

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 file is streaming and that more blocks can be expected 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

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.

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.

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.

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.

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.

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.

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.

Numbers

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).

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

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.

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.

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.)

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.

ConstString

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

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.

Lists

ConstList

TBD

VarList

TBD

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.

N-dimensional vector

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

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.

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.)

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.

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

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

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.

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.

TypedPropertiesLists

A TypedPropertiesList is used for example to store Shading- /ShadowMap- /and EffectMethods.

It stores a type-ID and a list of properties.

Offset	Size	Type	Description
0	2	uint16	TypedPropertiesList-TypeID
2	Variable	*NumAttrList*	property-list
Variable	Variable	*UserAttrList*	User properties for method

The TypePropertiesList-TypeID is defined by the context the TypePropertiesList is used in.
Please look into the specification of Shading- / ShadowMap- /and EffectMethods for more information.

All TypedProperetiesLists are using the same set of Properties-IDs.
As shown in the next Table, for every Value-Type, 100 IDs are reserved.
If there will be a need for more IDs per Value-Type, this table can easy be extended.

TypedProperties-IDs

ID	Type
1-100	*BlockAddr*
101-200	*float32/float64*
201-300	*uint32*
301-400	*uint16*
401-500	*uint8*
501-600	*String*
601-700	*Color*
701-800	*Bool*
801-900	*Matrix(list of float32/float64)*

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.

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.

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

Bit	Value	Name	Description
00	0x0001	High precision	Defines whether to use float32 (bit unset) or float64 (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.

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.

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.

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	AssetType/AssetEvent
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	92	ShadowMethod	Method	
0	101	Skeleton	Animation	
0	102	SkeletonPose	Animation	
0	103	SkeletonAnimation	Animation	
0	111	MeshPose	Animation	
0	112	VertexAnimation	Animation	
0	113	AnimationSet	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

Geometry blocks

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 2.1: Top-level structure of a MeshData block

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

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).

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.

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.

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.

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.

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.

Primitves are making use of the same IDs as a Typed-Properties-List.

PlaneGeometry (Primitve-Type = 1)

ID	Name	Type	Default
101	width	*float32*	100
102	height	*float32*	100
301	segmentsW	*uint16*	1
302	segmentsH	*uint16*	1
701	yup	*bool*	true
702	doubleSided	*bool*	false

CubeGeometry (Primitve-Type = 2)

ID	Name	Type	Default
101	width	*float32*	100
102	height	*float32*	100
103	depth	*float32*	100
301	segmentsW	*uint16*	1
302	segmentsH	*uint16*	1
303	segmentsD	*uint16*	1
701	tile6	*bool*	true

SphereGeometry (Primitve-Type = 3)

ID	Name	Type	Default
101	radius	*float32*	50
301	segmentsW	*uint16*	16
302	segmentsH	*uint16*	12
701	yup	*bool*	true

CylinderGeometry (Primitve-Type = 4)

ID	Name	Type	Default
101	topRadius	*float32*	50
102	bottomRadius	*float32*	50
103	height	*float32*	100
301	segmentsW	*uint16*	16
302	segmentsH	*uint16*	1
701	topClosed	*bool*	true
702	bottomClosed	*bool*	true
703	yup	*bool*	true
704	surfaceClosed	*bool*	true

ConeGeometry (Primitve-Type = 5)

ID	Name	Type	Default
101	radius	*float32*	50
102	height	*float32*	100
301	segmentsW	*uint16*	16
302	segmentsH	*uint16*	1
701	closed	*bool*	true
702	yup	*bool*	true

CapsuleGeometry (Primitve-Type = 6)

ID	Name	Type	Default
101	radius	*float32*	50
102	height	*float32*	100
301	segmentsW	*uint16*	16
302	segmentsH	*uint16*	15
701	yup	*bool*	true

TorusGeometry (Primitve-Type = 7)

ID	Name	Type	Default
101	radius	*float32*	50
102	tubeRadius	*float32*	50
301	segmentsR	*uint16*	16
302	segmentsT	*uint16*	8
701	yup	*bool*	true

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.

