Standard ECMA-363

1st Edition / December 2004

Universal 3D File Format

Standard

Rue du Rhône 114   CH-1204 Geneva   T: +41 22 849 6000   F: +41 22 849 6001

Standard

ECMA-363

1st Edition / December 2004

Universal 3D File Format

Brief history

In 2004, Ecma International formed Technical Committee 43 to specify Universal 3D (U3D) File Format specification. The Universal 3D File Format Specification is primarily intended for downstream 3D CAD repurposing and visualization purposes.

This Ecma Standard has been adopted by the General Assembly of December 2004.

Table of contents

1 Scope

This Standard defines the syntax and semantics of the Universal 3D file format, an extensible format for downstream 3D CAD repurposing and visualization, useful for many mainstream business applications. Salient features of the U3D file format described in this document include: execution architecture that facilitates optimal run-time modification of geometry, continuous-level-of-detail, domain-specific compression, progressive data streaming and playback, key-frame and bones-based animation, and extensibility of U3D format and run-time.

The U3D file format specification does not address issues regarding rendering of 3D content.

The U3D file format specification does not address issues regarding reliability of the transport layer or communications channel. It is assumed that reliability issues will be addressed by a different protocol layer.

NOTE
Extensibility feature will not be addressed in the first edition of this Standard, however, it will be added as a primary feature for the next edition of this Standard.

2 Conformance

A comforming implementation complies with all the mandatory clauses in this Standard.

3 References

American National Standards Institute, American National Standard for Information Systems--Programming Language--C, ANSI X3.159-1989

ECMA-335: Common Language Infrastructure (CLI), 2nd edition (December 2002) (ISO/IEC 23271)

IEEE Computer Society (1985), IEEE Standard for Binary Floating-Point Arithmetic, IEEE Std 754-1985.

IETF RFC 3629: UTF-8, a transformation format of ISO 10646. November 2003, http://www.faqs.org/rfcs/rfc3629.html.

ISO/IEC IS 10918-1 | ITU-T Recommendation T.81: JPEG, 1994, http://www.jpeg.org/jpeg.

TIFF™ 6.0 Specification, Adobe Systems Incorporated, June 1992.

W3C Recommendation on 1st October, 1996: Portable Network Graphics (PNG), ISO/IEC 15948:2003 (E)., http://www.w3.org/Graphics/PNG/.

4 Definitions

Term	Definition
Glyph	A symbolic figure, image, or shape that conveys information.
Mandatory clauses	All portions of the spectification except those marked “Informative”.
New Position	A position that is added to the mesh, point set, or line set.

Term	Definition
One Point projection	Working in a similar manner to a bellows camera, with one point projection the orientation of the image plane is completely independent of the view direction. An advantage of one point projection is that of dimensional correctness. That is, if the image plane is parallel to a plane of the model, then the dimensions of the model in that plane are to scale. This enables creation of an image that has both depth and dimensional correctness in the selected plane.
Resolution	Level of detail.
Screen	Drawing area available for rendering.
Split Position	A position in a mesh, point set or line set from which the new position will be created. The new position is described relative to the split position.
Third Position	A new face added to an author mesh uses three positions: the New Position, the Split Position, and the Third Position.
Three point projection	Is the most natural projection, and is used for conventional images. In this projection, the image plane is normal to the direction of the view, as in a conventional camera.

5 Notational Conventions

5.1 Diagrams and field descriptions

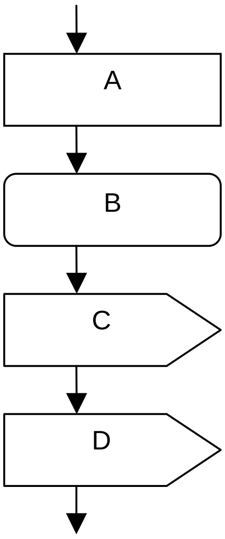
Boxes represent information stored as one of the basic data types described in 5.2 Data types. Ovals represent a logical collection of more than one of the basic data types. The information is grouped for clarity and the basic data types that compose the grouping are described explicitly in a following subsection of the document. Boxes with the right side corners cut off represent information that is compressed. Arrows convey ordering information.

Each entry in the diagram is further documented below the diagram. The logical groups are noted by name only. Basic data types are noted by an abbreviated data type symbol (defined in the next section) and the field name. The compressed data is documented as basic data type; followed by an open bracket "[", the compression context, and a close bracket, "]";followed by the field name.

The compression context identifies symbols with similar frequency statistics. The compression routine adapts to the frequency of symbols encountered with adaptive contexts. Adaptive context labels are prefixed with "c". The compression routine can take advantage of range limitation information. A range context indicates the symbols only use a limited portion of the range of the data type. Range contexts are not adaptive. Range context labels are prefixed with "r".

Clause 10 Bit Encoding Algorithm contains details regarding compression requirements.

Example: The following diagram and field description shows Data A, an unsigned 8-bit integer; followed by Data B, a grouping of multiple fields; followed by Data C, a compressed unsigned 32-bit integer with an adaptive context "cCcontext"; followed by Data D, a compressed unsigned 8-bit integer with a range context of 0 to 6.



The fields are then noted as follows:

1.1.1.1.1 U8: A

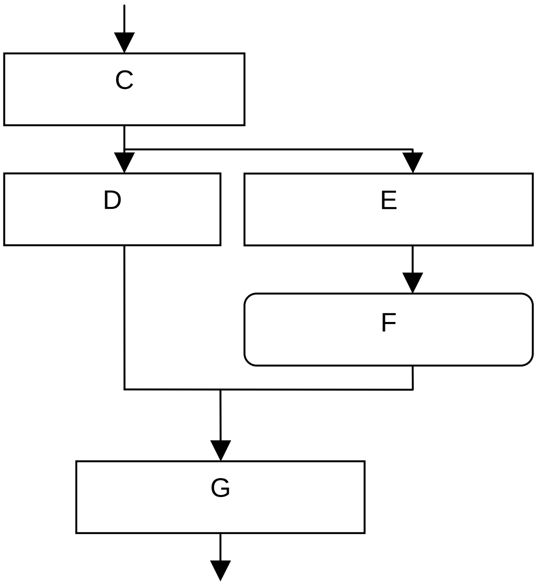
1.1.1.1.2 B

1.1.1.1.3 U32 [cCcontext]: C

1.1.1.1.4 U8 [r7]: D

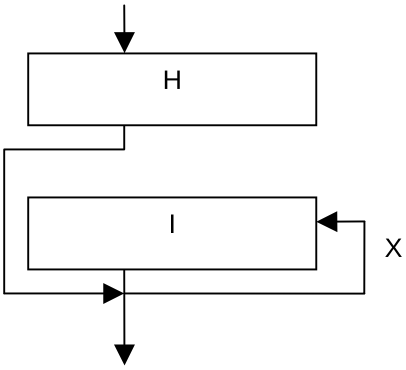
An arrow with a branch in its shaft represents two or more options for information to be stored in the file.

Example: The following diagram shows C followed by D followed by G, or C followed by E followed by F followed by G.



If the same data type repeats several times, a loop is used. The number of iterations appears next to the loop arrow. The number of iterations may depend on information presented earlier in the file.

Example: The following diagram shows data H followed by X data I.



Numbers:

By default, all numbers are decimal (base 10). Numbers prefixed with “0x” are hexadecimal numbers (base 16). For example, the number “0x10” is a hexadecimal number equivalent to the decimal number “16”.

5.2 Data types

The binary file will contain the following types: U8, U16, U32, U64, I16, I32, F32, and String. Clause 10 contains encoding requirements for these types.

5.2.1 U8

An unsigned 8-bit integer value.

5.2.2 U16

An unsigned 16-bit integer value.

5.2.3 U32

An unsigned 32-bit integer value.

5.2.4 U64

An unsigned 64-bit integer value.

5.2.5 I16

A signed two’s complement 16-bit integer value.

5.2.6 I32

A signed two’s complement 32-bit integer value.

5.2.7 F32

An IEEE 32-bit floating-point number.

5.2.8 String

The String type starts with an unsigned 16-bit integer that defines how many bytes of character data the string contains. The character encoding is defined per file in the file header block. Strings are always handled as case sensitive.

The empty string contains zero bytes of character data. The emptry string is used to indicate the name of a default palette entry. A field may use the empty string as a name referring to the default entry. The empty string shall not be used as the name of an object defined by a block in the file.

5.3 Functional notations

Some text descriptions use a functional notation for color values or quantized values. Those functions are described in this section.

5.3.1 rgb(R,G,B)

A color value with red, green, and blue components can be described using rgb(R,G,B). The values for R, G, and B indicate the intensity of that component. A value of 0.0 indicates black and a value of +1.0 indicates full intensity. The ordinary range is 0.0 to +1.0 although values outside this range are allowed. Gray colors are indicated by using the same value for R, G, and B.

5.3.2 rgba(R,G,B,A)

A color value with red, green, blue, and alpha components can be described using rgba(R,G,B,A). The values for R, G, and B are the same as in 5.3.1 rgb(R,G,B). The value for A indicates the opacity of the color value. 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.

5.3.3 InverseQuant(P,S,QPD,IQF)

Reconstruction of a quantized value is described using InverseQuant(P,S,QPD,IQF). The reconstructed value RV is calculated as

$$RV = P + (1 - 2*S) * QPD * IQF$$

where P is the predicted value, S is the sign of the prediction difference, QPD is the quantized prediction difference, and IQF is the inverse quantization factor.

RV, P and IQF are floating point numbers. S and QPD are integers.

The document specifies the inverse quantization function that must be used but does not specify a quantization function.

For information only: a suitable quantization function could calculate S and QPD from an original value OV as

$$S = 1 \text{ if } P > OV \text{ and } S = 0 \text{ if } P <= OV.$$

$$QPD = (|OV - P| + 0.5) * (1.0 / IQF).$$

Other quantization functions could be used.

6 Acronyms and Abbreviations

Acronym	Description
CAD	Computer Aided or Assisted Design.
CIL	Common Intermediate Language.
CLI	Common Language Infrastructure.
CLOD	Continuous level of detail.
DID	Data packet element ID. A DID is a 128-bit value with the same structure as a GUID.
GUID	Globally Unique Identifier. A unique 128-bit number that is produced by the OS or by some applications to identify a particular component, application, file, database entry, and/or user.
LOD	Level of detail.
Ref	Reference.

7 General Description

The purpose of the Universal 3D file format is to provide a reliable, easy to use, easy to implement file format that supports the streaming, progressive transmission, of 3D mesh and level of detail information in a standard way. The format is for content creation tool developers who enable a variety of end user applications with 3D data.

This format is intended to further the proliferation and ubiquity of 3D data. The motivation for this file format is to address the growing need to reuse existing 3D data for applications and usage models downstream from the engineering or design uses for which the data was originally created.

