U3D Supported Elements

Adobe® Acrobat® SDK

April 2007

Version 8.1

Adobe® Acrobat® SDK 8.1 U3D Supported Elements for Microsoft® Windows® and Mac OS®

Edition 2.0, April 2007

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Contents

Preface

Adobe® Acrobat® 7.0 and later provide support for U3D content that can be contained in PDF 3D annotations. This guide describes the elements in the specification *Universal 3D File Format* that are fully or partially supported by Acrobat.

What's in this guide

This guide describes the subset of the *Universal 3D File Format* that is supported by Acrobat 7.0 and later. It describes the subset that Acrobat reads.

The basis for this guide is the *Universal 3D File Format* 1st Edition (December 2004), updated for the *Universal 3D File Format* 3rd Edition (June 2006).

This guide provides the following information:

- Components of the *Universal 3D File Format* that are supported by Acrobat. For the purpose of this guide, a *component* is the content of a distinct U3D block.
- Subsets of supported components that are unsupported or have limited support.
- Best practices in the Acrobat implementation, and guidance for situations where an aspect of the specification *Universal 3D File Format* is ambiguous.

This guide reflects the section numbers and headings from the *Universal 3D File Format* guide. Ordering may diverge where more general descriptive notes applying to multiple components are present. Because of the changing nature of the *Universal 3D File Format*, this guide provides only an overview of Acrobat support for the format. The nature of this support and the number of supported components is subject to change. The structure of the guide is in terms of the file blocks encountered and sub parts that are supported.

Who should read this guide

This guide is for developers who want to create U3D content for use in PDF 3D annotations.

Related documentation

The resources in this table can help you learn about the Acrobat SDK and technologies related to the development of 3D annotations.

For information about	See
A guide to the documentation in the Acrobat SDK.	*Acrobat SDK Documentation Roadmap*
Developing plug-ins for Acrobat and Adobe Reader®, as well as for PDF Library applications.	*Developing Plug-ins and Applications*

For information about	See
Detailed descriptions of the APIs for Acrobat and Adobe Reader plug-ins, as well as for PDF Library applications.	*Acrobat and PDF Library API Reference*
Detailed descriptions of JavaScript™ APIs for adding interactivity to 3D annotations within PDF documents.	*JavaScript for Acrobat 3D Annotations API Reference*
A detailed description of the U3D file format standard.	*ECMA-363: Universal 3D File Format, Edition 1 (December 2004) and ECMA 363: Universal 3D File Format, Edition 3 (June 2006)*. These specifications are available at the ECMA web site www.ecma-international.org/.

1 U3D Elements Supported by Acrobat

This guide describes the elements in the specification *Universal 3D File Format* that are fully or partially supported by Acrobat. It focuses on what Acrobat can consume. This guide omits U3D elements that Acrobat does not support.

Conventions used in this guide

The section numbers and headings used in this guide correspond to the section numbers and headings in the *Universal 3D File Format*, 3rd Edition. These section numbers and headings are italicized to distinguish them from section headings of this guide.

The main headings in this section specify the element numbers corresponding to each block type that is supported by Acrobat 7.0 and later. (By inference, elements of the specification not mentioned may be unsupported.) This guide addresses supported U3D elements as follows:

- Block types with no subparts are supported in their entirety. Omitted block types are not supported.
- Subparts are listed, information is given about the nature of Acrobat support for them.

The following terms appear in the description of some elements:

- "Parsed and unused": Acrobat requires the particular element to be present in the U3D data, even though Acrobat does not use the element. If the element is not present, an error occurs.
- "Not parsed": The element is not required and Acrobat ignores it if present.

U3D section numbers and headings that appear in the body of this guide are identified with italics.

U3D elements

9.2.6 Meta Data

Meta data is parsed and stored.

Meta data for the file header can contain an entry with a meaning specific to Acrobat, such as the data described in "U3D Meta Data" on page 40.

Units scaling-factor meta data

Acrobat 7.0.7 and later support units scaling-factor meta data specified in the *Universal 3D File Format*, 1st Edition. Acrobat 8.1 adds support for units scaling-factor meta data for a new convention made possible by the *Universal 3D File Format*, 3rd Edition. If neither convention appears, the objects are unitless.

Units scaling factor specified using 9.4.1.6 "F64: Units Scaling Factor"

Beginning with Acrobat 8.1, if U3D units scaling factor (units) data described by section *9.4.1.2, U32: Profile Identifier* sets the Defined Units bit (0x00000008), Acrobat obtains units data from U3D data described by

section *9.4.1.6 F64 Units Scaling Factor* and ignores any units data provided in the `RHAdobeUnitsMeters` key. The Defined Units bit indicates the objects are defined with a units scaling factor.

Units scaling factor in the RHAdobeUnitsMeters key

Beginning with Acrobat 7.0.7, valid units scaling factor (units) data requires that the first meta data node entry of the 9.4.1 File Header (blocktype: 0x00443355) contain the following fields, in sequence:

1. *9.2.6.2 U32: Key/Value Pair Attributes*, with a value indicating string content.
2. *9.2.6.3 String: Key String*, containing the string:

   `RHAdobeUnitsMeters=`
3. *9.2.6.4 String: Value String*, containing an alphanumeric representation of a floating point number. This representation consists of a sequence of ASCII numeric characters (0 through 9) representing the whole number portion, optionally followed by a period character (.) and an additional sequence of ASCII numeric characters representing the fractional portion of the floating point value.

Assuming these conditions are met, the string content is interpreted as a single-precision floating point value called the Units Scaling Factor. Multiplying the Units Scaling Factor by the values of positions and lengths in the file converts the units for those values to meters. For example, if the units in the file are intended to represent centimeters, the Units Scaling Factor would be 0.01. A value of 5 in the file would mean 5 cm (0.05m).

If the U3D file contains data that corresponds to the *9.4.1.6 F64: Units Scaling Factor* ("Units scaling factor specified using 9.4.1.6 "F64: Units Scaling Factor"" on page 7), Acrobat 8.1 ignores any units scaling data provided in the sequence of data that includes the string `RHAdobeUnitsMeters=`.

9.4.1 File Header (blocktype: 0x00443355)

9.4.1.1 I32: Version

This section appears only in *Universal 3D File Format*, 1st Edition.

Version 0 is supported. All other version numbers result in a 3D parsing error.

9.4.1.1.1 I16: Major Version

Acrobat 7 and later support major version 0. When Acrobat reads U3D content that specifies major version numbers greater than 0, it presents a dialog box stating that this version is unsupported, and that the user needs to update their version of Acrobat. All negative major version numbers result in a 3D parsing error.

This change relates to the *Universal 3D File Format*, 3rd Edition support. This field supersedes the definition *9.4.1.1 I32: Version* present in the *Universal 3D File Format*, 1st Edition.

9.4.1.1.2 I16: Minor Version

Beginning with Acrobat 8.1, all minor versions are supported.

This change relates to the *Universal 3D File Format*, 3rd Edition support. This field supersedes the definition *9.4.1.1 I32: Version* present in the *Universal 3D File Format*, 1st Edition.

9.4.1.2 U32: Profile Identifier

(Acrobat 8.0 and later) Parsed.

Bit position	Description
0x00000002	Extensible profile, which is a feature introduced in *Universal 3D File Format*, 3rd Edition. Acrobat 8.1 supports this feature.
0x00000004	Specifies "No compression mode". It is recommended that files be compressed with compression mode enabled (0x00000004 bit is 0x00000000). Acrobat responds to this bit being set depending on the version: • Acrobat versions 7.0.0 to 7.0.7 may crash. • Acrobat 7.0.9 terminates parsing. • Acrobat 8.0 and later, parse noncompressed data.
0x00000008	Specifies "Defined units", which is a feature introduced in *Universal 3D File Format*, 3rd Edition. Acrobat 8.1 supports this feature, as described "Units scaling factor specified using 9.4.1.6 "F64: Units Scaling Factor"" on page 7.

9.4.1.5 U32: Character Encoding

Character encoding is UTF-8, as specified in the *Universal 3D File Format*.

9.4.1.6 F64: Units Scaling Factor

Parsed and used only by Acrobat 8.1 and later. This change relates to the *Universal 3D File Format*, 3rd Edition support.

9.4.3 Modifier Chain (blocktype: 0xFFFFFF14)

9.4.3.2 U32: Modifier Chain Type

Parsed and unused.

9.4.3.4 Modifier Chain Bounding Sphere

Parsed and unused.

9.4.3.5 Modifier Chain Axis-Aligned Bounding Box

Parsed and unused.