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).

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
Variable	Variable	*UserAttrList*	User attributes.

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

ID	Name	Type	Description	Default
1	pivotX	*float32*		0
2	pivotY	*float32*		0
3	pivotZ	*float32*		0
4	visibility	*uint8*	Not really used yet	true

Table 26: Container block properties.

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.)

The MeshInstance-Properties includes the Container-properties.

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

Table 26: MeshInstance block properties.

Skybox (ID 31)

A SkyBox renders a 360° panorama around your scene. The origin of this panorama 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

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.

The following is not relevant for AwayBuilder atm, but could be in future versions, and will be relevant for AWDToolsC4D.

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	Variable	*NumAttrList*	Light source properties (see separate table.) the shadowMapper is includet as properties.
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 ShadowMapper is defined within the Properties.

Light Properties

P = PointLight, **D** = DirectionalLight

ID	Name	Type	P	D	Description	Default
1	radius	float32	+	-	A radius at which the light intensity 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
7	Ambient-Color	color	+	+		0xffffff
8	Ambient-Level	float32	+	+		1.0
9	ShadowMapper-Type	uint8	+	+	defines the type of shadowmapper to use. if this is !=0, castShadows=false;	0
10	DepthMapSize	uint8	+	+	shadowMapper-property - Options are [256, 512, 2048]	2048
11	CoverageRatio	float32	+	+	shadowMapper-property	0.5
12	CascadesNum	uint16	+	+	shadowMapper-property	3
21	directionX	float32	-	+	The direction of the light (x)	0
22	directionY	float32	-	+	The direction of the light (y)	-1
23	directionZ	float32	-	+	The direction of the light (z)	1

Table 21: Light block properties.

Shadow Mapper

The TypeID of the Shadowmapper is stored as Light-Property 9.
The default TypeID is 0 = no ShadowMapper.
If a ShadowMapper is set, (TypeID!=0), light.castShadows is set to true.

Type ID	Shadow Mapper type	Category
1	DirectionalShadowMapper	Mapper for DirectionalLights
2	NearDirectionalShadowMapper	Mapper for DirectionalLights
3	CascadeShadowMapper	Mapper for DirectionalLights
4	CubeMapShadowMapper	Mapper for PointLights

All ShadowMapper-Properties are stored as Light-Properties.

DirectionalShadowMapper (ShadowMapperID = 1)

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

NearDirectionalShadowMapper (ShadowMapperID = 2)

ID	Name	Type	Description
10	DepthMapSize	*uint8*	Defaults to 2048. Options are [256, 512, 2048]
11	CoverageRatio	*float32*	default = 0.5

CascadeShadowMapper (ShadowMapperID = 3)

ID	Name	Type	Description
10	DepthMapSize	*uint8*	Defaults to 2048. Options are [256, 512, 2048]
12	Number of Cascades	*uint16*	Defaults to 3. Options are [1, 2, 3, 4]

CubeMapShadowMapper (ShadowMapperID = 4)

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

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.)

Material blocks

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

A material can have a addional type-proerty set in the Materialproperties called “spezialID”.

spezialID	Material type
0 (=Default)	SinglePassMaterial
1	MultiPassMaterial

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 will be fully opaque, and all other pixels will be completely transparent.	0.0
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.	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.

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
998	ShadowMapMethodBlockMethod	ShadowMapMethodBlock
999	EffectMethodBlockMethod	EffectMethodBlock

AmbientMethods

EnvMapAmbientMethod (methodID = 1)

ID	Name	Type	Default	Description
1	cubeTexture	*BlockAddr*	*defaultTexture*	

DiffuseMethods

GradientDiffuseMethod (methodID = 52)

ID	Name	Type	Default	Description
1	gradient-texture	*Blockaddr*	*defaultTexture*	

WrapDiffuseMethod (methodID = 53)

ID	Name	Type	Default	Description
101	warp factor	*float32*	*0.5*	

LightMapDiffuseMethod (methodID = 54)