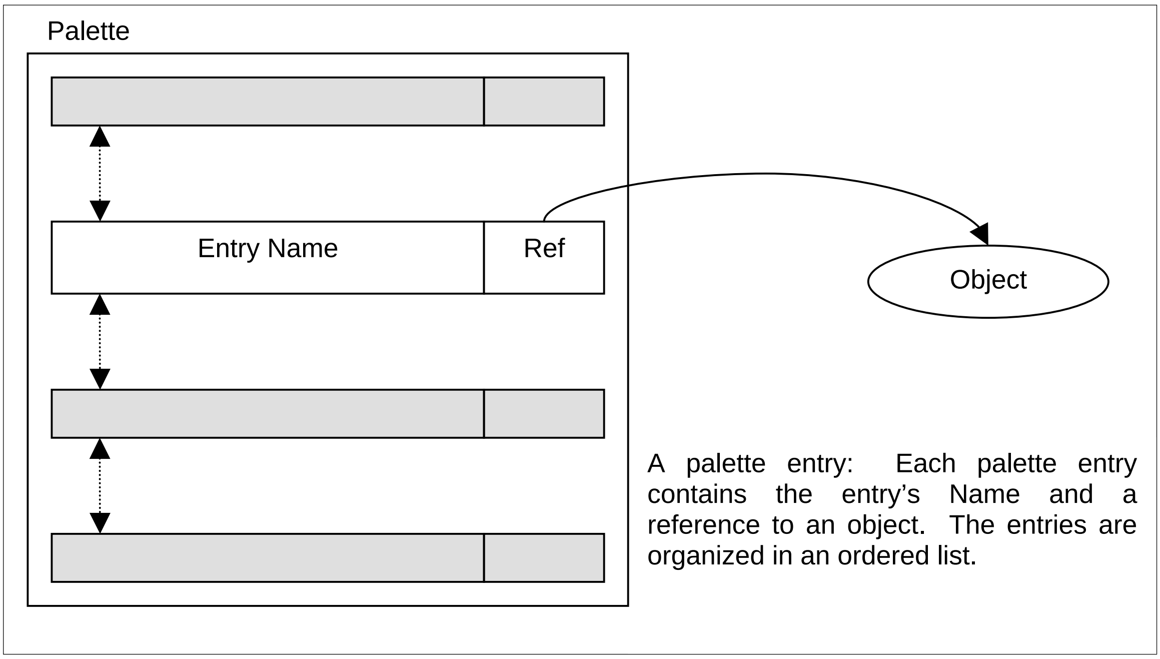
8 Architecture

This section describes the run-time architecture for the U3D file format. As the file format is a serialization of the run-time architecture, it is important to understand that architecture to fully understand the file format. The architecture here defines a foundation on which 3D applications can be built.

8.1 Execution architecture

The execution architecture is based on the interaction of several key elements: palettes, nodes, the scene graph, resources, and modifier chains. The palettes control access to nodes and resources. Nodes have spatial information and hierarchical relationships that build the scene graph. The nodes reference resources through the palettes. Together a node and a resource compose a 3D object. The resources contain the majority of the information to create an object while the light weight nodes are designed to take advantage of information sharing through references to the resources. Multiple nodes may use the same resource, so the nodes can be thought of as instances of the resources in the scene. Some types of resources and nodes are used as modifiers in modifier chains that manage the manipulation of data. The sections below give more detailed information about these elements and how they interact.

8.2 Palettes



A palette entry: Each palette entry contains the entry's Name and a reference to an object. The entries are organized in an ordered list.

Resources and nodes are accessed through palettes. The palettes are: model resource palette, light resource palette, view resource palette, texture resource palette, shader resource palette, material resource palette, motion resource palette, and node palette. The palettes are designed to control access to resources and nodes and to aid in information sharing.

A palette is organized as an ordered list of entries composed of an identifying name and a reference to an object or a reference to null. Entries can be accessed through the palette by specifying a name or by iterating through the list of entries contained in a palette.

A new palette entry is created by specifying a name and a reference to an object or a null reference. The palette entry is added to the list of entries in the palette. Because entries are identified by name, each name within a palette must be unique. If the name of a new palette entry is the same as an existing entry, the new entry replaces the existing entry. When an entry is deleted, it is removed from the list of entries.

To access an entry in a palette, a client object specifies an identifying name to a palette. If an entry with that name exists, the palette returns the object reference or null reference that is associated with the entry. Because client objects reference palette entries indirectly, an object may have the name for a palette entry that does not exist. If a client object specifies a name that is not in the palette, the client object is warned that the palette entry does not exist. The client object is responsible for correctly responding when the palette returns a null reference or the entry requested does not exist.

Objects can also register with the palette as an observer of a palette entry. The object is then notified by the palette when changes are made to the entry or the referenced object.

Each palette has one default entry associated with it. The default entries are identified by the empty string ("") and their properties are detailed below. The default entries may not be modified. The values of the properties of the default entries were chosen to provide reasonable or neutral behaviour for objects that use the default.

8.3 Node resources

The following subsections describe the resources that are referenced by nodes through the palettes. Each resource type has a corresponding node type. The resources contain the majority of the information needed to create a 3D object used when rendering. Nodes supply additional information that differentiates the 3D objects in the scene that share the same resource. The division of information between the resources and light weight nodes facilitates data sharing.

Each node contains hierarchical information about its parents and children and spatial information that is relative to that node's parents. The node types are: group, model, light, and view. The group node contains only the spatial and hierarchical information. More information on nodes can be found in 8.7 Scene graph.

A node is the first modifier in a node modifier chain. More information on modifier chains can be found in 8.6 Modifier chains.

The default node is a group node with no parents. The default node is located at the world origin (the identity transform).

There is no default model node, default light node, nor default view node.

8.3.1 Model resource

Model resources contain information used to create renderable geometry. The information includes geometry or a method to generate geometry and shading information that determines how the geometry is rendered. Model resources are also the first modifier in a model resource modifier chain. The output of a model resource modifier chain can then be used by model node modifier chains.

There is a preference for having the coordinate system oriented such that the Z-axis is in the up direction.

The default model resource is an empty CLOD mesh generator. This empty CLOD mesh generator has all count fields in the maximum mesh description set to zero.

8.3.2 Light resource

Light resources contain the information specific to lights. The supported types of lights are ambient, directional, point, and spot. The light resource defines the attributes associated with the lights including color and specularity. Point and spot lights also have an attenuation factor. Spot lights have an associated angle and decay rate. The light nodes provide spatial position and orientation and hierarchical information.

The default light resource is an ambient light that is enabled, no specularity, and color values rgb(0.75, 0.75, 0.75).

8.3.3 View resource

View resources have information regarding rendering, fog, and the portion of the scenegraph that is available.

View nodes provide spatial position and orientation and hierarchical information as well as the how the view is presented. Specifically, the node defines the clipping, projections (e.g. one, two or three point perspective or orthogonal), the view port, backdrops and overlays.

View resources are intended to contain information that is likely to be shared by instances of the view. View nodes are intended to contain information that is likely to be different for each instance of the view.

A scene graph can have several view nodes that define different viewpoints in the world. Although there is no default view node, there is a preference for having the coordinate system oriented such that the Z-axis is in the up direction with the Y-axis oriented in the direction of the view.

The default view resource has the following properties: pass count one, root node is the default node, and fog disabled.

8.4 Shading resources

The shading resources are used to determine the visual appearance of geometry when rendered. A list of shaders is applied to the geometry. Each shader refers to a number of textures and a material. A more detailed description of shading is given in 8.9 Rendering and Shading.

8.4.1 Texture resource

Texture resources are image data that may be applied to geometry when shading to modify its appearance. The texture resource contains image data and information about the size, method of compression, and color components of the image data.

The default texture resource is an 8x8 RGB 24bit image. The upper left and lower right 4x4 areas of the image have color values of rgba(1.0, 1.0, 1.0,1.0). The upper right and lower left 4x4 areas of the image have color values of rgba(1.0, 0.40, 0.20, 1.0).

8.4.2 Material resource

The material resource describes the appearance of a surface at the lowest level. The material describes which shading attributes are enabled and the colors associated with those attributes. The available attributes are: ambient color, diffuse color, specular color, emissive color, opacity and reflectivity.

The default material resource has an ambient color of rgb(0.75, 0.75, 0.75), an opacity of 1.0, and reflectivity of 0.0. Other colors associated with the default material are rgb(0.0, 0.0, 0.0).

8.4.3 Lit texture shader resource

The lit texture shader resource details which material and textures are used when rendering geometry and how the textures and material should be combined (i.e. blended). The lit texture shader resource accesses the material and textures through the palettes. The lit texture shader identifies the lighting properties used, the number of rendering passes, and the application of textures to geometry.

The default lit texture shader has lighting enabled, alpha test disabled, and does not use vertex color. Although not used, the alpha test reference value is 0.0, the alpha test function is ALWAYS, and the color blend function is FB_ALPHA_BLEND. The render pass enabled flags has a value of 0x00000001 indicating the default lit texture shader is only used in the first render pass.

The default lit texture shader uses the default material and no textures. Because no textures are used, the shader channels and alpha texture channels fields have no bits set.

8.5 Motion resource

The motion resource contains animation data. The data is stored in a number of tracks. Each track is composed of key frames with rotation, displacement and time information. A motion track can be used to animate the relative spatial information for a node or a bone in a bone hierarchy.

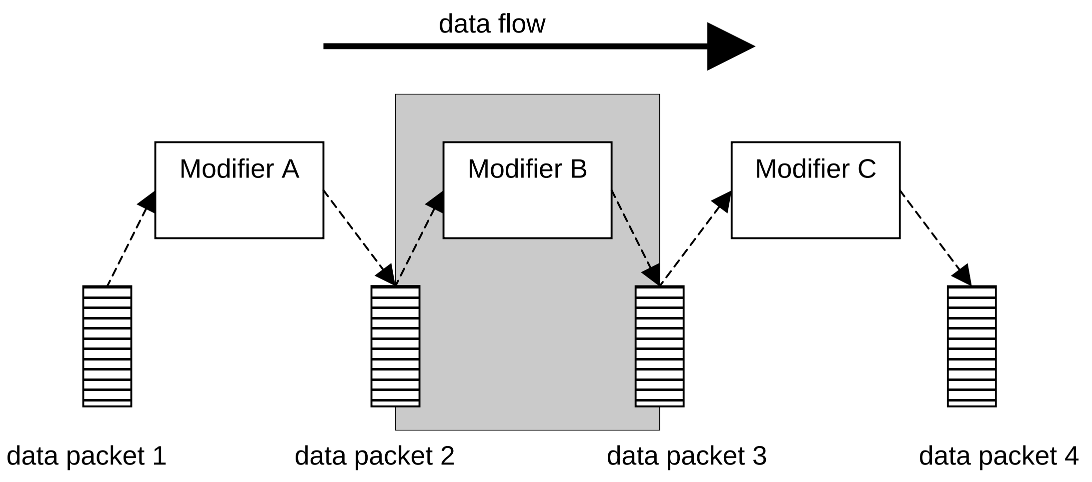
The default motion resource has zero motion tracks.