Important note regarding U3D modifier declarations

Certain blocks may only be read if presented as declarations for object modifiers. As a result, such blocks must reside within a modifier chain. The following sections specify such object modifiers:

- *9.5.1 Group Node (blocktype: 0xFFFFFF21)*
- *9.5.2 Model Node (blocktype: 0xFFFFFF22)*
- *9.5.3 Light Node (blocktype: 0xFFFFFF23)*
- *9.5.4 View Node (blocktype: 0xFFFFFF24)*
- *9.6.1.1 CLOD Mesh Declaration (blocktype: 0xFFFFFF31)*
- *9.6.2.1 Point Set Declaration (blocktype: 0xFFFFFF36)*
- *9.6.2.2 Point Set Continuation (blocktype: 0xFFFFFF3E)*
- *9.6.3.1 Line Set Declaration (blocktype: 0xFFFFFF37)*
- *9.6.3.2 Line Set Continuation (blocktype: 0xFFFFFF3F)*
- *9.7.2 Subdivision Modifier (blocktype: 0xFFFFFF42)*
- *9.7.3 Animation Modifier (blocktype: 0xFFFFFF43)*
- *9.7.5 Shading Modifier (blocktype: 0xFFFFFF45)*
- *9.8.5.1 Texture Declaration (blocktype: 0xFFFFFF55)*

9.4.5 New Object Type (blocktype: 0xFFFFFF16)

Acrobat 8.1 supports this feature, which is supported by *Universal 3D File Format*, 3rd Edition.

Acrobat 8.1 supports only one extension block type, RHAdobeMeshResource. See "Right Hemisphere Adobe 3D Mesh Compression" on page 20.

9.4.5.1 String: New Object Type Name

Parsed.

9.4.5.2 U32: Modifier Type

Parsed and unused.

9.4.5.3 Extension ID

Parsed.

9.4.5.4 U32: New Declaration Block Type

Parsed. For information on how Acrobat interprets this feature, see "9.4.6 New Object Block (blocktype: 0x00000100 to 0x00FFFFFF)" on page 12.

9.4.5.5 U32: Continuation Block Type Count

Parsed.

9.4.5.6 U32: New Continuation Block Type

Parsed and unused.

9.4.5.7 String: Extension Vendor Name

Parsed.

9.4.5.8 U32: Extension URL Count

Parsed.

9.4.5.9 String: Extension Information URL

Parsed and unused.

9.4.5.10 String: Extension Information String

Parsed.

9.4.6 New Object Block (blocktype: 0x00000100 to 0x00FFFFFF)

This change relates to *Universal 3D File Format*, 3rd Edition support. This set of blocks is supported as of Acrobat 8.1.

Acrobat supports only the extension block type RHAdobeMeshResource, as described in "Right Hemisphere Adobe 3D Mesh Compression" on page 20.

9.2.1 U32: Block Type

Parsed. To declare a block of type RHAdobeMeshResource, the value should match that of the field *9.4.5.4 U32: New Declaration Block Type* within a valid *9.4.5 New Object Type* declaration of the new block type described in "9.4.5 New Object Type (blocktype: 0xFFFFFF16)" on page 10.

All other block types will be ignored.

9.5.X.2 Parent Node Data

Note: The presence of "X" in an element number indicates a numeric sequence. For example, 9.5.X refers to numbers of the form 9.5.1, 9.5.2, and so forth.

9.5.X.2.1 U32: Parent Node Count

In Acrobat versions 7.0 through 8.1, a value greater than 1 results in data duplication, so heavy usage of this feature may result in reduced application performance.

Acrobat versions 7.0 through 7.0.7 do not support multi-level multi-parenting (that is, nodes having multiple parents, who in turn have multiple parents). Acrobat 8.0 and later support multi-level multi-parenting.

9.5.1 Group Node (blocktype: 0xFFFFFF21)

9.5.2 Model Node (blocktype: 0xFFFFFF22)

9.5.2.4 U32: Model Visibility

Acrobat 7.0 through 7.0.9 may have performance problems with models that specify value front and back face (3). Acrobat 8.0 and later do not have such problems.

9.5.2.3 String: Model Resource Name

Each model node must reference a unique resource. If model nodes share the same resource, the first model node will receive it, and subsequent model nodes will be empty.

9.5.3 Light Node (blocktype: 0xFFFFFF23)

Supported.

9.5.4 View Node (blocktype: 0xFFFFFF24)

Supported with the exceptions noted.

9.5.4.3 String: View Resource Name

Parsed and unused.

9.5.4.6.2 F32: View Orthographic Height

Parsed and unused.

9.5.4.6.3.1 F32: View Projection Vector X

Parsed and unused.

9.5.4.6.3.2 F32: View Projection Vector Y

Parsed and unused.

9.5.4.6.3.3 F32: View Projection Vector Z

Used to scale field of view.

9.5.4.8 U32: Backdrop Count

Not parsed.

9.5.4.9 Backdrop Properties

Not parsed.

9.5.4.10 U32: Overlay Count

Not parsed.

9.5.4.11 Overlay Properties

Not parsed.

9.6.1.1 CLOD Mesh Declaration (blocktype: 0xFFFFFF31)

Should be part of an object; that is, be preceded by a 9.5.2 Model Node (blocktype: 0xFFFFFF22).

9.6.1.1.6 Skeleton Description

Parsed and unused (that is, no skeletal animation, skinning, or bones).

9.6.1.2 CLOD Base Mesh Continuation (blocktype: 0xFFFFFF3B)

9.6.1.3 CLOD Progressive Mesh Continuation (blocktype: 0xFFFFFF3C)

9.6.2.1 Point Set Declaration (blocktype: 0xFFFFFF3B)

Support in Acrobat 7.0.7 and later. Acrobat 7.0.7 through 8.0 display the point set only when the selected render mode is 'Vertices' and fail to display the point set in other render modes. Acrobat 8.1, displays the point set in all render modes.

The data should be part of an object; that is, the data should be preceded by a block as described in *9.5.2 Model Node (blocktype: 0xFFFFFF22)*.

9.6.2.1.3.2 U32: Point Count

Parsed and unused.

9.6.2.1.3.9 Shading Description

Parsed and unused. Material attributes cannot be applied to lines. Use per vertex attributes within 9.6.3.2.4.6.

9.6.2.1.4.1 Quality Factors

Parsed and unused.

9.6.2.1.5 Skeleton Description

Parsed and unused (that is, no skeletal animation, skinning, bones).

9.6.2.2 Point Set Continuation (blocktype: 0xFFFFFF3E)

Support in Acrobat 7.0.7 and later. Acrobat 7.0.7 through 8.0 display the point set only when the selected render mode is 'Vertices' and fail to display the point set in other render modes. Acrobat 8.1 displays the point set in all render modes.

The data should be part of an object; that is, the data should be preceded by a block as described in *9.5.2 Model Node (blocktype: 0xFFFFFF22)*.

9.6.2.2.2 U32: Chain Index

Parsed and unused; specified to always be 0.

9.6.2.2.4.4 New Normal Info

Parsed and unused.

9.6.2.2.4.6.3 U32 [cNormlIdx]: Normal Local Index

Parsed and unused.

9.6.2.2.4.6.4 New Line Diffuse Color Coords

Parsed but it is possible that only the first color encountered will be rendered correctly.

9.6.2.2.4.6.5 New Line Specular Color Coords

Parsed and unused.

9.6.2.2.4.6.6 New Line Texture Coords

Parsed and unused.

9.6.3.1 Line Set Declaration (blocktype: 0xFFFFFF37)

Should be part of an object; that is, be preceded by a 9.5.2 Model Node (blocktype: 0xFFFFFF22).

9.6.3.1.3.9 Shading Description

Parsed and unused. Material attributes cannot be applied to lines. Use per vertex attributes within 9.6.3.2.4.6.

9.6.3.1.5 Skeleton Description

Parsed and unused (that is, no skeletal animation, skinning, bones).

9.6.3.2 Line Set Continuation (blocktype: 0xFFFFFF3F)

9.6.3.2.2 U32: Chain Index

Parsed and unused; specified to always be 0.

9.6.3.2.4.4 New Normal Info

Parsed and unused.

9.6.3.2.4.6.3 U32 [cNormlIdx]: Normal Local Index

Parsed and unused.

9.6.3.2.4.6.5 New Line Specular Color Coords

Parsed and unused.

9.6.3.2.4.6.6 New Line Texture Coords

Parsed and unused.

9.7.2 Subdivision Modifier (blocktype: 0xFFFFFF42)

9.7.2.2 U32: Chain Index

Parsed and unused.

9.7.2.3 U32: Subdivision Attributes

Parsed and unused.

9.7.2.5 F32: Subdivision Tension

Parsed and unused.

9.7.2.6 F32: Subdivision Error

Parsed and unused.

9.7.3 Animation Modifier (blocktype: 0xFFFFFF43)

9.7.3.2 U32: Chain Index

Parsed and unused.

9.7.3.3 U32: Animation Modifier Attributes

Parsed and unused.

9.7.3.4 F32: Time Scale

Parsed and unused.

9.7.3.6 Motion Information

Only the first Motion Information entry is supported.

9.7.3.6.2 U32: Motion Attributes

Parsed and unused.

9.7.3.6.3 F32: Time Offset

Parsed and unused.

9.7.3.6.4 F32: Time Scale

Parsed and unused.

9.7.3.7 F32: Blend Time

Parsed and unused.

9.7.5 Shading Modifier (blocktype: 0xFFFFFF45)

Requires a mesh declared within a *9.5.2 Model Node (blocktype: 0xFFFFFF22)* prior to this chunk in file order.