ID	Name	Type	Default	Description
401	blendMode	*uint8*	*MULTIPLY*	options: add / multiply
1	lightMap-texture	*blockaddr*	*defaultTexture*	

CellDiffuseMethod (methodID = 55)

ID	Name	Type	Default	Description
401	levels	*uint8*	*3*	
101	smoothness	*float32*	*0.1*	

SubsurfaceScatteringDiffuseMethod (methodID = 56)

ID	Name	Type	Default	Description
101	**Scattering**	*float32*	*0.2*	
102	**Translucency**	*float32*	*1*	
601	**Scatter Color**	*color*	*0xFFFFFF*	

SpecularMethods

CellSpecularMethod (methodID = 103)

D	Name	Type	Default	Description
101	cut-off	*float32*	0.5	
102	smoothness	*float32*	0.1	

FresnelSpecularMethod (methodID = 104)

ID	Name	Type	Default	Description
701	BasedOnSurface	*bool*	true	
101	power	*float32*	5	
102	Reflectance	*float32*	0.1	this is not set in the VO

NormalMethods

HeightMapNormalMethod (methodID = 151)

ID	Name	Type	Default	Description
101	World width	*float32*	5	
102	World height	*float32*	5	
103	World depth	*float32*	5	

SimpleWaterNormalMethod (methodID = 152)

ID	Name	Type	Default	Description
1	Second Normal map	*blockaddr*	*defaultTexture*	

ShadowMapMethods / EffectMethods

ShadowMapMethodBlockMethod (methodID = 52) / EffectMethodBlockMethod (methodID = 52)

ID	Name	Type	Default	Description
1	TargetBlock	*Blockaddr*	*defaultTexture*	The id of the MethodBlock

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

CubeTexture block (ID 83)

Offset	Size	Type	Description
0	11	*BlockHeader*	As defined in Table 6.
11	1	*unit8*	0: external 1: embed
12	Variable	*VarString*	Look-up name
Variable	Variable	*ByteArray* or *ConstString*	Image data (JPEG/PNG/ATF-file stream) or URL to external file.
Variable	Variable	*ByteArray* or *ConstString*	Image data (JPEG/PNG/ATF-file stream) or URL to external file.
Variable	Variable	*ByteArray* or *ConstString*	Image data (JPEG/PNG/ATF-file stream) or URL to external file.
Variable	Variable	*ByteArray* or *ConstString*	Image data (JPEG/PNG/ATF-file stream) or URL to external file.
Variable	Variable	*ByteArray* or *ConstString*	Image data (JPEG/PNG/ATF-file stream) or URL to external file.
Variable	Variable	*ByteArray* or *ConstString*	Image data (JPEG/PNG/ATF-file stream) or URL to external file.
Variable	Variable	*NumAttrList*	CubeTexture properties (none)
Variable	Variable	*UserAttrList*	CubeTexture user attributes.

Table 28: CubeTexture block fields

SharedMethod-Block(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).

EffectMethods

EffectMethods are stored in SharedMethod-Blocks, so they can be accessed by multiple Materials.

ColorMatrixMethod (methodID = 401)

ID	Name	Type	Default
801	matrix	ConstList of *float32*	default = identity matrix

ColorTransformMethod (methodID = 402)

ID	Name	Type	Default
101	alphaMultiplier	float32	1
102	redMultiplier	float32	1
103	greenMultiplier	float32	1
104	blueMultiplier	float32	1
601	colorOffset	color	0x00000000

EnvMapMethod (methodID = 403)

ID	Name	Type	Default
1	cubeTexture	*BlockAddr*	DefaultCubeTexture
101	alpha	float32	1
2	mask	BlockAddr	Default-Texture

LightMapMethod (methodID = 404)

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

ProjectiveTextureMethod (methodID = 405)

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

RimLightMethod (methodID = 406)

ID	Name	Type	Description
601	color	*color*	0xffffff
101	Strength	float32	0.4
102	power	float32	2

AlphaMaskMethod (methodID = 407)