8.6 Modifier chains

Modifiers manipulate data associated with resources, nodes, and textures. A modifier receives as input an array of data elements called a data packet. Each element is identified by a data element id (DID) that is used to determine the type of data stored in the element. Each modifier defines a set of data elements as outputs and a set of dependencies for each of the outputs. The outputs may be dependent on data elements from the input data packet or data elements output by the modifier.

Examples of data elements are transforms, renderable groups, and simulation time.

The modifier chain object collects and orders the modifiers, ensures that the required inputs for each modifier are available, and maintains the dependency information associated with each data element. The modifier chain passes a data packet to a modifier and constructs a new data packet based on that modifier’s outputs and the previous data packet. The modifier chain will then present the data packet to the next modifier or as an output of the modifier chain if all of the modifiers have been evaluated.



Modifier chain operation: The modifier chain presents data packet 1 to modifier A. Modifier A creates outputs based on input data packet 1. The modifier chain uses those outputs and data packet 1 to create data packet 2 which it passes to modifier B. Modifier B creates outputs based on data packet 2. The modifier chain uses the outputs from modifier B and data packet 2 to create data packet 3 which it passes to Modifier C. Modifier C creates outputs based on data packet 3. The modifier chain uses those outputs to create data packet 4 which will be the output of the modifier chain.

When creating a new data packet, the modifier chain adds the outputs of a modifier to the data packet. The modifier chain consumes a data element in the input data packet if that data element has the same DID as an output data element. Each of the remaining data elements from the input data packet is copied to the new data packet unless the input element depends, directly or indirectly, on one of the elements consumed by the modifier chain. Finally, the output data elements of the modifier are added to the new data packet.

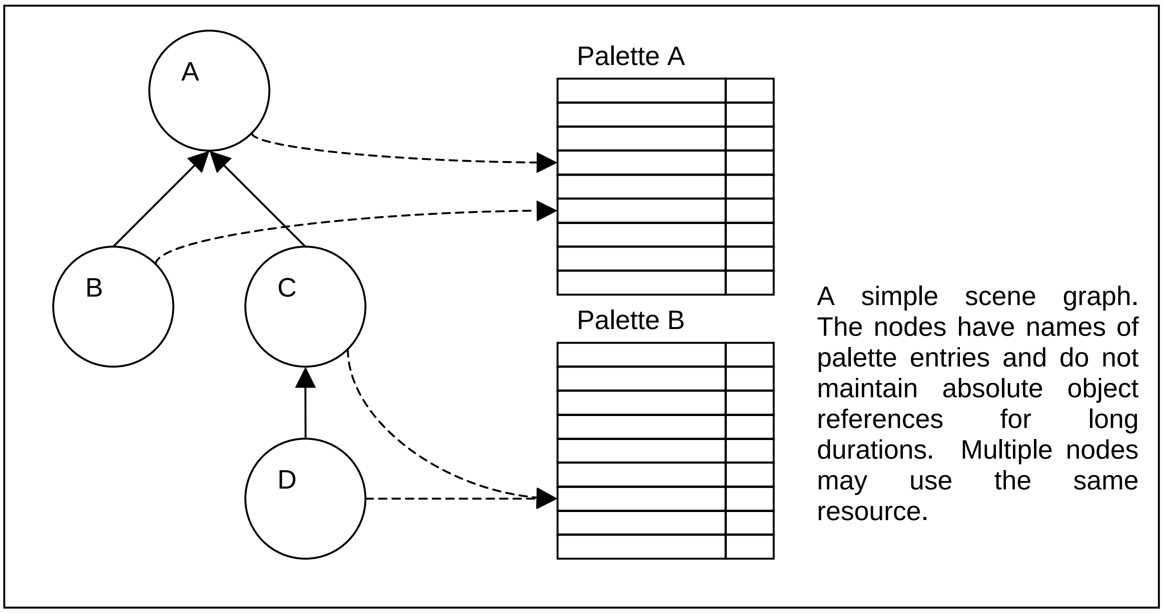
The modifier chain object allows the lazy evaluation of the modifiers. Data elements may be cached and the modifier chain may then evaluate the dependency information when data elements are modified outside of the modifiers in the chain to determine which, if any, of the modifiers must be re-evaluated to update the modifier chain object's output.

DID Name	DID Description
Bone weights	Weighting factors that associate vertices with bones
Renderable Group	A group of renderable elements
Renderable Group Bounds	A bounding structure for a Renderable Group
Simulation Time	Time value used for animation and simulation
Skeleton	Bone structure used in animation and inverse kinematics
Transform Set	Set of transforms to place objects in space
View Frustum	The volume of space visible to the view
View Size	The dimensions of the view port
View Transform	Transform that places the view in space

8.7 Scene graph

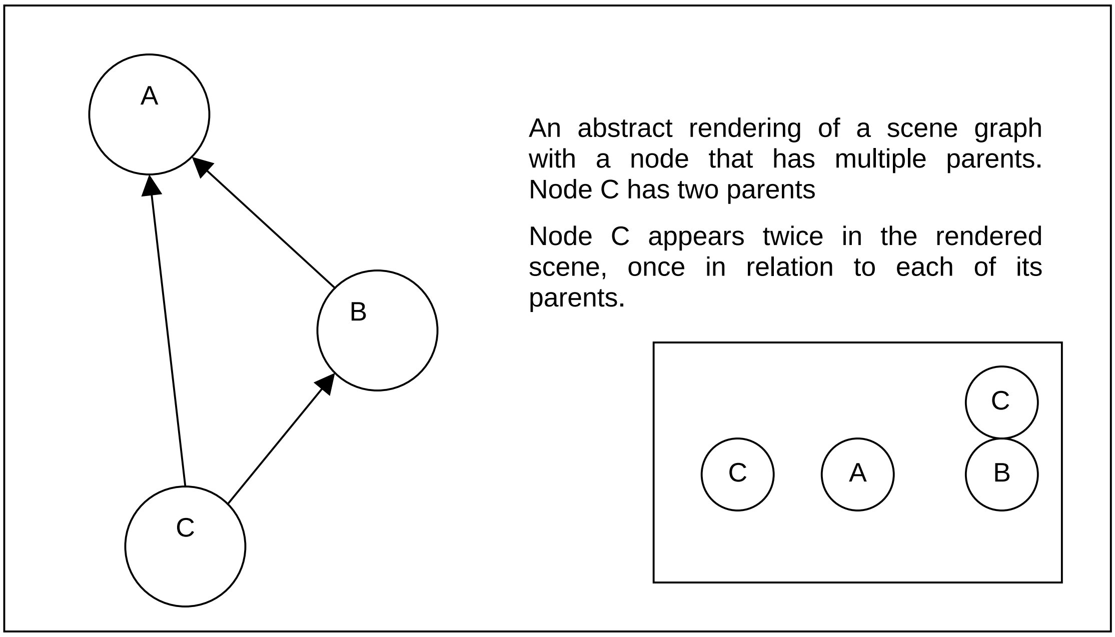
The scene graph maintains the hierarchical and spatial relationships between nodes. Each node may have zero or more children and zero or more parents. Each node contains information about its parents, and its position relative to each parent. The relationship information in the nodes is used to build the scene graph.

Group nodes are the most basic node type. They contain positional and relational information that is used to place them in the scene graph. Group nodes are useful for collecting other related nodes together. The default node in the node palette is a group node identified by the empty string (""). The default node is referred to as *the world*. The default node's transform is always the identity transform.



A simple scene graph. The nodes have names of palette entries and do not maintain absolute object references for long durations. Multiple nodes may use the same resource.

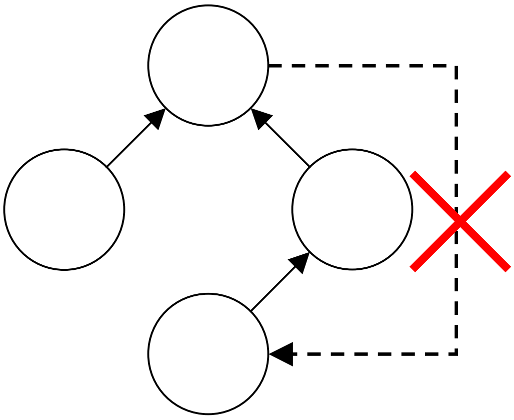
Nodes have a transform for each parent that specifies its position relative to that parent. The node's spatial position is defined by the node's parent's transform multiplied by the node's transform for that parent. The node will appear once relative to each appearance of each parent. In this way, one node may appear many times in a scene. A node with no parents will not appear in the scene. To appear in the scene, a node must be a descendent of *the world*.



An abstract rendering of a scene graph with a node that has multiple parents. Node C has two parents

Node C appears twice in the rendered scene, once in relation to each of its parents.

Because a node's transforms are determined based on its parents' transforms, all relationships in the scene graph must be acyclic. A node's transforms can not be evaluated relative to its own transforms.



An illegal scene graph. Cyclical relationships make evaluation of the transforms impossible.

8.8 CLOD mesh generator

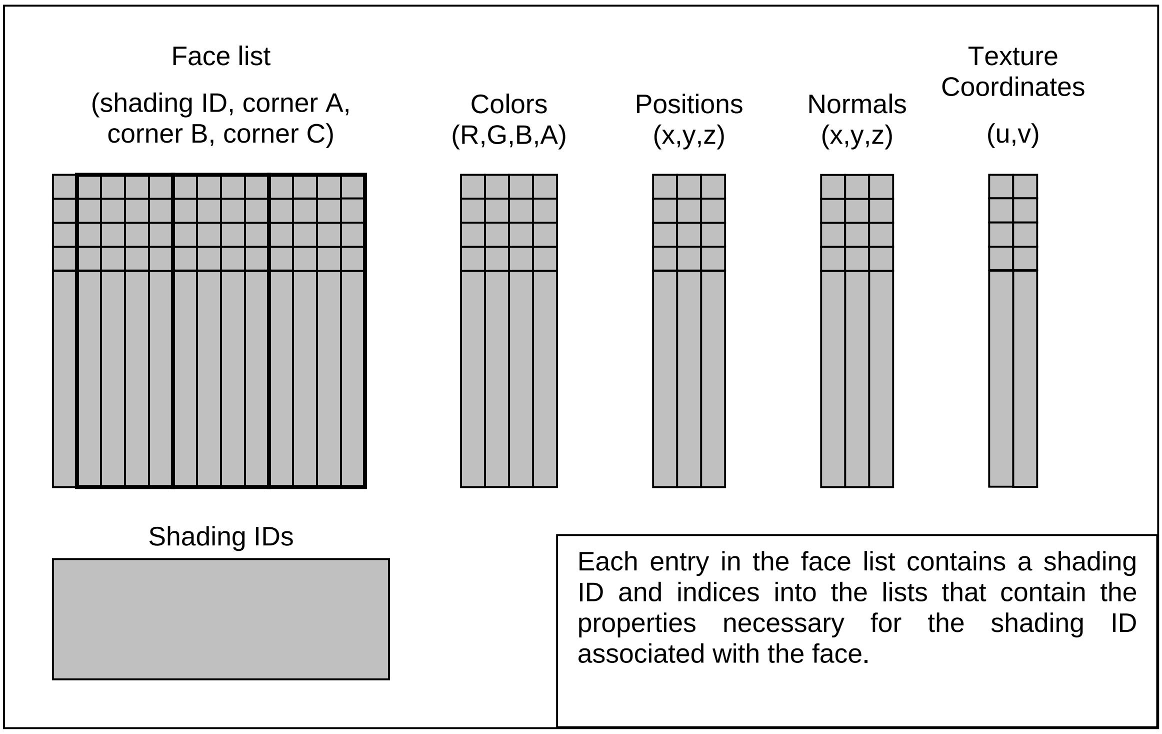
The continuous level of detail (CLOD) mesh generator is a modifier that creates one or more CLOD triangle meshes. The CLOD meshes contain information in addition to the geometry that describes how to add or remove geometry from the mesh. The level of detail, or resolution, of the mesh is the amount of the total available geometry that is actually used.