9.7.5.2 U32: Chain Index

Parsed and unused.

9.7.5.3 U32: Shading Attributes

Parsed and unused.

9.8.1 Light Resource (blocktype: 0xFFFFFF51)

9.8.1.2 U32: Light Attributes

Parsed and unused.

9.8.1.5.1 F32: Light Attenuation Constant Factor

Acrobat 7.0.7 and later parse but do not support this factor. As a result, point and spotlights make no illumination contribution to the scene.

9.8.1.5.2 F32: Light Attenuation Linear Factor

Acrobat 7.0.7 and later parse but do not support this factor. As a result, point and spotlights make no illumination contribution to the scene.

9.8.1.5.3 F32: Light Attenuation Quadratic Factor

Acrobat 7.0.7 and later parse but do not support this factor. As a result, point and spotlights make no illumination contribution to the scene.

9.8.1.6 F32: Light Spot Angle

Parsed. Angle treated as being in degrees, and for a full angle, cone edge to cone edge. This is not specified in the *Universal 3D File Format*.

9.8.3 Lit Texture Shader (blocktype: 0xFFFFFF53)

9.8.3.2 U32: Lit Texture Shader Attributes

Parsed and unused.

9.8.3.3 F32: Alpha Test Reference

Parsed and unused.

9.8.3.4 U32: Alpha Test Function

Parsed and unused.

9.8.3.5 U32: Color Blend Function

Parsed and unused.

9.8.3.6 U32: Render Pass Enabled Flags

Parsed and unused.

9.8.3.8 U32: Alpha Texture Channels

Parsed and unused.

9.8.3.10 Texture Information

Only high-end cards may support more than one Texture Information layer. Examples of high-end cards include NVIDIA GeForce3 and GeForce4, but not NVIDIA GeForce4 MX. Software supports one Texture Information layer. Other layers are parsed but not rendered.

9.8.3.10.4 U8: Blend Source

Parsed and unused.

9.8.3.10.6 U8: Texture Mode

TM_NONE and TM_REFLECTION are supported.

TM_NONE is utilized by diffuse rendered textures.

TM_REFLECTION is not supported by the software renderer, and may not be supported by older or built -in graphics cards, such as older versions of the Intel Graphics Media Accelerator (GMA) and NVIDIA GeForce2.

The renderer supports one layer of each type. Rendering of multiple layers of the same type is undefined.

9.8.3.10.7 F32: Texture Transform Matrix Element

Texture scaling and rotation is not fully supported.

9.8.3.10.8 F32: Texture Wrap Transform Matrix Element

Not parsed.

9.8.4 Material Resource (blocktype: 0xFFFFFF54)

9.8.4.2 U32: Material Attributes

Parsed and unused.

9.8.4.4 Diffuse Color Parsed.

The value may modulate the per-vertex values of a mesh.

9.8.4.5 Specular Color

Parsed, but not supported by software renderer, and may not be supported by older or built-in graphics cards, such as older versions of the Intel Graphics Media Accelerator (GMA) and NVIDIA GeForce2.

9.8.4.6 Emissive Color

Parsed, but not supported by software renderer, and may not be supported by older or built-in graphics cards, such as older versions of the Intel Graphics Media Accelerator (GMA) and NVIDIA GeForce2. Emissive color currently does not modulate textures; it is added to the final surface color.

9.8.5.1 Texture Declaration (blocktype: 0xFFFFFF55)

9.8.5.1.2.3 U8: Texture Image Type

Parsed and unused.

9.8.5.1.4.1 U8: Compression Type

JPEG-24, PNG, and JPEG-8 are supported. TIFF is unsupported.

9.8.5.1.4.2 U8: Texture Image Channels

Parsed and unused.

9.8.5.1.4.4 U32: Image Data Byte Count

Parsed and unused.

9.8.5.1.4.6 String: Image URL

Parsed and unused.

9.8.5.2 Texture Continuation (blocktype: 0xFFFFFF5C)

Uses information gleaned from 9.8.5.1.

One continuation block is supported per texture.

9.8.6 Motion Resource (blocktype: 0xFFFFFF56)

2 Right Hemisphere Adobe 3D Mesh Compression

This chapter specifies the 3D mesh compression format developed by Right Hemisphere and supported by Acrobat 8.1. Right Hemisphere Adobe 3D mesh compression introduces a new block within the framework of the extension mechanism of the ECMA-363, *Universal 3D File Format*, *3rd Edition* (U3D). This block significantly improves the mesh compression performance of the format.

Beginning with Acrobat 8.1, Acrobat supports this format. Acrobat continues to support original mesh and other block types, as described in "U3D Elements Supported by Acrobat" on page 7. The Right Hemisphere Adobe 3D mesh compression format is an alternative choice for creating smaller file size representations of meshes, being able to store equivalent sets of data to that of the existing U3D uncompressed mesh blocks. The extension mechanism is specified in *Universal 3D File Format*, *3rd Edition*, which is available at www.ecma-international.org.

The compressed mesh chunk is written to a U3D file using the chunk extension mechanism of the U3D format (BlockType_FileNewObjectTypeU3D = 0xFFFFFF16). See *Universal 3D File Format*, *3rd Edition*, section *9.4.5 New Object Type (blocktype: 0xFFFFFF16)*.

RHAdobeMeshResource type declaration

The following table specifies (in sequential order) the contents of the *9.4.5 New Object Type (blocktype: 0xFFFFFF16)* that declares a type of RHAdobeMeshResource.

ECMA-363 Section #	ECMA-363 Section title	Whether parsed and required value
9.4.5.1	String: New Object Type Name	Parsed. Must have this value: `RHAdobeMeshResource`
9.4.5.2	U32: Modifier Type	Parsed and ignored.

9.4.5.3	Extension ID	Parsed. Must have these values:
		`0x96a804a6, 0x3fb9, 0x43c5`
		`0xb2, 0xdf, 0x2a, 0x31, 0xb5, 0x56,`
		`0x93, 0x40`

9.4.5.4	U32: New Declaration Block Type	Parsed and required. A unique identifier assigned by the originator of the RHAdobeMeshResource data. The value must appear in all RHAdobeMeshResource chunks and must not match any predefined unique identifiers, such as 0x5a4f173e or 0x11223344 or 0xaabbccdd.
9.4.5.5	U32: Continuation Block Type count	Parsed and ignored
9.4.5.6	U32: New Continuation Block Type	Parsed and ignored This data is an array of values, whose size is defined by *9.4.5.5 ContinuationBlockType count*.

ECMA-363 Section #	ECMA-363 Section title	Whether parsed and required value
9.4.5.7	String: Extension Vendor Name	Parsed. Must have this value: `Right Hemisphere Adobe Systems`
9.4.5.8	U32: Extension URL Count	Parsed and ignored
9.4.5.9	String: Extension Information URL	Parsed and ignored This data is an array of values, whose size is defined by *9.4.5.8 U32: Extension URL Count*.
9.4.5.10	String: Extension Information String	Parsed. Must have this value: `version 1.0` **Note:** If any other value is provided for this entry, processing is terminated.

Compressed chunk format for the RHAdobeMeshResource type

This section specifies the chunks that comprise the new declaration block that contains the compressed mesh data for the RHAdobeMeshResource type.

Each chunk consists of the following parts, which must appear in the order listed:

- Character Encoding
- U8 ChunkFlags
- Array of Materials declarations
- Vertex positions
- Normals (only when bExcludeNormals flag is Off)
- Diffuse colors
- Specular colors
- Texture coordinates
- Face Material IDs
- Position face indices
- Normal face indices (only when the bExcludeNormals flag is Off)
- Diffuse color face indices
- Specular color face indices
- Texture coordinates face indices
- Skeleton description (optional)

For definitions of type information terms used in this document, refer to section 5.2 of the *Universal 3D File Format, 3rd Edition*.

Chunk header

The chunk header provides information about the mesh data contained in the RHAdobeMeshResource type.

Character Encoding

Two bytes for string length, followed by a character string that specifies character encoding. For information on character encoding, see *Universal 3D File Format* section *9.4.1.5 U32: Character Encoding*. Acrobat 8.1 supports UTF-8.

U8 ChunkFlags

Bit position	Meaning
0,1,2,3	Chunk version (0-15). Must be 0.
4	Specifies if there is subchunk extension data present (see "SubChunk extension data (optional)" on page 22). In Acrobat 8.1, this flag must be 0.
5	Indicates the presence of a skeleton description
6,7	MaterialsCount encoding type (0,1,2 or 3). • Type 0, means that MaterialsCount is 1. • Type 1, means that MaterialsCount is the next U8 value in the stream. • Type 2, means that MaterialsCount is the next U16 value in the stream. • Type 3, means that MaterialsCount is the next U32 value in the stream. For information on Types 1, 2 or 3, see "U8|U16|U32 MaterialsCount (optional)" on page 23

SubChunk extension data (optional)

Present only if bit 4 is ON in ChunkFlags described in "U8 ChunkFlags" on page 22. Provides reference information for additional subchunks that exist within Future Expansion SubChunks, described in "Future Expansion SubChunks" on page 37.