ID	Name	Type	Description
701	UseSecondaryUV	*bool*	false
1	texture	BlockAddr	Default_texture

RefractionEnvMapMethod (methodID = 408)

ID	Name	Type	Description
1	envMap(CubeTexture)	BlockAddr	Default_Cube_texture
101	RefractionIndex	float32	0.1
102	Dispersion R	float32	0.01
103	Dispersion G	float32	0.01
104	Dispersion B	float32	0.01
105	Alpha	float32	1

OutlineMethod (methodID = 409)

ID	Name	Type	Description
601	OutlineColor	color	0x00000000
101	OutlineSize	float32	1
701	ShowInnerLines	bool	true
702	DedicatedMesh	bool	false

FresnelEnvMapMethod (methodID = 410)

ID	Name	Type	Description
1	envMap	BlockAddr	Default_Cube_Texture
101	alpha	float32	1

FogMethod (methodID = 411)

ID	Name	Type	Description
101	Min-Distance	float32	0
102	Max-Distance	float32	1000
601	Color	color	0x808080

ShadowMapMethod-Block (ID 92)

A ShadowMap is a Block that contains a ShadowMapMethod, and the AWD-ID (for the light it should be applied to).In the AWD-Block-Structure it must appear after the associated Light-Block, and before any Material-Block that is using it.

Offset	Size	Type	Description
0	11	*BlockHeader*	As defined in Table 6.
11	Variable	*VarString*	Look-up name.
Variable	32	*Block-Address*	LightID
Variable	Variable	*NumAttrList*	Method
Variable	Variable	*UserAttrList*	User attributes.

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

ShadowMapMethods

Type ID	ShadowMapMethod Type	Category
1001	CascadeShadowMapMethod	Composite-Method
1002	NearShadowMapMethod	Composite-Method
1101	FilteredShadowMapMethod	Methods
1102	DitheredShadowMapMethod	Methods
1103	SoftShadowMapMethod	Methods
1104	HardShadowMapMethod	Methods

CascadeShadowMapMethod (methodID = 1001)

ID	Name	Type	Default
1	baseMethod	*Method*	ShadowMapMethodBase (?)

NearShadowMapMethod (methodID = 1002)

ID	Name	Type	Default
1	baseMethod	*Method*	ShadowMapMethodBase (?)

FilteredShadowMapMethod (methodID = 1101)

ID	Name	Type	Default
101	Alpha	*float32*	1
102	Epsilon	*float32*	0.002

DitheredShadowMapMethod (methodID = 1102)

ID	Name	Type	Default
101	Alpha	*float32*	1
102	Epsilon	*float32*	0.002
201	Samples	*uint32*	5
103	Range	*float32*	1

SoftShadowMapMethod (methodID = 1103)

ID	Name	Type	Default
101	Alpha	*float32*	1
102	Epsilon	*float32*	0.002
201	Samples	*uint32*	5
103	Range	*float32*	1

HardShadowMapMethod (methodID = 1104)

ID	Name	Type	Default
101	Alpha	*float32*	1
102	Epsilon	*float32*	0.002

Animation blocks

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	11	*BlockHeader*	As defined in Table 6.
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.

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.

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	11	*BlockHeader*	As defined in Table 6.
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.

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 has transformation
1	Variable	Matrix4x3	Transformation for joint (if any)

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

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	11	*BlockHeader*	As defined in Table 6.
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 frame structures (see below.)
Variable	Variable	*UserAttrList*	Animation user attributes.

Table 34: Structure of SkeletonAnimation blocks.

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.

MeshPose (ID 111) / MeshPoseAnimation (ID 112)

The MeshPose block defines the Vertex- Animation state of a Geometry .

The MeshPoseanimationBlock is similar to the MeshPose-Block. The only difference is, that, it contains a addional list of Animation-States and a additional UINT16 (Number Of Frames/lenght of the addional list +1).