The CLOD mesh generator requires as input an author mesh and author mesh resolution updates and creates a set of renderable meshes, a set of renderable mesh updates for each mesh created, and a CLOD controller to manage the level of detail of the renderable meshes.

More details can be found in 9.6.1 CLOD Mesh Generator.

8.8.1 Author mesh

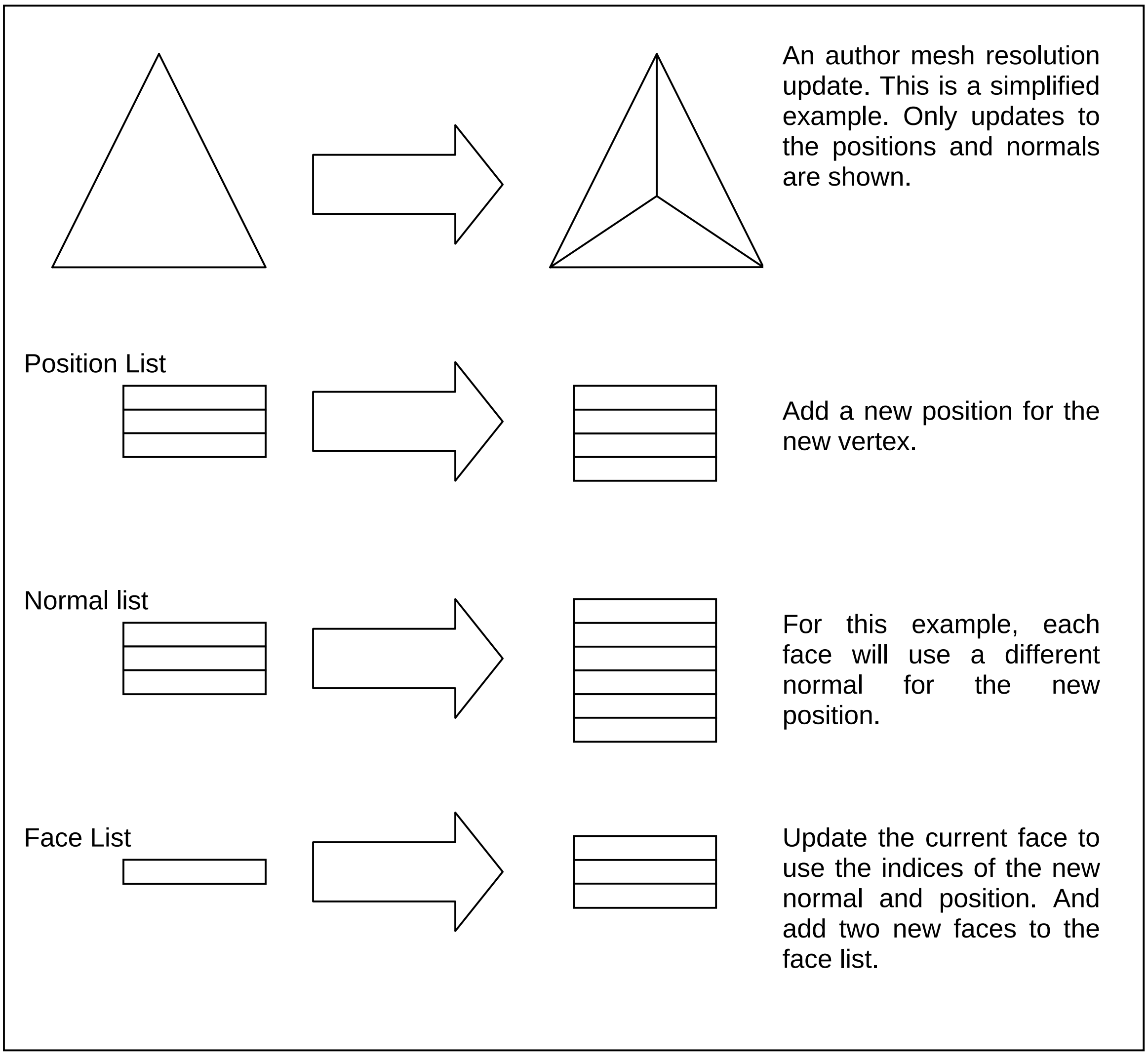
The author mesh structure is designed to be easy to modify and compress. The author mesh has an associated group of shading IDs, lists of attribute values that will be associated with the vertices of the mesh’s triangles (e.g. texture coordinates, position vectors, colors, normal vectors), and a list of faces that specify the shading ID associated with each face and the attributes associated with that face’s vertices. The shading IDs identify which shaders are used for a face. The shading IDs dictate which attributes must be specified for each triangle corner. For example, if a shading ID specifies 2 texture layers, 2 sets of texture coordinates are required for each corner of the triangle.



Each entry in the face list contains a shading ID and indices into the lists that contain the properties necessary for the shading ID associated with the face.

8.8.2 Author mesh resolution updates

The author mesh resolution updates describe how to change the resolution of an author mesh. A single update changes the number of vertex positions in the mesh by one. To increase the resolution, the update contains the new entries for each attribute, the new faces, and updates to existing faces. To decrease the resolution, the same information is used to remove the new faces and undo the updates to the previously existing faces. The Author mesh resolution update always increases the number of vertex positions by one. The Shading ID associated with a face is constant through resolution updates. The other properties associated with a face may also change. For example, the face’s corners may be associated with new vertex positions, normals, colors, or texture coordinates.



An author mesh resolution update. This is a simplified example. Only updates to the positions and normals are shown.

Add a new position for the new vertex.

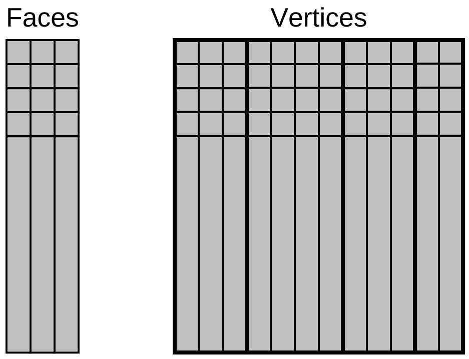
For this example, each face will use a different normal for the new position.

Update the current face to use the indices of the new normal and position. And add two new faces to the face list.

8.8.3 Renderable mesh

Renderable meshes regroup the data in an author mesh so that it is optimized for rendering. The CLOD mesh generator creates a group of renderable meshes and resolution updates for those meshes from an author mesh and resolution updates for that mesh. While an author mesh may have more than one shading ID associated with its faces, a single renderable mesh may have only one shading ID. The CLOD mesh generator will create a renderable mesh and updates for each shading ID used by the author mesh.

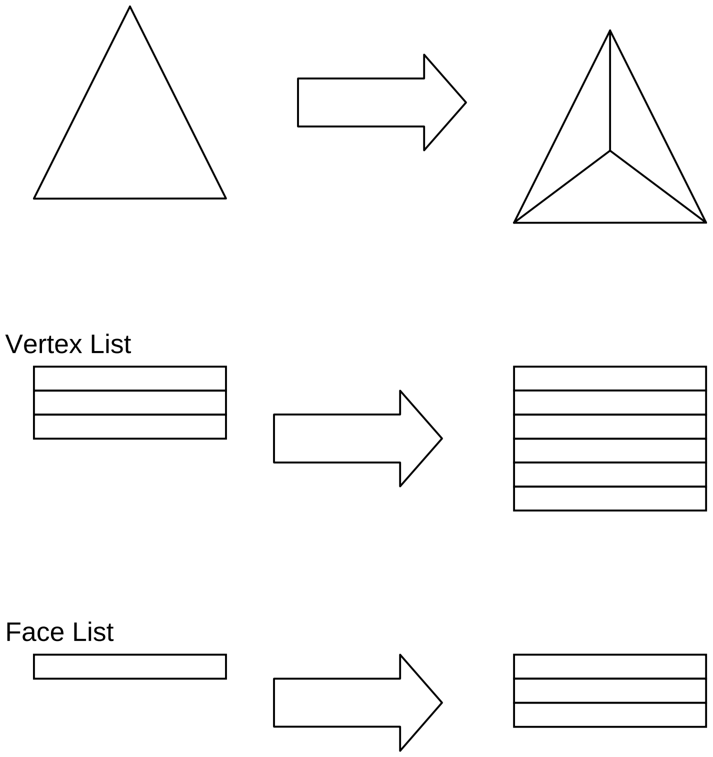
The renderable mesh consists of a list of vertices and a list of faces. Each vertex has a complete set of per vertex attributes as specified in the author mesh. The faces contain indices into the vertex list for each vertex.



The faces of the mesh index in to the vertex list. Each entry in the vertex list contains all of the properties necessary for the Shading ID associated with the renderable mesh.

8.8.4 Renderable mesh resolution updates

The renderable mesh resolution updates describe how the renderable mesh is modified to increase or decrease the level of resolution in a render mesh. The process is similar to the author mesh resolution updates described above. The update adds a new vertex position. Because the vertex position is not unique for each entry in the vertex list, multiple new entries may be created in the vertex list for the update. To increase the resolution, the update specifies the new vertices to add to the vertex list, the new faces to add to the face list, and the updates to existing faces. To decrease the resolution, the same information is used to remove the new faces and undo the changes to the previously existing faces.



Render mesh resolution update.

Add new entries to the vertex list to add the new position and normals. A new entry is needed for each combination.

Update the existing face to use one of the new vertices. Add two new faces.

8.8.5 CLOD modifier

The CLOD mesh generator may convert an author mesh into more than one renderable mesh. The CLOD modifier manages the renderable mesh resolution updates to maintain a visually consistent level of detail across renderable meshes generated from a common author mesh.

The CLOD controller translates the desired level of detail of an author mesh into levels of detail appropriate for each renderable mesh.

8.9 Rendering and Shading

Each of the nodes in the node palette is the first modifier in an instance modifier chain. The final data packet in the modifier chain is accessible to clients of the run-time system. This section describes some of the data elements in the final data packet and how those data elements may be used to draw renderable elements.