U32 SizeOfTheMeshDescriptionBlock

This value indicates the number of bytes to skip to reach the start of Future Expansion SubChunks, described in "Future Expansion SubChunks" on page 37. This number is relative to the start of the main mesh chunk as the starting point ("Main mesh chunk" on page 23).

If present, this value must be 0.

U16 NumberOfSubChunks

Number of subchunks following the main mesh definition block. It should currently be set to a value of 0 if present in the stream.

SubChunkInfo

Data pairs consisting of a chunk size and additional information. Each data pair refers to sub-chunk information. The data pairs are written the number of times described in "U16 NumberOfSubChunks" on page 22.

U32 SubChunkSize

Size of the sub-chunk. Any of the sub-chunks can be independently located and read, by doing these tasks:

1. Moving to the location in the file indicated by "U32 SizeOfTheMeshDescriptionBlock" on page 22.
2. Moving further in the file by successive amounts of SubChunkSize entries. It should be set to a value of 0 if present in the stream.

U16 SubChunkTypeInfo

Chunk type information. It should be set to a value of 0 if present in the stream.

Main mesh chunk

The main mesh chunk contains the 3D mesh data.

U8|U16|U32 MaterialsCount (optional)

The MaterialsCount value is written as U8, U16 or U32 value depending on the MaterialsCount encoding type described in "U8 ChunkFlags" on page 22 (1, 2 and 3 respectively). If the encoding type is 0, the MaterialsCount must be 1 and the Array of Materials declarations is not read from the stream.

Array of Materials declarations

Array size is MaterialsCount. Each material declaration consists of the following items:

U8 MaterialFlag

Bit position	Description
0,1	Texture coordinates dimensions for the first texture layer (or 0 in case the mesh doesn't have texture layers).
2,3	Number of bytes required to encode OriginalShadingID, as follows: 0 indicates 1 byte 1 indicates 2 bytes 2 indicates 3 bytes 3 indicates 4 bytes

Bit position	Description
4,5	Number of bytes required to encode the NumberOfTextureLayers, as follows: 0 indicates 1 byte 1 indicates 2 bytes 2 indicates 3 bytes 3 indicates 4 bytes
6	HasDiffuseColors flag
7	HasSpecularColors flag

U8|U16|U24|U32 NumberOfTextureLayers

Can be encoded as 1, 2, 3 or 4 bytes (depending on bits 4,5 described in "U8 MaterialFlag" on page 23).

U8|U16|U24|U32 OriginalShadingID

Can be encoded as 1, 2, 3 or 4 bytes (depending on bits 2,3 described in "U8 MaterialFlag" on page 23).

Array of Texture Coordinate Dimension for layers 1 ... NumberOfAdditionalTextureLayers

NumberOfAdditionalTextureLayers = NumberOfTextureLayers – 1, as the first layer data is written as Bits 0,1 of "U8 MaterialFlag" on page 23 above.

Each byte stores texture coordinate dimensions for four layers (two bits per layer).

The total number of bytes in an array is described by the following pseudo-code:

```
If (NumTextureLayer-1) mod 4 is more than 0 then
   size = (NumTextureLayers-1)/4 + 1
else
size = (NumTextureLayers-1)/4
```

Note: Texture coordinate dimension for layer 0 is already written as described in "U8 MaterialFlag" on page 23.

U8 ValuesFlags

Bit position	Description
0,1,2,3,4,5	If a bit is set, it means that the corresponding value is not 0 and should be read from the stream.
0	NumFaces should be read from the stream ("U8|U16|U24|U32 NumFaces" on page 25)
1	NumPositions should be read from the stream ("U8|U16|U24|U32 NumPositions" on page 25)
2	NumNormals should be read from the stream ("U8|U16|U24|U32 NumNormals" on page 25)
3	NumDiffuseColors should be read from the stream ("U8|U16|U24|U32 NumDiffuseColors" on page 25)

Bit position	Description
4	NumSpecularColors should be read from the stream ("U8|U16|U24|U32 NumDiffuseColors" on page 25)
5	NumTexCoords should be read from the stream ("U8|U16|U24|U32 NumTexCoords" on page 25)
6,7	Value size in bytes for items U8|U16|U24|U32 NumFaces through U8|U16|U24|U32 NumTexCoords. The bit values have these meanings: 0 indicates 1 byte 1 indicates 2 bytes 2 indicates 3 bytes 3 indicates 4 bytes

Using bits to represent discrete data is more efficient than using bytes, especially because the data can be repeated thousands of times if the content is a large list of small mesh objects.

U8|U16|U24|U32 NumFaces

Optional, should only be read if Bit 0 is set as described in "U8 ValuesFlags" on page 24.

Value size in bytes depends on bits 6,7 as described in "U8 ValuesFlags" on page 24.

U8|U16|U24|U32 NumPositions

Optional, should only be read if bit 1 is set as described in "U8 ValuesFlags" on page 24.

Value size in bytes depends on bits 6,7 as described in "U8 ValuesFlags" on page 24.

U8|U16|U24|U32 NumNormals

Optional, should only be read if bit 2 is set as described in "U8 ValuesFlags" on page 24. Value size in bytes depends on bits 6,7 as described in "U8 ValuesFlags" on page 24

U8|U16|U24|U32 NumDiffuseColors

Optional, should only be read if bit 3 is set in "U8 ValuesFlags" on page 24.

Value size in bytes depends on bits 6,7 as described in "U8 ValuesFlags" on page 24.

U8|U16|U24|U32 NumSpecularColors

Optional, should only be read if bit 4 is set as described in "U8 ValuesFlags" on page 24.

Value size in bytes depends on bits 6,7 as described in "U8 ValuesFlags" on page 24.

U8|U16|U24|U32 NumTexCoords

Optional, should only be read if bit 5 is set as described in "U8 ValuesFlags" on page 24.

Value size in bytes depends on bits 6,7 as described in "U8 ValuesFlags" on page 24.

Vertex positions

U8 DataType

The following types of floating point data are defined:

- Type 0 - Uncompressed F32 data
- Type 2 - Quantized U32 values
- Type 3 - Quantized alpha-theta encoded normals

Encoded data block for NumPositions * 3 floating point values

Encoding format and block size depend on DataType, described in "Floating point data encoding types" on page 28.

Normals (only when bExcludeNormals flag is Off)

U8 DataType

The following types of floating point data are defined:

- Type 0 - Uncompressed F32 data
- Type 2 - Quantized U32 values
- Type 3 - Quantized alpha-theta encoded normals

Encoded data block for NumNormals * 3 floating point values

Encoding format and block size depend on DataType (see "Floating point data encoding types" on page 28).

Diffuse colors

Diffuse colors are RGBA colors.

The ordinary range for the color components is 0.0 to +1.0. The value 0.0 corresponds to black and the value +1.0 corresponds to full intensity. Values outside the ordinary range are allowed.

The ordinary range for the alpha component is 0.0 to +1.0. The value 0.0 corresponds to fully transparent and the value +1.0 corresponds to fully opaque. Values outside the ordinary range are allowed.

U8 DataType

The following types of floating point data are defined:

- Type 0 - Uncompressed F32 data
- Type 2 - Quantized U32 values

Encoded data block for NumDiffuseColors * 4 floating point values

Encoding format and block size depend on DataType (see "Floating point data encoding types" on page 28).

Specular colors

Specular colors are RGBA colors.

The ordinary range for the color components is 0.0 to +1.0. The value 0.0 corresponds to black and the value +1.0 corresponds to full intensity. Values outside the ordinary range are allowed.

The ordinary range for the alpha component is 0.0 to +1.0. The value 0.0 corresponds to fully transparent and the value +1.0 corresponds to fully opaque. Values outside the ordinary range are allowed.

U8 DataType

The following types of floating point data are defined:

- Type 0 - Uncompressed F32 data
- Type 2 - Quantized U32 values

Encoded data block for NumSpecularColors * 4 floating point values

Encoding format and block size depend on DataType (see "Floating point data encoding types" on page 28).

Texture coordinates

This section specifies the format of texture coordinates data.

U8 DataType

The following types of floating point data are defined:

- Type 0 - Uncompressed F32 data
- Type 2 - Quantized U32 values

Encoded data block for NumTexCoords * 4 floating point values

Encoding format and block size depend on DataType (see "Floating point data encoding types" on page 28).

Face Material IDs

Encoded array of U32 Face Material ID values [NumFaces] .

Position face indices

Encoded array of position face indices [NumFaces*3] .

Normal face indices (only when the bExcludeNormals flag is Off)

Encoded array of normal face indices [NumFaces*3] .

Diffuse color face indices

Encoded array of diffuse color face indices. Only for faces which are assigned a material which has m_uDiffuseColors flag set.

Specular color face indices

Encoded array of specular color face indices. Only for faces which are assigned a material which has m_uSpecularColors flag set.

Texture coordinates face indices

Encoded array of texture coordinates face indices. The number of texture coordinate index triplets per face is determined by m_uNumTextureLayers member variable.

Skeleton description (optional)

Skeleton description is only present if bit 5 is set in ChunkFlags, as described in "U8 ChunkFlags" on page 22.