Offset	Size	Type	Description
0	Variable	VarString	Look-up name
Variable	4	BlockAddr	targetGeometryID
Variable	2	uint16	Number of Frames
Variable	2	uint16	Number of Subgeometries
	2	uint16	number of streams
	2*number of streams	uint16	types of streams
Variable	Variable	*NumAttrList*	Mesh-Animation properties
Variable	Variable	*AnimationState*	Mesh-Animation-State
Variable	Variable	*AnimationStateList*	List of addional Mesh-Animation-States
Variable	Variable	*UserAttrList*	Animation user attributes.

Table 34: Structure of MeshPose / MeshPoseAnimation Blocks.

Mesh Animation Properties

All Mesh-Animation-properties are optional, so we have a minimal fileSize for the MeshAnimations with default settings. (you might want to overwrite the settings by code later anyway).

ID	Name	Type	Default
1	Loop	*bool*	true
2	StitchFinalFrame	*bool*	false
3	useTranlation	*bool*	false
4	absolutePositions	*bool*	true
5	parsingStyle	uint16	Default = 0 0: all verticles saved 1: use vertOffset 2: vector3D + uint32
6	vertOffset	List of uint32	0 (can be used if the encoder orders verticles by unMorphed/Morphed) must be set for each subgeometry
7	parsingStyleFrame	bool	Default = false. If true, the parsingStyle-property is expected to be written in every frame

Mesh Animation State

The Length of Stream and the List of Vertex-position are repeated for each SubGeometry.

Offset	Size	Type	Description
0	2	uint16	Frame Duration
2 (optional)	2	uint16	ParsingStyle This is only set when “parsingStyleFrame” is true !
variable	4	uint32	Length of all Streams together
Variable	Variable	*variable*	All Streams

ID	Stream type
1	Vertex positions
2	Original-Indicies (see parsingStyle 2)
3	UV coordinates (not used yet)
4	Vertex normals (not used yet)

Streamdata

The Number of Streams and theire Types are defined by the MeshPoseAnimationBlock, because they should be the same for each frame.

Offset	Size	Type	Description
0	2	uint32	Length of Stream
4	*variable*	*variable*	StreamData

Parsing Style

The ParsingStyle tells the Parser how the StreamData of the Streams have to be interpreted.

By setting the MeshPoseBlock-property “parsingStyleFrame” to True, the parsingStyle is not expected to be the same for all frames. If “parsingStyleFrame” is False, the parsingStyle will not be written in the MeshAnimationState (the frames).

Style	Description
0 (default)	The Data of the Stream is expected to be present for each Vert of the SubGeometry
1	The Encoder has ordered the VertBuffer by nonAnimated and animated, and only the data for the animated verts are stored. To make this work, the MeshBlock-Property “vertOffset” must be set (for each SubGeometry). This can only be used if “parsingStyleFrame” is false!
2	Only the data for the animated Verts is stored. To make this happen, a addional Stream has to be saved (and parsed as first stream) containing the original-list-Indicies of the animated verts.

The Encoder have to calculate the parsing-style to use, so we end up with the lowest file-size possible.

AnimationSet (ID 113)

A AnimationSet-Blocks is used to group multiple Animation-Blocks into one AnimationsSet.

It can contain references to SkeletonAnimationBlocks and MeshPoseAnimationBlocks.

If it contains Blocks of both Types, the Parser will create 2 Animationblocks.

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 *BlockAddr*	List of MeshPoseIDs
Variable	Variable	*UserAttrList*	Animation user attributes

UVAnimation (ID 121)

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 attributes

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

Miscellaneous blocks

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).

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 null 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.

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	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.

Part III: Using AWD

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 GitHub (https://github.com/awayTools) for the latest information about available AWD tooling.

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.

The libawd library for C++

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

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.)

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
AwayBuilder	Yes	Yes	Yes
Maxon Cinema4D	Yes	Yes	No

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.)

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.

AWD data type limitations

Feature	Limit	Reason
Length of VarStrings	65535 single-byte characters, less multi-byte characters.	16-bit length field.
Number of numeric attributes (per list)	65535	16-bit ID field
Number of text attributes (per list)	>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.

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.

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.