These data elements can be found in the final data packet of the node modifier chain. Interfaces supported by these data elements will be specified in a future edition of this standard.

Details of rendering systems and culling systems are outside the scope of this specification.

8.9.1 Transform Set

The transform set data element is present for all node types.

The transform set is a set of transformations from local coordinate space to global coordinate space. These transformations include the effects of all parent-child relationships and animation modifiers. The transform set for a child node will have one transform for each transform in the transform sets of its parents. A node with no parents will have an empty transform set.

A node will appear once in the world for each transform in the transform set. If the transform set is empty, the node is disconnected from the world and does not appear in the rendered view.

8.9.2 Renderable Group

The renderable group data element is present for model nodes. The renderable group may also be present for new node types added to the system through the extensibility mechanism.

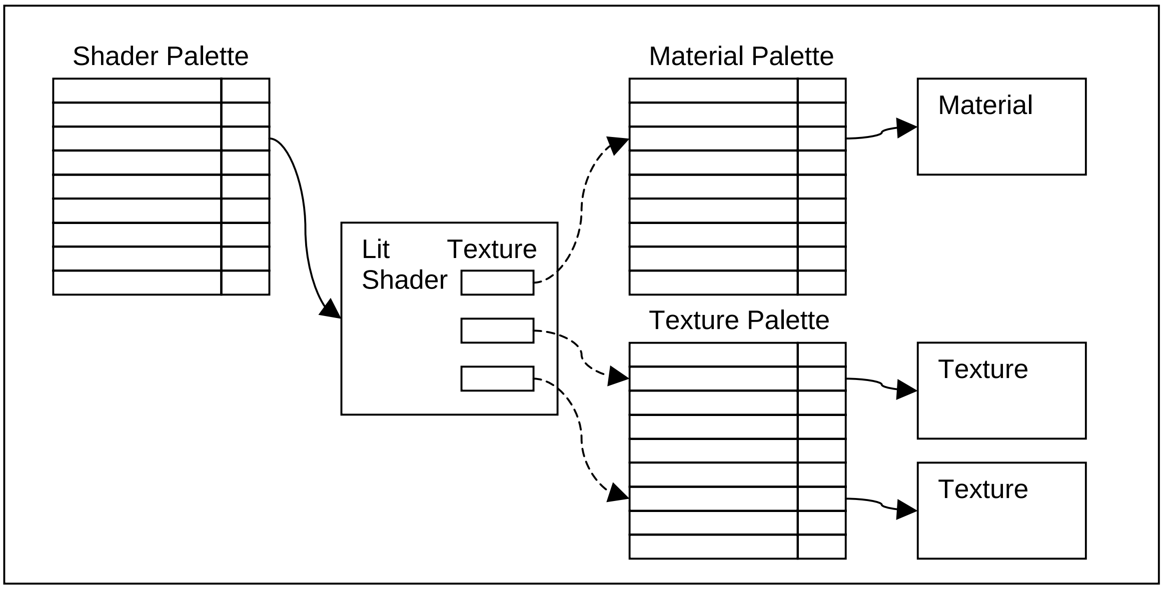
The renderable group holds a renderable element group and an associated shading group. There are three types of renderable element groups: renderable mesh group, renderable line group, and renderable point group.

A data packet may contain more than one renderable group. Different types of renderable elements are not mixed in the same renderable group.

8.9.2.1 Shader

A shader contains information needed to determine the appearance of a surface during rendering. This edition specifies one shader called the Lit Texture Shader. Future editions may specify additional types of shaders.

The Lit Texture Shader includes references to Material Resources and Texture Resources. The Lit Texture Shader also indicates how to combine those resources when rendering.



8.9.2.2 Shading Group

The shading group holds a set of shader lists. There is one shader list for each renderable element in the renderable element group. The shader list is a list of shaders that should be used to draw the renderable element. The shader list contains names of shaders in the shader resource palette. The shader list is ordered and the shaders are used in order.

8.9.2.3 Renderable Mesh Group

The renderable mesh group contains a set of renderable meshes. Each renderable mesh is associated with a different shader list in the shader group. The renderable mesh consists of a vertex array and a face array. Each vertex in the vertex array contains all of the per vertex attributes (such as position, normal, texture coordinates, etc.) for that vertex. Each face in the face array contains three indices into the vertex array; one index for each corner of the face. All faces in the renderable mesh are triangles. Each face in the face array is drawn according to the parameters of the shaders in the shader list.

8.9.2.4 Renderable Line Group

The renderable line group contains a set of renderable line sets. Each renderable line set is associated with a different shader list in the shader group. The renderable line set consists of a vertex array and a line array. Each vertex in the vertex array contains all of the per vertex attributes (such as position, normal, texture coordinates, etc.) for that vertex. Each line in the line array contains two indices into the vertex array; one index for each end of the line segment. Each line in the line array is drawn according to the parameters of the shaders in the shader list.

8.9.2.5 Renderable Point Group

The renderable point group contains a set of renderable point sets. Each renderable point set is associated with a different shader list in the shader group. The renderable point set consists of a vertex array. Each vertex in the vertex array contains all of the per vertex attributes (such as position, normal, texture coordinates, etc.) for that vertex. Each vertex in the vertex array is drawn according to the parameters of the shaders in the shader list.

8.9.3 Renderable Bound

The renderable bound data element is present for model nodes. The renderable bound may also be present for new node types added to the system through the extensibility mechanism.

The renderable bound may be either a bounding sphere or a bounding box. The choice of which type of renderable bound to support is up to the implementation of the various nodes and modifiers.

The final data packet shall contain either one renderable bounding sphere or one renderable bounding box or both.

The renderable bound encompasses all renderable elements of the renderable geometry group. The bounding sphere or box does not need to be a tightest bounding sphere or box.

If the data packet contains more than one renderable group, the renderable bound shall encompass all the renderable groups.

The renderable bound can be used by a culling system to determine which nodes may have visual impact on a particular frame rendering.

The renderable bound is described in local coordinate space. The transforms in the transform set must be applied to the renderable bound for use by the culling system.

To make full use of the modifier chain's lazy evaluation feature, modifiers should avoid making the renderable bound depend on the renderable group.

8.10 Serialization

This section describes how the objects stored in the various palettes are serialized. Clause 9 provides details on the formatting of particular objects. Clause 10 contains additional details on compression requirements.

8.10.1 Object serialization

Each object is serialized as a sequence of one or more blocks. The first block is called the declaration block. Any subsequent blocks are called continuation blocks. The declaration block contains enough information to create the object and place it in the correct palette location. For modifiers, the declaration block also indicates placement within the modifier chain. Most types of objects have only the declaration block. Objects which require a large amount of information use continuation blocks to carry most of the data.

Each block is assigned a priority number. The priority number is used for sequencing the blocks and for interleaving the blocks from multiple objects. Declaration blocks have a priority number of zero. The priority number increases for each continuation block; the amount of increase must be greater than zero. The maximum priority number is 0x7FFFFFFF.

8.10.2 File structure

A file is structured as a sequence of blocks. Blocks with lower priority numbers precede blocks with higher priority numbers. The first block is the File Header Block. The File Header Block is the only required block for a U3D file. The File Header Block is followed by declaration blocks. Continuation blocks may follow the declaration blocks. 9.1 contains more details on sequencing of blocks.

8.10.3 Block structure

Each block contains size fields so that the loader can determine the end of a block if the data in that block is not required or if a decoder for that block type is unavailable.

Each block has a data section and a meta data section. The format of the data section will vary based on the type of the object. The interpretation of the data section is specified in Clause 9 of this specification. The format of the meta data section is always a sequence of Key/Value pairs. Although the format of the meta data is defined in 9.2.6, the interpretation of the content of the Key/Value pairs is outside the scope of this specification.

Each block contains a block type field to identify the formatting of the data section.

8.10.4 File reference

A U3D file can reference other U3D files using a File Reference block. When the referencing file is loaded, the referenced files are also loaded. Using this mechanism, a large file can be partitioned into several smaller files. 9.4.2 File Reference contains details.

8.10.5 Declaration block section

The declaration block section contains the information necessary to create all of the objects in the file. When a loader has completed processing the declaration section, all of the object in the file have been created and added to the appropriate palettes.

Before processing the declaration section can be considered complete, the processing of declaration sections of any referenced files must also be complete.

Details on when rendering may begin are outside the scope of the spec and are left to an implementation.

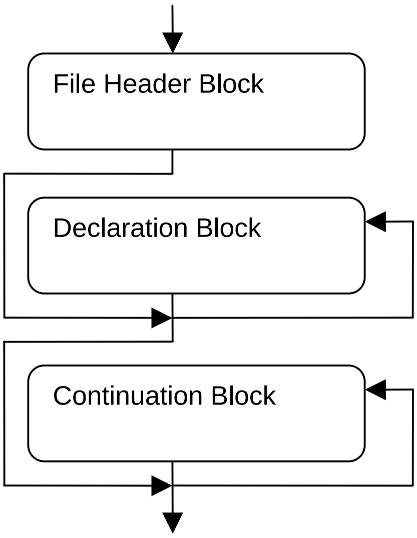
8.11 Extensibility

NOTE
Extensibility feature will not be addressed in the first edition of this Standard. It is intended to be added as a primary feature for the next edition of this Standard.

9 File Format

9.1 File structure

A file is structured as a sequence of blocks. The first block is the File Header Block. The File Header Block is followed by declaration blocks. Continuation blocks may follow the declaration blocks. Each block contains size fields so that the loader can determine the end of a block if the data in that block is not required or if a decoder for that block type is unavailable.



9.1.1 File Header Block

The File Header Block contains information about the file. The loader uses the File Header Block to determine how to read the file.

9.1.2 Declaration Block

Declaration Blocks contain information about the objects in the file. All objects must be defined in a Declaration Block. The File Header Block is considered to be a Declaration Block.

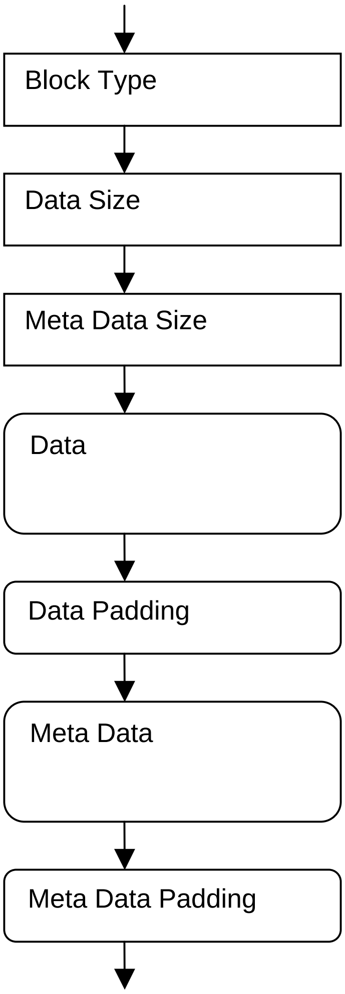
9.1.3 Continuation Block

The Continuation Blocks can provide additional information for objects declared in a Declaration Block. Each Continuation Block must be associated with a Declaration Block.