The encoding format conforms to U3D skeleton description, see section *9.6.1.1.6 Skeleton Description* of the *Universal 3D File Format, Edition*.

Data encoding types

This section specifies encoding for different data types.

Floating point data encoding types

Right Hemisphere Adobe compressed 3D mesh defines multiple encodings for floating point data.

Type 0 (FPDATATYPE_F32)

```
F32 x[size];
F32 y[size];
F32 z[size];
```

Note: It is written in XXX:YYY:ZZZ: format, not in XYZXYZXYZ. Storing data in this manner gives better compression with ZIP.

Source data is not altered in any way (lossless).

Example: Vertices (0.0, 1.0, 2.0), (3.0, 4.0, 5.0), (6.0, 7.0, 8.0) will be encoded as 0.0, 3.0, 6.0, 1.0, 4.0, 7.0, 2.0, 5.0, 8.0.

Type 2 (FPDATATYPE_QUANTIZEDCOMPRESSED_FLOATS)

```
F32 min;
F32 max;
U32 NumQuants; // Optional if min != max.
```

```
U32 q[NumQuants-1] //Encoded array of quantized U32 values (optional, only
if min != max).
```

For details on encoding U32 values, see "Integer data encoding" on page 30.

Each value is converted into F32 value with this formula:

```
F32Val = min + U32Val * (max-min) / NumQuants;
```

Note: If min equals to max, then no data array is written, because it means that all values are equal (for example, this happens if a mesh object is a plane and all Z values are equal).

In this example, consider the set of vertices (10.0, 20.0, 0.0), (50.0, 20.0, 0.0), (60.0, 28.0, 0.0), (80.0, 30.0, 0.0), (25, 21.5, 0.0). These vertices are encoded as shown below.

For the X values: 10.0, 50.0, 60.0, 80.0, 25.0:

10.0 (F32 min)

80.0 (F32 max)

1000 (U32 NumQuants)

0, 571, 714, 1000, 214 (A set of quantized U32 values, which is later going to be UINT-encoded)

Each source F32 value is reconstructed from U32 quantized data like this:

10.0 + 0*(80.0 – 10.0)/1000 = 10.0

10.0 + 571*(80.0 – 10.0)/1000 = 49.97

10.0 + 714*(80.0 – 10.0)/1000 = 59.98

10.0 + 1000*(80.0 – 10.0)/1000 = 80.0

10.0 + 214*(80.0 – 10.0)/1000 = 24.98

For the Y values: 20.0, 20.0, 28.0, 30.0, 21.5, encoding is this:

20.0 (F32 min)

30.0 (F32 max)

10 (U32 NumQuants)

0, 0, 8, 10, 2 (A set of quantized U32 values, which is later going to be UINT-encoded)

Each source F32 value can be reconstructed from U32 quantized data like this:

20.0 + 0*(30.0 – 20.0)/10 = 20.0

20.0 + 0*(30.0 – 20.0)/10 = 20.0

20.0 + 8*(30.0 – 20.0)/10 = 28.0

20.0 + 10*(30.0 – 20.0)/10 = 30.0

20.0 + 2*(30.0 – 20.0)/10 = 22.0

For the Z values: 0.0, 0.0, 0.0, 0.0, 0.0, encoding is this:

0.0 (F32 min)

0.0 (F32 max)

Notice that NumQuants and quantized U32 array are omitted because min==max. Which means that all values in the array are equal ["min"].

Type 3 (FPDATATYPE_QUANTIZEDCOMPRESSED_NORMALS)

This section addresses only normals data.

```
U32 NumQuants;
U32 qAlphas[NumQuants-1]; // Encoded array of quantized U32 Alpha values.
U32 qThetas[NumQuants-1]; //Encoded array of quantized U32 Theta values.
```

For details on U32 values encoding, see "Integer data encoding" on page 30.

Alpha/theta is converted into an F32 value with this formula:

```
F32Val = U32Val * (2 * Pi) / NumQuants;
```

Decoded normal is evaluated using these formulas:

```
nx = sin(theta) * cos(alpha)
ny = sin(theta) * sin(alpha)
nz = cos(theta)
```

Integer data encoding

UINT data can be encoded in a few ways. The first byte of the encoded UINT data block stores the encoding type. The encoder automatically determines the best way to encode data: by evaluating sizes of encoded data for some encoding methods (without doing the encoding) and by actually encoding the data and checking the size of the encoded block for others (RH39, arithmetic static, arithmetic dynamic, and UIC1).

Integer data is encoded as described in the following table.

Bit position	Description
0,1,2,3,4,5	Encoding type (maximum 64 types, currently 11 types defined, described in "Encoding types" on page 31)
6,7	Number of bytes required for encoding buffer size, as follows: 0 indicates 1 byte (U8) 1 indicates 2 bytes (U16) 2 indicates 3 bytes (U24) 3 indicates 4 bytes (U32)

Other data encoding types

This section describes the other data encoding types used in the RHAdobeMeshResource type.

Encoding types

The following table describes the encoding types. It is possible to remove encoding methods 0 - 6 with minimal increase in file size. An arithmetic encoder with static context is able to efficiently encode this data.

EncodingType	Description
0	All values are zero. This happens rather often when encoding Material IDs. The total size of the chunk in this case is one byte (only the "U8 EncodingType").
1	All values are equal to a specified value. The U32 value is specified after the EncodingType. The size of the chunk is five bytes: 1) U8 EncodingType=1 2) U32 value_to_be_replicated
2	Values are stored in four bits. This chunk type may be useful for encoding Material IDs in case where there are more than one and fewer than 17 materials per mesh. Chunk size: 1 + (NumberOfUINTs / 2) + (NumberOfUINTs % 2) Chunk format: 1) U8 EncodingType=2 2) U8[(NumberOfUINTs / 2) + (NumberOfUINTs % 2)]. Each byte stores 2 values - higher 4 bits are used for values with odd index, lower - for even.
3	Values are stored in one byte. Encoder uses this method if the maximum value in the input array is 255 or less. Chunk size: 1 + NumberOfUINTs Chunk format: 1) U8 EncodingType=3 2) U8[NumberOfUINTs]
4	Values are stored in two bytes. Encoder uses this method if the maximum value in the input array is 65535 or less. Chunk size: 1 + NumberOfUINTs*2 Chunk format: 1) U8 EncodingType=4 2) U16[NumberOfUINTs]

EncodingType	Description
5	Values are stored in three bytes. Encoder uses this method if the maximum value in the input array is 16777215 or less. Chunk size: 1 + NumberOfUINTs*3 Chunk format: 1) U8 EncodingType=5 2) U24[NumberOfUINTs]
6	Values are stored in four bytes. This is the worst case scenario. Chunk size: 1 + NumberOfUINTs*4 Chunk format: 1) U8 EncodingType=6 2) U32[NumberOfUINTs]
7	RH39 encoding. The encoder has to perform the actual compression in order to know the size of the compressed data. Chunk format: U8 EncodingType=7 RH39-encoded data, as described "RH39 data encoding" on page 33.
8	Arithmetic encoding (Static context) as in U3D format specification. The encoder has to perform the actual compression in order to know the size of the compressed data. Chunk format: 1) U8 EncodingType=8 2) U32 size of arithmetic-coded data - it is possible avoid writing this U32, but probably minor changes in IBXBitStreamCompressedX class are required. 3) Data encoded with arithmetic encoder

EncodingType	Description
9	Arithmetic encoding (Dynamic context) as in U3D format specification. The encoder has to perform the actual compression in order to know the size of the compressed data. Chunk format: 1) U8 EncodingType=9 2) U32 size of arithmetic-coded data - it is possible to avoid writing this U32, but probably minor changes in IBXBitStreamCompressedX class are required. 3) Data encoded with arithmetic encoder For static context the probabilities are pre-evaluated, for dynamic - they are populated at run-time. See section 10.4 of the *Universal 3D File Format*.
10	UIC1 compression. The encoder has to perform the actual compression in order to know the size of the compressed data. Chunk format: U8 EncodingType=10 UIC1-encoded data. See UIC1 format description in paragraph "UIC1 data encoding" on page 34.

RH39 data encoding

Data consists of eight different operators/commands followed by data. An operator also stores the length of optional data or in some cases an offset used to determine position within data. Operators can be single byte or double byte (depending on the length of the optional data). Bit 7 specifies whether an operator is single or double byte.

For single-byte operators, bit 7 is not set, bits 4,5,6 store operator type, bits 0,1,2,3 store data length. Data length can be from 1 to 16 (0 means 1 byte, …, 15 means 16 bytes).

For a double-byte operator, bit 7 of the first byte is set, bits 4,5,6 of the first byte store operator type, bits 0,1,2,3 of the first byte and bits 0..7 of the second stored data length. The length value assembled from the first and second bytes is 17 less than the actual data length, reflecting the fact that a data length less than 17 is represented with a single-byte operator. More specifically, you must add 17 to the length value assembled from the first and second bytes.

Data length can be from 17 to 4112.

```
U8 opcodeandlen=processor.ReadByte();
U8 opcode=(opcodeandlen>>4) & 7;
U32 len=opcodeandlen & 0x0f;
if(opcodeandlen&0x80) //2 bytes
   len=17 + (len<<8) + processor.ReadByte();
else//1 byte
   len++;
```

For an implementation that decodes U32 data, see the appendix, "Decoding Procedures for Right Hemisphere Adobe 3D Mesh Compression" on page 44.

The following table shows the relationship between operators, data types, and data length calculations. In this table, the term *length* is synonyms with *data length*. Usually, an operator is followed by data. Data type depends on the operator type. It also affects the data length.

Operator	Data type of the subsequent data	Decoding data length and using data length to read subsequent data

0	Raw U32 values	Size of data is 4*length bytes (four times the data-length value).
		`for(U32 i=0;i<length;i++)`
		`  value[i]=ReadU32;`

1	U4 positive difference relative to current "offset" value	Size of data is (Length/2 + length%2) bytes
		`if(odd)`
		`  val=offset + (b4>>4); else//even {`
		`  b4=processor.ReadByte();`
		`  val=offset + (b4&0x0f); }`

2	U8 positive difference relative to current "offset" value	Size of data is 1*length bytes.
		`for(U32 i=0;i<length;i++)`
		`  value[i]=offset + ReadU8();`

3	U16 positive difference relative to current "offset" value	Size of data is 2*length bytes.
		`for(U32 i=0;i<length;i++)`
		`  value[i]=offset + ReadU16();`

4	Raw U16 values	Size of data is 2*length bytes.
		`for(U32 i=0;i<length;i++)`
		`  value[i]=ReadU16();`

5	None	Values identical to current "offset". No optional data follows this operator.
		`for(U32 i=0;i<length;i++)`
		`  value[i]=offset;`

6	Raw U8 values	Size of data is 1*length bytes.
		`for(U32 i=0;i<length;i++)`
		`  value[i]=ReadU8();`

7	Various types, depending on lower 4 bits of the operator byte	Specifies the current "offset" value. The number of bytes in the offset value can be 1, 2, 3 or 4. This is controlled by the lower 4 bits of the operator byte.
		`if(length==0) offset=ReadU8();`
		`else if(length==1) offset=ReadU16();`
		`else if(length==2) offset=ReadU24();`
		`else if(length==3) offset=ReadU32();`

UIC1 data encoding

This algorithm is very good at encoding unsigned integer values, when the values are somewhat similar. In other words, it would effectively compress: 10, 14, 18, 10, 11, 40, 6413, 6420, 19, 3, 6410… But will give

poor results on this data: 7831, 5, 98234, 4612, 112, 3312… This means that it is good at encoding face indices, material indices and even quantized floating points.

The stream is encoded with different commands. Commands are 4-bit (total of 16 commands). The command is stored in the high four bits of a byte. The low four bits are occupied by the operand. The command with id=2 is not defined and cannot be encountered.

Implementation uses two small arrays of size 32: one for storing previous decoded values and one for deltas. Some commands refer to these arrays.

The last decoded value is always appended to the array of decoded values.

The delta between previous decoded value and the current decoded value is added to the delta array only if the values are different (delta!=0) and abs(delta)<0x7FFFFFFF.

Note: Previous vertical values are used only in face index encoding. Consider the following sample:

10 12 14

13 34 62

60 11 55

The following table shows the previous application of vertical values.

Value	Provides vertical value for
10	13
12	34
14	62
13	60
34	11
62	55

All pseudocode below uses the following variables:

```
UINT8 cmd=ReadU8();         //Together with the operand
UINT32 cmdshift4=cmd>>4;    //Operand
```

For an implementation that decodes UIC1 data, see the appendix, "Decoding Procedures for Right Hemisphere Adobe 3D Mesh Compression" on page 44.

Single byte commands in which decoded UINT values are represented as single bytes

Operator	Operator name	Description and example implementation
0	**BackPtrValueU4_1**	The decoded UINT should be taken from the array of previous values. The index into the array is stored in the 4 bits of the command. If the previous value is equal to the current value, then the current value MUST be encoded as BackPtrValueU4_1(0). `rvalue=GetBackValue(cmdshift4);`
5	BackPtrValueU4_2	Similar to BackPtrValueU4_1, but the index into the array of previous values is 16+operand. `rvalue=GetBackValue(16+cmdshift4);`

1	DeltaI4	Signed increment for the value (+1..+8) or (-1..-8). Mapping is:
		0 -> +1, 1 -> +2, …, 7 -> +8, 8 -> -1, 9 -> -2, …, 15 -> -8.
		`if(cmd&16)`
		`  rvalue+=(cmd>>5) + 1;`
		`else`
		`rvalue-=(cmd>>5) + 1;`

6	BackPtrDeltaU4	Value is increased by a delta from the previous deltas array. `rvalue+=GetBackDelta(cmdshift4); double byte`
3	DeltaU4Positive_V	Value is set to previous vertical value + operand + 1 `rvalue=prevVertical + (cmdshift4 + 1);`
4	DeltaU4Negative_V	Value is set to previous vertical value – operand - 1 `rvalue=prevVertical - (cmdshift4 + 1);`
7	DeltaU4Positive_V2	Value is set to previous vertical value + 17 + operand `rvalue=prevVertical + (cmdshift4 + 17);`
8	DeltaU4Negative_V2	Value is set to previous vertical value - 17 - operand `rvalue=prevVertical - (cmdshift4 + 17);`
9	DeltaU12Positive	Value += (3..4099) `rvalue+=cmdshift4 + (((UINT32)ReadU8())<<4) + 3;`
10	DeltaU12Negative	Value -= (3..4099) `value-=cmdshift4 + (((UINT32)ReadU8())<<4) + 3;`
11	ValueU12	A 12 bit value `rvalue=cmdshift4 + (((UINT32)ReadU8())<<4);` 3 byte command (20 bits for operand)
12	ValueU20	20 bit value rvalue=cmdshift4 + (((UINT32)ReadU16())<<4);

Multiple-byte commands (1/3/4/5 byte(s))

Operator	Constant name	Description and example implementation

15	Extended	Consists of 14 subcommands (subcommand is stored in operand):
		`0=Eq_00000000, rvalue=0;`
		`1=Eq_00000001, rvalue=1;`
		`2=Eq_00000002, rvalue=2;`
		`3=Eq_00000003, rvalue=3;`
		`4=Eq_00000004, rvalue=4;`
		`5=Eq_000000FF, rvalue=0xFF;`
		`6=Eq_0000FF00, rvalue=0xFF00;`
		`7=Eq_00FF0000, rvalue=0xFF0000;`
		`8=Eq_FF000000, rvalue=0xFF000000;`
		`9=Eq_00FFFFFF, rvalue=0x00FFFFFF;`
		`10=Eq_FFFFFFFF, rvalue=0xFFFFFFFF;`

11	ValueU24	A 24 bit value
		`UINT32 bl=ReadU8();`
		`rvalue=((UINT32)ReadU16()<<8)+bl;`

12	ValueU32	A 32 bit value (worst case 4 bytes -> 5 bytes) `rvalue=ReadU32();`
13	LongReplicate16	Followed by a U16 repeat count. The last value will be replicated 36+ReadU16() times (36..65571).
14		Not defined.
15		Not defined.

Batch commands (less than one byte per UINT)

Operator	Constant name	Description and example implementation

13	BackPtrValueU22	Two values in one byte. First value should be taken from the array of previous values with index [0..3], second value should also be taken from the array of previous values [0..3].
		`rvalue=GetBackValue(cmdshift4>>2);`
		`Next Value would be GetBackValue(cmdshift4&3);`

14	Replicate	Short replicate command. The last value will be repeated 3..18 times (depending on the operand). Very short replicate (2 equal values in a row) would be BackPtrValueU22(0,0). Also see LongReplicate16 command in "Multiple-byte commands (1/3/4/5 byte(s))" on page 37.

Future Expansion SubChunks

This point in the chunk format, after all the above defined data, is reserved for future expansion subchunks. These consist of areas of data that are not obliged to be read under the current version of this specification. Each subchunk shall be referenced by an entry within data described by "SubChunk extension data (optional)" on page 22.

Examples

Encoding of a 1x1x1 cube.

Source vertex coordinates array:

#	X	Y	Z
0	-0.5	-0.5	0.5
1	0.5	-0.5	0.5
2	-0.5	0.5	0.5
3	0.5	0.5	0.5
4	-0.5	0.5	-0.5
5	0.5	0.5	-0.5
6	-0.5	-0.5	-0.5
7	0.5	-0.5	-0.5

Encoding (63 bytes total):

02 00 00 00 bf 00 00 00 3f 84 eb 0f 00 0a 07 00 4c b8 fe 5d 5d 5d 00 00 00 bf 00 00 00 3f 84 eb 0f 00 0a 07 0d 4c b8 fe 0e 40 00 00 00 00 bf 00 00 00 3f 84 eb 0f 00 0a 06 4c b8 fe 0e 40 0e

Bytes for encoding X values are marked red, bytes for encoding Y values are marked green, and bytes for encoding Z values are marked blue).