9.2 Block structure

All block types have the same basic structure. The Block Type, Data Size, and Meta Data Size determine how the remainder of the block is interpreted by the loader. Data Padding and Meta

Data Padding fields are used to keep 32-bit alignment relative to the start of the File Header Block. The start of the Block Type field, Data section and Meta Data section are all 32-bit aligned.



9.2.1 U32: Block Type

Block Type identifies the type of object associated with this block. The interpretation of the data section of this block depends on the Block Type. This specification defines valid block type values for the base profile. The New Object Type block may be used to define additional valid block type values for the extensible profile. Block type values other than those defined for the base profile or defined through a New Object Type block shall not be used.

9.2.2 U32: Data Size

Data Size is the size of the Data section in bytes. Data Size does not include the size of the Data Padding.

9.2.3 U32: Meta Data Size

Meta Data Size is the size of the Meta Data section in bytes. Meta Data Size does not include the size of the Meta Data Padding.

9.2.4 Data

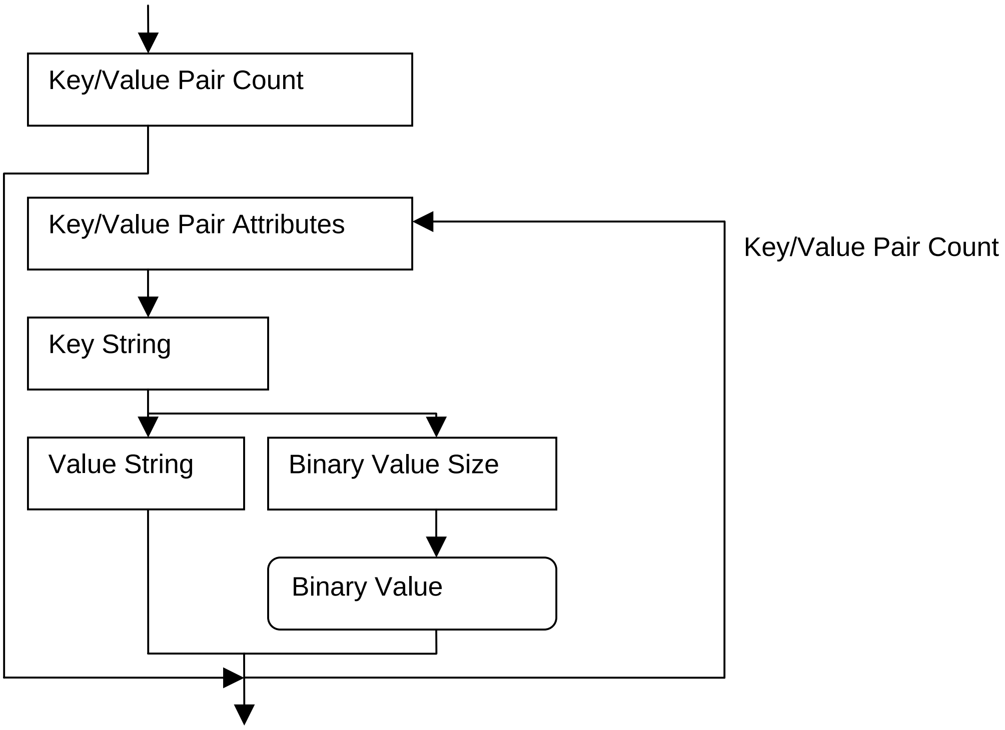
Data Size bytes of data. The interpretation of the Data section depends on the Block Type.

9.2.5 variable: Data Padding

Data Padding is a variable size field. Zero to three bytes are inserted to maintain 32-bit alignment for the start of the Meta Data section. The value of the padding bytes is 0x00.

9.2.6 Meta Data

Meta Data is Meta Data Size bytes of data. The Meta Data section contains a sequence of Key/Value pairs. The interpretation of the content of the Key/Value pairs is outside the scope of this specification.



9.2.6.1 U32: Key/Value Pair Count

Key/Value Pair Count is the number of Key/Value pairs in this Meta Data section.

9.2.6.2 U32: Key/Value Pair Attributes:

The Key/Value Pair Attributes indicate formatting options for the Key/Value pair. The following attribute values can be OR'd together:

0x00000000 - indicates the Value is formatted as a String

0x00000001 - indicates the Value is formatted as a binary sequence.

0x00000002 - indicates the Value is HIDDEN and should not be displayed by the viewer.

0x00000010 - indicates that this meta data should be used when double-clicked.

0x00000020 - indicates the Value should be displayed by the viewer in a right-click menu.

0x00000040 - indicates the Key should be displayed by the viewer in a right-click menu.

0x00000100 - indicates the Value is an ACTION and should executed by the viewer.

0x00000200 - indicates the Value is a FILE and should opened by the viewer.

0x00000400 - indicates the Value is MIME DATA and should opened by the viewer.

All other attribute values are reserved.

The attributes must indicate whether the value is formatted as a string or as binary. Note that this indication is determined from a single bit that may be set to zero or one. All other attributes are optional.

String values are subject to possible translation depending on the client platform. Binary values are not subject to any form of string translation.

Additional attributes of the key/value pair may be defined by including an Attribute String in the Key String. Multiple Attribute Strings can appear in a given Key String.

An Attribute String shall have the form #myAttribute or #myAttribute=myValue. The myAttribute string may not contain any whitespace, "=" or "#" characters. The attribute starts with the first character after the initial "#" character, and ends with the last character before a whitespace, "#", or "=" character, or the end of the Key String.

When an attribute value is specified, the value starts with the first character after the "=" character, and ends with the last character before a whitespace or "#" character, or the end of the Key String. If a value contains whitespace or "#" characters, it should be enclosed in quotes ("). When a value string contains a quote, replace the quote with two consecutive quotes. The quotes that delimit the value are not considered part of the value definition.

Examples:

#bold#italic	- defines two attributes bold and italic
#bold #italic	- also defines two attributes bold and italic
#height=7mm	- defines a height attribute with a value of 7mm
#height="7 mm"	- defines a height attribute with a value of 7 mm
#index="#7"	- defines an index attribute with a value of #7

#warning="Never yell ""Fire!"" in a crowded theater" - defines a warning attribute containing the quoted string "Fire!". The actual value of the attribute is Never yell "Fire!" in a crowded theater.

9.2.6.3 String: Key String

A String representing the key used to look up a value.

9.2.6.4 String: Value String

A String representing the value associated with a key.

9.2.6.5 U32: Binary Value Size

The size of the data that is associated with a key.

9.2.6.6 Binary Value

Binary Value is Binary Value Size bytes of data that is associated with a key.

9.2.7 variable: Meta Data Padding

Meta Data Padding is a variable size field. Zero to three bytes are inserted to maintain 32-bit alignment for the start of the next block. The value of any padding bytes is 0x00.

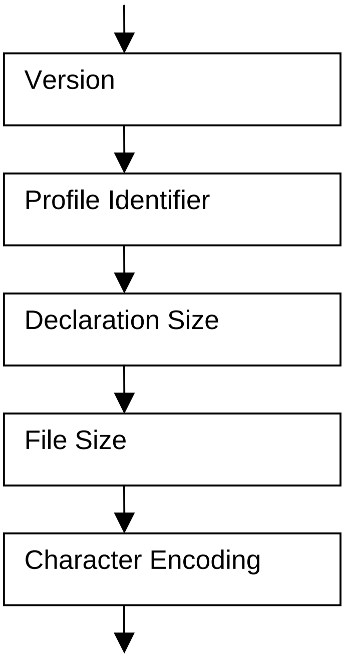
9.3 Block definitions

All blocks are considered declaration blocks unless stated otherwise. Blocks may contain names that reference objects that have not been defined. When loading a file, these names will be accepted and it should be assumed that the needed objects will be loaded or created at some future time. Object implementations shall use fallback values for references to undefined objects. Definition of fallback values is implementation dependent.

9.4 File structure blocks

9.4.1 File Header (blocktype: 0x00443355)

The File Header is the only required block in a file. It contains information about the rest of the file. The File Header block is considered a declaration block. The Priority Update Block is the continuation block type associated with the File Header.



9.4.1.1 I32: Version

Version is the version of the file format used to write this file. The current version number is zero. Until compliance has been validated for an encoder, all files created by that encoder shall use a version number less than zero.

9.4.1.2 U32: Profile Identifier

Profile Identifier is used to identify optional features used by this file. Valid values are

0x00000000 – Base profile; no option features used

0x00000002 – Extensible profile; uses extensibility features

0x00000004 – No compression mode

Profile bits may be combined using the OR operator.

The Extensible profile bit indicates this file may contain New Object Type blocks and other blocks defined for extensibility. A loader that does not support the extensible profile is not required to process those blocks or any blocks in a file with the extensibility profile bit set. It is recommended that such a loader make a best effort to load those portions of the file that it can load.

The no compression mode bit indicates this file does not contain any compressed values. Where the file format syntax defined in Clause 9 calls for a compressed value, the corresponding uncompressed value is used in a file with the no compression mode bit set. For example, a compressed U16 will be replaced with a U16. All readers shall support both the default compressed mode and the no compression mode of operation. The setting of the no compression mode bit applies to the entire U3D file but does not apply to other files referenced by the U3D file.

9.4.1.3 U32: Declaration Size

Declaration Size is the number of bytes in the Declaration Block section of the file. Declaration Size includes the size of the File Header block and all declaration blocks including any padding bytes in those blocks.

9.4.1.4 U64: File Size

File Size is the number of bytes in this file. File Size includes the size of all blocks including the File Header block and any padding bytes in those blocks. File Size does not include the size of any external files referenced by the contents of any block.