Detailed description of encoded bytes

02 – encoding type "2", see "Type 2 (FPDATATYPE_QUANTIZEDCOMPRESSED_FLOATS)" on page 28

00 00 00 bf – minimum X value (float -0.5)

00 00 00 3f – maximum X value (float 0.5)

84 eb 0f 00 – quant (unsigned int 1043332)

The source array of X values (-0.5, 0.5, -0.5, 0.5, -0.5, 0.5, -0.5, 0.5) is quantized into unsigned int array (0, 1043332, 0, 1043332, 0, 1043332, 0, 1043332). The quantized array is encoded as (0a 07 00 4c b8 fe 5d 5d 5d).

0a – encoding type "10" (UIC1 compression), see "10" on page 33

07 – encoded buffer size (7 bytes)

The following 7 bytes (00 4c b8 fe 5d 5d 5d) are a UIC1-encoded array of unsigned int values (0, 1043332, 0, 1043332, 0, 1043332, 0, 1043332), see "UIC1 decoding of array" on page 38.

Y and Z coordinates are encoded in the similar manner (min, max, quant, UIC1-encoded array).

UIC1 decoding of array

This section describes the result of decoding this UIC1-encoded array:

```
00 4c b8 fe 5d 5d 5d
```

Data	Command and length	Decoded result
00	`BackPtrValueU4_1(0)`	Value is “0”. Array: (0)
4c b8 fe	`ValueU20(1043332)`	Value is “1043332”. Array: (0, 1043332)
5d	`BackPtrValueU22(1, 1)`	Values are “0” and “1043332”. Array: (0, 1043332, 0, 1043332)
5d	`BackPtrValueU22(1, 1)`	Values are “0” and “1043332”. Array: (0, 1043332, 0, 1043332, 0, 1043332)
5d	`BackPtrValueU22(1, 1)`	Values are “0” and “1043332”. Array: (0, 1043332, 0, 1043332, 0, 1043332, 0, 1043332)

Decoded values: (0, 1043332, 0, 1043332, 0, 1043332, 0, 1043332)

See also “UIC1 data encoding” on page 34.

3 U3D Meta Data

This chapter specifies extensions to the U3D format to specify the storage of meta data derived from other 3D files reformatted and represented within U3D. This format enables viewer applications to distinguish between meta data gleaned from a CAD file and meta data added by a third party.

Acrobat 8.1 and later supports the U3D meta data format described by this chapter.

Conventions

The section numbers and headings used in this section correspond to the section numbers and headings in the *Universal 3D File Format*, 3rd Edition. These section numbers and headings are italicized to distinguish them from section headings of this guide.

For definitions of type information terms used in this document, refer to section *5.2 Data types* of the *Universal 3D File Format*, *3rd Edition*.

U3D meta data

U3D meta data uses a single Key/Value pair within the U3D meta data structure *9.2.6 Meta data* of each block header where file meta data is desired. Meta data strings are stored using a character storage format as defined within the U3D field *9.4.1.5 U32: Character Encoding*. For the editions of U3D supported, the sole valid encoding is UTF-8. At most one U3D meta data entry can be provided within a U3D Block structure (defined in *9.2 Block structure)*.

U3D meta data requires that the first meta data node entry of the *9.4.1 File Header (blocktype: 0x00443355)* contain the following fields, in sequence:

1. *9.2.6.2 U32: Key/Value Pair Attributes*, with a value indicating string content (0x00000000)
2. *9.2.6.3 String: Key String*, containing the string:

   ```
   RHAdobeMeta
   ```

3. *9.2.6.4 String: Value String*, containing one or more namespaces ("Namespace meta data" on page 40). Where possible, structures have been designed to approximate a simplified XML. Within this value string, discrete words within the XML-like structure are delineated by the space character (single byte decimal 32 for UTF-8).

Note: Each appearance of U3D meta data supports a flat list of meta data properties. That is, it does not support nesting of meta data.

Namespace meta data

Namespace meta data consists of a Namespace identifier, followed by one or more Item name and value pair entries in succession, followed by Namespace terminator. For example:

```
<namespace name="24578"/>
```

Namespace identifier

A namespace identifier consists of a Namespace name identifier, followed by a Namespace name string, and then a Namespace identifier terminator.

Namespace name identifier

This consists of the string'<namespace name=', containing the '<' character - (single byte decimal 60 for UTF-8), followed by the nine characters 'namespace', the space character (single byte decimal 32), the four characters 'name' and the '=' character (single byte decimal 61).

Namespace name string

The name of the namespace, stored in a format as specified in "Storage of strings from the original data source" on page 42.

Namespace identifier terminator

This consists of the '>' character - (single byte decimal 62).

Item name and value pair

This contains a single name and value pair. It is represented by one each of the following fields in the following order (Item identifier to Namespace identifier terminator) . For example:

```
<item name=anItemName value="theItemsValue"/>
```

Item identifier

This consists of the string '<item', or more specifically the '<' character - (single byte decimal 60) followed by the four characters 'item'.

Item name identifier

This consists of the string 'name=', or more specifically the characters 'name' followed by the '=' character (single byte decimal 61) .

Item name string

The name associated with an item, stored in a format as specified by "Storage of strings from the original data source" on page 42.

Item name terminator and value identifier

This consists of the string 'value=', or more specifically a sequence of five characters 'value' followed by the '=' character (single byte decimal 61).

Item value string

The value associated with an item, stored in a format as specified by "Storage of strings from the original data source" on page 42.

Item identifier terminator

This consists of the string '/>', or more specifically the '/' character - (single byte decimal 47), followed by the '>' character - (single byte decimal 62).

Namespace terminator

This consists of the string '</namespace>', or more specifically the '<' character - (single byte decimal 60), followed by the '/' character - (single byte decimal 47), the eight characters 'namespace', and the '>' character - (single byte decimal 62).

Storage of strings from the original data source

Strings from the original data source are represented by an internal sequence of characters surrounded by a starting and ending double quote character (single byte decimal 34). The character sequence is stored in a format as described in *9.4.1.5 U32: Character Encoding*.

In conformance with storage of XML strings, the single-byte characters detailed in the table "Representation of characters" on page 429 are not legal within the internal sequence of characters. To represent them, use the character entity reference (CER) corresponding to each one as detailed in the table.

In addition, the following characters should not be included within a valid internal sequence of characters: 1-8 decimal (0x01 - 0x08 hex), 11-12 decimal (0x0B-0x0C hex), and 14-31 decimal (0x0E-0x1F hex).

Representation of characters

Character	Decimal byte value	CER
"	34	"
&	38	&
<	60	<
>	62	>
'	39	'
Tab	9	
LF	10
CR	13

Example

The following example shows a single namespace containing two items.

```
<namespace name="24578">
   <item name="Area:" value="377.092 inch^2"/>
   <item name="Density:" value="0.036 lbmass/inch^3"/>
</namespace>
```

4 New Features and Changes

This chapter summarizes the new features and changes introduced in Acrobat 8.1 and earlier.

Acrobat 8.1

Block extensions

Acrobat 8.1 adds support for a more current version of ECMA-363, called *Universal 3D File Format, 3rd Edition*. This edition enables extensions of the pre-defined block types defined in the specification. Acrobat 8.1 takes advantage of these extensions by adding support for the following features:

- Right Hemisphere Adobe 3D Compressed Mesh, which is a more compressed form of mesh blocks. For information on this format, see "Right Hemisphere Adobe 3D Mesh Compression" on page 20.
- U3D meta data, which specify the storage of meta data derived from other3D files reformatted and represented within U3D. For more information on this format, see "U3D Meta Data" on page 40.

New expression using existing blocks

Acrobat 8.1 adds support for a standardized expression of units scaling factor data. For more information on this format, see "Units scaling-factor meta data" on page 7.

Acrobat 8.0

Adds support for the *9.4.1.2 U32: Profile Identifier* data.

Adds support for multi-level multi-parenting in *9.5.X.2.1 U32: Parent Node Count*.

A Decoding Procedures for Right Hemisphere Adobe 3D Mesh Compression

This appendix presents examples of data decoding procedures used with Right Hemisphere Adobe 3D Mesh compression. These examples are for a U32 data decoder (below) and a UIC1 data decoder (page 46).

U32 data decoder

The following example shows an implementation of a U32 data decoder. For more information on U32 data encoding, see "UIC1 data encoding" on page 34.