9.4.1.5 U32: Character Encoding

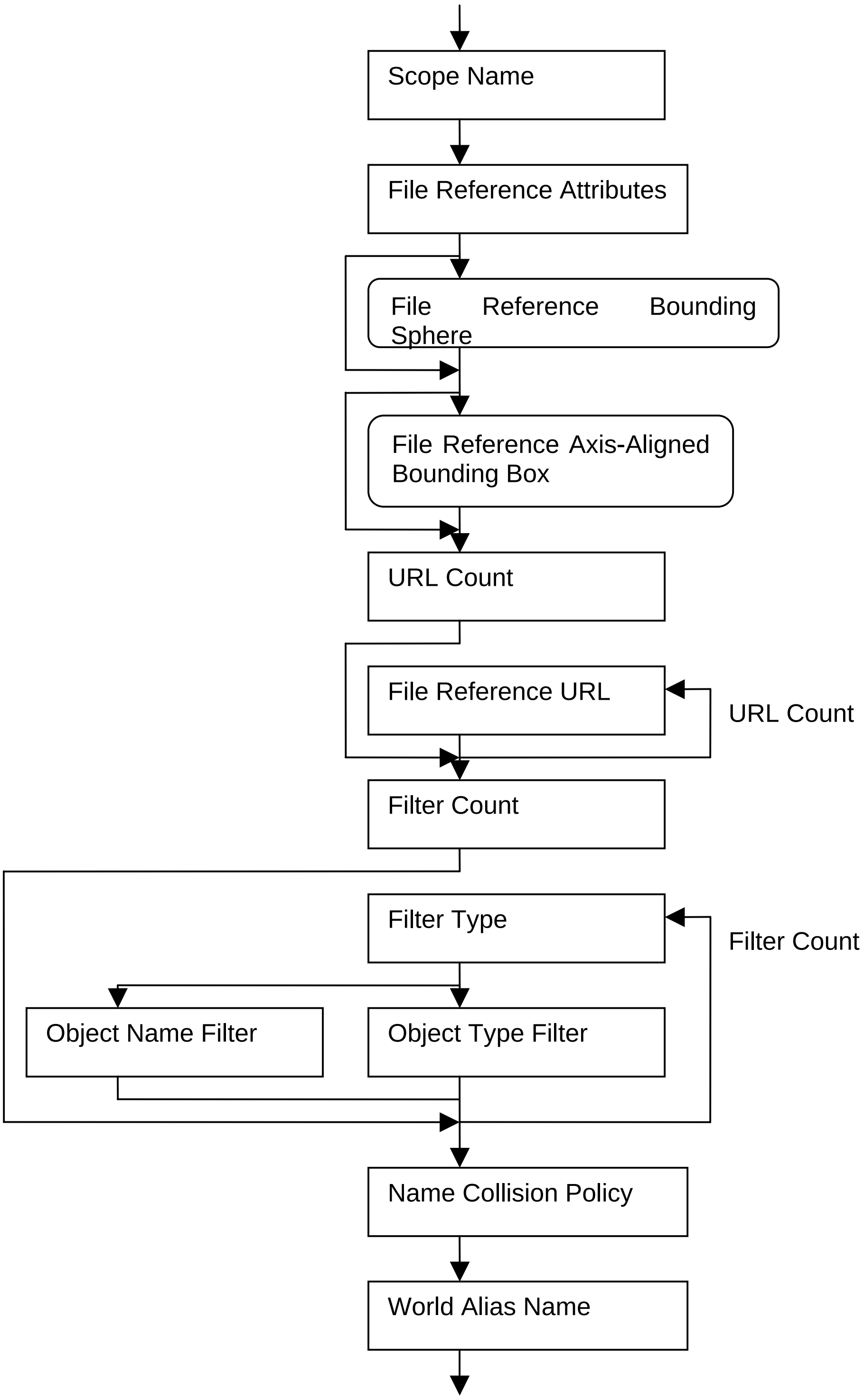
Character Encoding is the encoding used for strings in this file. The Internet Assigned Numbers Authority (IANA) website at http://www.iana.org/assignments/character-sets contains the assignment of MIB enum values to various character set encodings. Character Encoding can be used for translation of strings for a client application.

For the current version of U3D, the Character Encoding shall be UTF-8. UTF-8 corresponds to a MIB enum value of 106.

9.4.2 File Reference (blocktype: 0xFFFFFF12)

A File Reference block contains information for finding a single file that is associated with this file and is loaded with it. Multiple locations for the file may be specified. The File Reference block may also contain filters that load a portion of the file based on name or object type.

An implementation could keep track of which external file reference was used to load which objects but is not required to do so.



9.4.2.1 String: Scope Name

Scope Name is used to identify the external file reference. Depending on the collision policy, the scope name may be used to modify the names of objects in the referenced file.

9.4.2.2 U32: File Reference Attributes

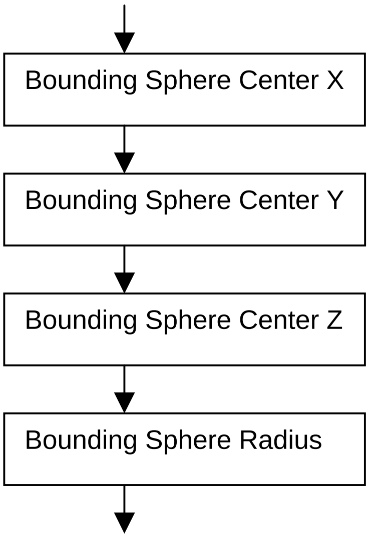
File Reference Attributes is a bitfield indicating the presence of optional information about the external file. The bounding information is optional. All other values are reserved.

0x00000001 – Bounding sphere information present

0x00000002 – Axis-aligned bounding box present

9.4.2.3 File Reference Bounding Sphere

The bounding sphere should contain all of the geometry expected to be produced by the modifier chains in the external file. This bounding sphere is an initial estimate of the extent of the geometry and may be updated by the run-time after loading. The bounding sphere information in this block may be used to determine whether to load the external file.



9.4.2.3.1 F32: Bounding Sphere Center X

Bounding Sphere Center X is the X coordinate of the center of the bounding sphere.

9.4.2.3.2 F32: Bounding Sphere Center Y

Bounding Sphere Center Y is the Y coordinate of the center of the bounding sphere.

9.4.2.3.3 F32: Bounding Sphere Center Z

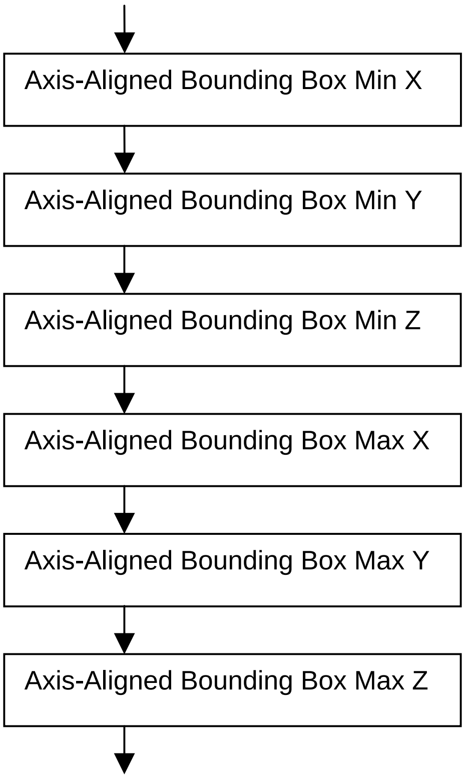
Bounding Sphere Center Z is the Z coordinate of the center of the bounding sphere.

9.4.2.3.4 F32: Bounding Sphere Radius

Bounding Sphere Radius is the radius of the bounding sphere.

9.4.2.4 File Reference Axis-Aligned Bounding Box

The axis-aligned bounding box should contain all of the geometry expected to be produced by the modifier chains in the external file. This axis-aligned bounding box is an initial estimate of the extent of the geometry and may be updated by the run-time after loading. The axis-aligned bounding box information in this block may be used to determine whether to load the external file.


Axis-Aligned Bounding Box Min X
Axis-Aligned Bounding Box Min Y
Axis-Aligned Bounding Box Min Z
Axis-Aligned Bounding Box Max X
Axis-Aligned Bounding Box Max Y
Axis-Aligned Bounding Box Max Z

9.4.2.4.1 F32: Axis-Aligned Bounding Box Min X

X coordinate of the bounding box minimum corner

9.4.2.4.2 F32: Axis-Aligned Bounding Box Min Y

Y coordinate of the bounding box minimum corner

9.4.2.4.3 F32: Axis-Aligned Bounding Box Min Z

Z coordinate of the bounding box minimum corner

9.4.2.4.4 F32: Axis-Aligned Bounding Box Max X

X coordinate of the bounding box maximum corner

9.4.2.4.5 F32: Axis-Aligned Bounding Box Max Y

Y coordinate of the bounding box maximum corner

9.4.2.4.6 F32: Axis-Aligned Bounding Box Max Z

Z coordinate of the bounding box maximum corner

9.4.2.5 U32: URL Count

URL Count is the number of URL strings that follow.

9.4.2.6 String: File Reference URL

File Reference URL is a String identifying the external file location. Multiple locations can be specified for the external file. The loader shall load the file from one of the locations. HTTP and FTP protocols will be recognized with absolute and relative addressing.

9.4.2.7 U32: Filter Count

Filter Count is the number of filters to apply when loading the referenced file. If the filter count is zero, then all objects from the referenced file are loaded. If the filter count is greater than zero, then objects from the referenced file shall only be loaded if they match the specification of at least one of the filters. A modifier object shall be loaded if and only if the object it modifies is loaded.

9.4.2.8 U8: Filter Type

Filter Type is the type of the filter.

0x00 – Object Name Filter

0x01 – Object Type Filter

9.4.2.9 String: Object Name Filter

Object Name Filter is a string used to filter objects by name. An object shall be loaded if its name matches Object Name Filter.

The Object Name Filter may contain the wildcard characters question mark ‘?’ and asterisk ‘*’. The question mark wildcard matches any one character at that position. The asterisk wildcard matches any zero or more characters at that position. The numerical value and size of the wildcard characters is dependent on the Character Encoding defined in the File Header block.

9.4.2.10 U32: Object Type Filter

Object Type Filter is used to filter objects by type. An object shall be loaded if the block type of its declaration block matches Object Type Filter.

9.4.2.11 U8: Name Collision Policy

A name collision occurs when the file being loaded contains an object with the same name as an object that already exists either loaded previously or created programmatically. Name Collision Policy indicates how name collisions are to be handled. Valid values are:

0x00 – Replace existing object with the new object from external file.

0x01 – Discard the new object from external file.

0x02 – Prepend scope name to object name for all objects from the external file

0x03 – Prepend scope name to new object name if there is a collision.

0x04 – Append instance number to new object name if there is a collision.

Prepending the scope name avoids collisions but does not prevent them in all cases. The new name with prepended scope name may still collide with an existing object. In this situation, the new object from the external file will replace that existing object.

When appending instance numbers, instance numbers shall be chosen to avoid collision with previously loaded objects.