Example A.1 ***U32 data decoder***

```
#define COMPRESSION_TYPE_ALL_ZERO 0
#define COMPRESSION_TYPE_REPLICATE 1
#define COMPRESSION_TYPE_U4 2
#define COMPRESSION_TYPE_U8 3
#define COMPRESSION_TYPE_U16 4
#define COMPRESSION_TYPE_U24 5
#define COMPRESSION_TYPE_U32 6
#define COMPRESSION_TYPE_RH397
//.RH39 coding commands
#define RH39OPCODE_RAWU8 6//raw U8 values
#define RH39OPCODE_RAWU16 4//raw U16 values
#define RH39OPCODE_RAWU32 0//raw U32 values
#define RH39OPCODE_OFFSET 7//'offset' value, operand is the number of bytes in
offset (can be 1|2|3|4)
#define RH39OPCODE_REPLICATE 5//identical values
#define RH39OPCODE_DIFFERENCE4 1//U4 positive delta values (relative to
'offset')
#define RH39OPCODE_DIFFERENCE8 2//U8 positive delta values (relative to
'offset')
#define RH39OPCODE_DIFFERENCE16 3//U16 positive delta values (relative to
'offset')
bool DecompressArray(U8 *pEncoded, U32 *pDecoded, U32 count, U32 EndodedSize)
{
   U32 offset=0;
   cRH39BufferProcessor processor(pEncoded, EndodedSize);
   U8 EncodingType=processor.ReadByte();
   if((EncodingType==COMPRESSION_TYPE_ALL_ZERO) ||
      (EncodingType==COMPRESSION_TYPE_REPLICATE))
   {//all equal
      U32 val=(EncodingType==
         COMPRESSION_TYPE_ALL_ZERO)? 0 : processor.Read(4);
      for(U32 i=0;i<count;i++) pDecoded[i]=val;
      return true;
   }
   else if(EncodingType==COMPRESSION_TYPE_U4)
   {//half bytes
      U8 b4;
      for(U32 i=0;i<count;i++)
```

```
      {
        if(i&1)
          pDecoded[i]=b4>>4;
        else
        {
          b4=processor.ReadByte();
          pDecoded[i]=(b4&0x0f);
        }
      }
      return true;
    }
    else if((EncodingType>=COMPRESSION_TYPE_U8) &&
      (EncodingType<=COMPRESSION_TYPE_U32))
    {//1|2|3|4 bytes per value
      for(U32 i=0;i<count;i++)
        pDecoded[i]=processor.Read(1+EncodingType-COMPRESSION_TYPE_U8);
      return true;
    }
    else if(EncodingType==COMPRESSION_TYPE_RH39)
    {//RH compression based on .rh35 format by Alexander Chelemekhov,
improvements by Dmitriy Kivilev
      while(count)
      {
        if(processor.Overflow())
          break;//input buffer error
        U8 opcodeandlen=processor.ReadByte();
        U8 opcode=(opcodeandlen>>4) & 7;
        U32 len=opcodeandlen & 0x0f;
        if(opcodeandlen&0x80)//2 bytes
          len=17 + (len<<8) + processor.ReadByte();
        else//1 byte
          len++;
        //just read the offset value [1|2|3|4 bytes]
        if(opcode==RH39OPCODE_OFFSET)
          offset=processor.Read(len);
        else

        {//all 7 other opcodes
          if(count<len)
            break;//wrong input buffer
          count-=len;
          U8 b4;
          for(U32 i=0;i<len;i++)
          {
            U32 val;
            if(opcode==RH39OPCODE_RAWU8)
              val=processor.Read(1);
            else if(opcode==RH39OPCODE_RAWU16) val=processor.Read(2);
            else if(opcode==RH39OPCODE_RAWU32) val=processor.Read(4);
            else if(opcode==RH39OPCODE_DIFFERENCE8) val=
              offset + processor.Read(1);
            else if(opcode==RH39OPCODE_DIFFERENCE16) val=
              offset + processor.Read(2);
            else if(opcode==RH39OPCODE_REPLICATE)
              val=offset;
```

```
            else//RH39OPCODE_DIFFERENCE4
            {
               if(i&1)
                  val=offset + (b4>>4);
               else {
                  b4=processor.ReadByte();
                  val=offset + (b4&0x0f);
               }
            }
            *pDecoded++=val;
         }
      }
   }
   return !count;
}
return false;//unknown encoding type
}
```

UIC1 data decoder

The following example shows an implementation of a UIC1 data decoder. For more information on UIC1 data encoding, see "RH39 data encoding" on page 33.

Example A.2 ***UIC1 decoder***

```
UINT32 PrevVals[MAX_PREV_VALS];
INT32 PrevDeltas[MAX_PREV_DELTAS];
UINT32 CurValPos, CurDltPos;
UINT32 GetBackValue(UINT32 which)
{
   return PrevVals[(CurValPos>=which)? (CurValPos-which) :
      (MAX_PREV_VALS+CurValPos-which)];
}

INT32 GetBackDelta(UINT32 which)
{
   return PrevDeltas[(CurDltPos>=which)? (CurDltPos-which) :
   (MAX_PREV_DELTAS+CurDltPos-which)];
}

void AddBackValue(UINT32 value)
{
   CurValPos++;
   if(CurValPos==MAX_PREV_VALS)
      CurValPos=0;
   PrevVals[CurValPos]=value;
}
void AddBackDelta(INT32 value)
{

   urDltPos++;
   if(CurDltPos==MAX_PREV_DELTAS)
      CurDltPos=0;
   PrevDeltas[CurDltPos]=value;
```

```
}

UINT32 InValueQueue=0;
UINT8 queueval;

void ReadCommand(UINT32 &rvalue, UINT32 prevVertical)
{
   UINT32 prevval=rvalue;
   if(InValueQueue)
   {
      InValueQueue--;
      rvalue=GetBackValue(queueval);
   }
   else
   {//command

      UINT8 cmd=ReadU8();//together with the operand
      UINT32 cmdshift4=cmd>>4;
      switch(cmd&0xF)
      {
      case COMMAND_BackPtrValueU4_1:
         rvalue=GetBackValue(cmdshift4);
         break;

         case COMMAND_BackPtrValueU4_2:
            rvalue=GetBackValue(16+cmdshift4);
            break;

         case COMMAND_BackPtrValueU22:
            rvalue=GetBackValue(cmdshift4>>2);
            queueval=cmdshift4&3;
            InValueQueue=1;
            break;

         case COMMAND_Replicate:
            rvalue=GetBackValue(0);
            queueval=0;
            InValueQueue=cmdshift4+2;
            break;

         case COMMAND_BackPtrDeltaU4:
            rvalue+=GetBackDelta(cmdshift4);
            break;

         case COMMAND_DeltaI4:
            if(cmd&16)
               rvalue+=(cmd>>5) + 1;
            else
               rvalue-=(cmd>>5) + 1;
            break;

         case COMMAND_DeltaU4Positive_V:
            rvalue=prevVertical + (cmdshift4 + 1);
            break;
```

```
case COMMAND_DeltaU4Negative_V:
   rvalue=prevVertical - (cmdshift4 + 1);
   break;

case COMMAND_DeltaU4Positive_V2:
   rvalue=prevVertical + (cmdshift4 + 17);
   break;

case COMMAND_DeltaU4Negative_V2:
   rvalue=prevVertical - (cmdshift4 + 17);
   break;

case COMMAND_DeltaU12Positive:
   rvalue+=cmdshift4 + (((UINT32)ReadU8())<<4) + 3;
   break;

case COMMAND_DeltaU12Negative:
   rvalue-=cmdshift4 + (((UINT32)ReadU8())<<4) + 3;
   break;

case COMMAND_ValueU12:
   rvalue=cmdshift4 + (((UINT32)ReadU8())<<4);
   break;

case COMMAND_ValueU20:
   rvalue=cmdshift4 + (((UINT32)ReadU16())<<4);
   break;

case COMMAND_Extended:
   switch(cmd>>4)
   {
      case EXTENDED_COMMAND_Eq_00000000:
      case EXTENDED_COMMAND_Eq_00000001:
      case EXTENDED_COMMAND_Eq_00000002:
      case EXTENDED_COMMAND_Eq_00000003:
      case EXTENDED_COMMAND_Eq_00000004:

         value=cmd>>4;
         break;

      case EXTENDED_COMMAND_Eq_000000FF: rvalue=0x000000FF;   break;
      case EXTENDED_COMMAND_Eq_0000FF00:   rvalue=0x0000FF00;   break;
      case EXTENDED_COMMAND_Eq_00FF0000:   rvalue=0x00FF0000;   break;
      case EXTENDED_COMMAND_Eq_FF000000:   rvalue=0xFF000000;   break;
      case EXTENDED_COMMAND_Eq_00FFFFFF:   rvalue=0x00FFFFFF;   break;
      case EXTENDED_COMMAND_Eq_FFFFFFFF:  rvalue=0xFFFFFFFF;   break;
      case EXTENDED_COMMAND_LongReplicate16:
         queueval=0;
         InValueQueue=ReadU16() + 36;
         break;
      case EXTENDED_COMMAND_ValueU24:
      {UINT32 bl=ReadU8();rvalue=((UINT32)ReadU16()<<8)+bl;}
         break;

      case EXTENDED_COMMAND_ValueU32:
```

```
                rvalue=ReadU32();
                break;
            }
            break;

      }
   }
   AddBackValue(rvalue);
   if(rvalue!=prevval)
   {
//we don't put "0" into the delta buffer, because cur==prev is encoded
differently
      if(rvalue>prevval)
      {
         UINT32 d=rvalue-prevval;
         if(d<0x7FFFFFFF)//no overflow
            AddBackDelta((INT32)d);
      }
      else
      {//rvalue<prevval

         UINT32 d=prevval-rvalue;
         if(d<0x7FFFFFFF)//no overflow
            AddBackDelta(-(INT32)d);
      }
   }
}
```