9.4.2.12 String: World Alias Name

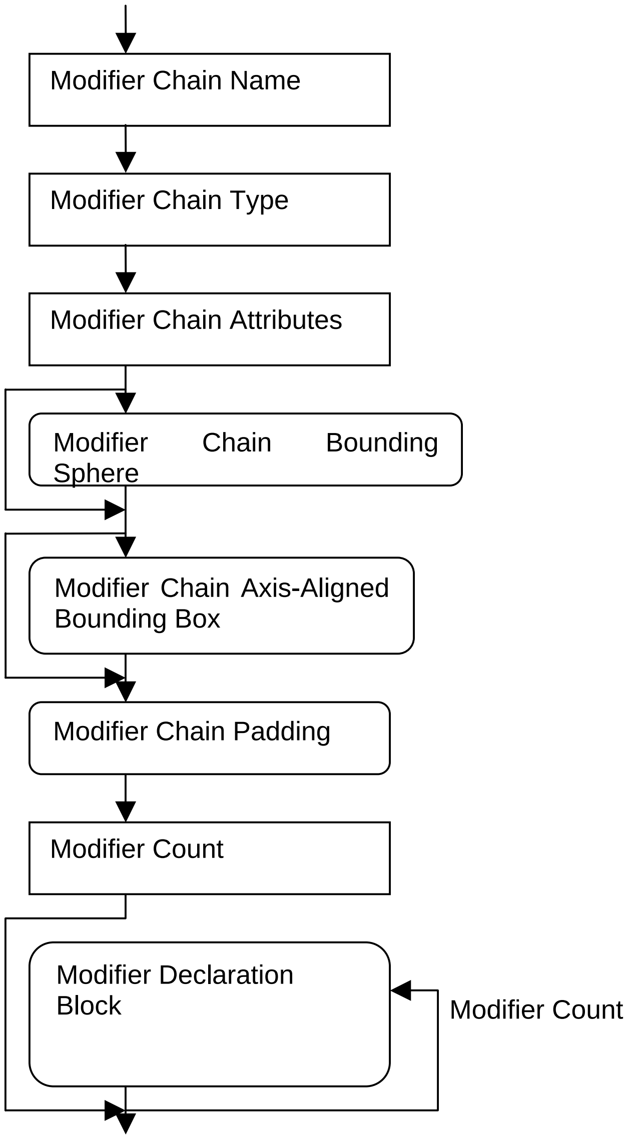
The world is the default node. The name of the default node is the empty string. Any references to the default node in the external file are replaced with a reference to the node named by World Alias Name.

9.4.3 Modifier Chain (blocktype: 0xFFFFFF14)

Modifier Chain blocks are used to contain the declaration blocks for an object and its modifiers.

If an object does not have any modifiers, then the declaration block for that object may be contained in a modifier chain block but is not required to be contained in a modifier chain block.

If an object does have modifiers, then the declaration blocks for the object and its modifiers shall be contained in a modifier chain block.


Modifier Chain Name
Modifier Chain Type
Modifier Chain Attributes
Modifier Chain Bounding Sphere
Modifier Chain Axis-Aligned Bounding Box
Modifier Chain Padding
Modifier Count
Modifier Declaration Block
Modifier Count

9.4.3.1 String: Modifier Chain Name

Modifier Chain Name is the name of the modifier chain and also the name of all modifiers in the chain.

9.4.3.2 U32: Modifier Chain Type

Modifier Chain Type indicates the type of modifier chain.

0 – Node modifier chain (also called instance modifier chain).

1 – Model Resource modifier chain (also called resource modifier chain).

2 – Texture Resource modifier chain (also called texture modifier chain).

9.4.3.3 U32: Modifier Chain Attributes

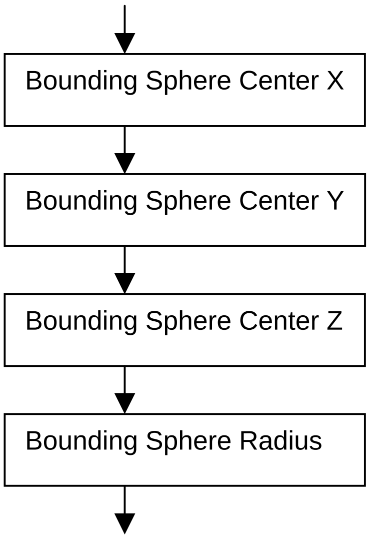
Modifier Chain Attributes is a bitfield indicating the presence of optional information about the modifer chain. The bounding information is optional. All other values are reserved.

0x00000001 – Bounding sphere information present

0x00000002 – Axis-aligned bounding box present

9.4.3.4 Modifier Chain Bounding Sphere

The bounding sphere should contain all of the geometry expected to be produced by the modifier chain. This bounding sphere is an initial estimate of the extent of the geometry and may be updated by the run-time after loading. The bounding sphere information in this block may be used to exclude the modifier chain from loading.



9.4.3.4.1 F32: Bounding Sphere Center X

X coordinate of the center of the bounding sphere

9.4.3.4.2 F32: Bounding Sphere Center Y

Y coordinate of the center of the bounding sphere

9.4.3.4.3 F32: Bounding Sphere Center Z

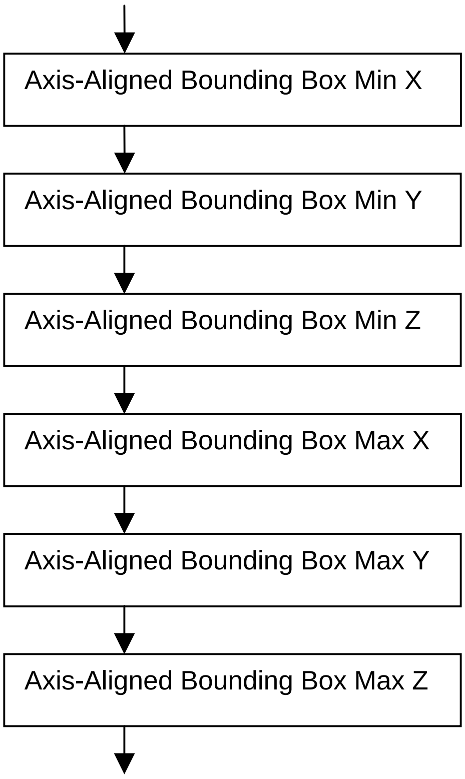
Z coordinate of the center of the bounding sphere

9.4.3.4.4 F32: Bounding Sphere Radius

Radius of the bounding sphere

9.4.3.5 Modifier Chain Axis-Aligned Bounding Box

The axis-aligned bounding box should contain all of the geometry expected to be produced by the modifier chain. This axis-aligned bounding box is an initial estimate of the extent of the geometry and may be updated by the run-time after loading. The axis-aligned bounding box information in this block may be used to exclude the modifier chain from loading.


Axis-Aligned Bounding Box Min X
Axis-Aligned Bounding Box Min Y
Axis-Aligned Bounding Box Min Z
Axis-Aligned Bounding Box Max X
Axis-Aligned Bounding Box Max Y
Axis-Aligned Bounding Box Max Z

9.4.3.5.1 F32: Axis-Aligned Bounding Box Min X

X coordinate of the bounding box minimum corner

9.4.3.5.2 F32: Axis-Aligned Bounding Box Min Y

Y coordinate of the bounding box minimum corner

9.4.3.5.3 F32: Axis-Aligned Bounding Box Min Z

Z coordinate of the bounding box minimum corner

9.4.3.5.4 F32: Axis-Aligned Bounding Box Max X

X coordinate of the bounding box maximum corner

9.4.3.5.5 F32: Axis-Aligned Bounding Box Max Y

Y coordinate of the bounding box maximum corner

9.4.3.5.6 F32: Axis-Aligned Bounding Box Max Z

Z coordinate of the bounding box maximum corner

9.4.3.6 variable: Modifier Chain Padding

Modifier Chain Padding is a variable size field. Zero to three bytes shall be inserted to maintain 32-bit alignment for the start of the Modifier Count field. This padding also provides 32-bit alignment for the start of the Modifier Declaration Blocks. The value of any padding bytes is 0x00.

9.4.3.7 U32: Modifier Count

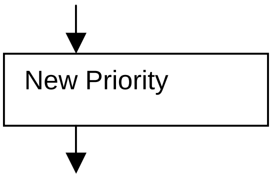
Modifier Count is the number of modifiers in the modifier chain.

9.4.3.8 Modifier Declaration Block

Modifier Declaration Block is a declaration block for a modifier in the modifier chain. All declaration blocks for modifiers must be contained in a Modifier Chain Block. The modifier name in the Modifier Declaration Block shall match the Modifier Chain Name. Details of the Modifier Declaration Block can be found in the sections for those blocks.

9.4.4 Priority Update (blocktype: 0xFFFFFF15)

Priority Update blocks indicate the priority number of following continuation blocks. Priority Update blocks are in the continuation section of the file. The Priority Update block is considered a continuation of the File Header block. Priority Update blocks are not required.



9.4.4.1 U32: New Priority

Blocks which follow this block have a priority number of New Priority. A lower priority number means the block appears earlier in the file. The value of New Priority in this block shall not be less than the value of New Priority in priority update blocks earlier in the file. New Priority shall be greater than zero.

9.5 Node blocks

Nodes are the entities that populate the scene graph. Each node type contains a name, the number of parents it has, the name of each parent, and a transform for each parent specifying the position and orientation of the node relative to that parent. Nodes (except for the group node, covered below) also have an associated resource that is specified by name. To allow data sharing, multiple nodes may use the same resource. Nodes may also contain additional fields that are used during rendering for each instance of a resource.

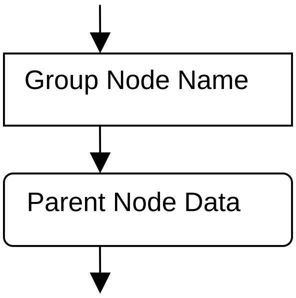
9.5.1 Group Node (blocktype: 0xFFFFFF21)

The Group Node contains: a name, the number of parents, the parents' names, and a transform relative to each parent. Group nodes are used to collect other nodes to build up larger objects.

The Group Node produces the following outputs: Transform Set.

The Group Node's outputs have no dependencies.

Example: A car may be composed of many model nodes to make up the body, several light nodes for lights on the car, and a few view nodes to simulate the car's mirrors. Instead of choosing one of the nodes to be the parent node and having all other nodes rendered relative to that node, they can all be children of a group node named "car." This allows any of the children nodes to be modified or deleted without affecting the other nodes.



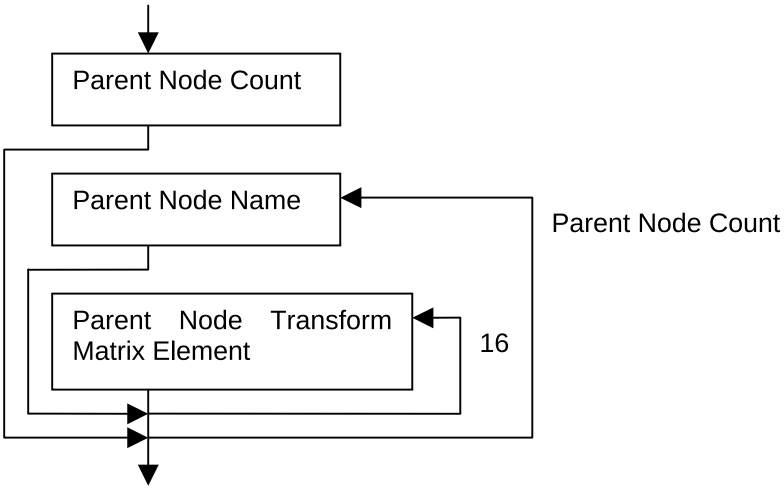
9.5.1.1 String: Group Node Name

Group Node Name is the name of the group node.

9.5.1.2 Parent Node Data

Recursive parent child relationships (e.g. Node_1 is a child of Node_2 is a child of Node_1) will cause infinite loops when evaluating transforms (because Node_2's transform depends on Node_1's transform, and Node_1's transform depends on Node_2's transform). Recursion in the parent child hierarchy is illegal and will generate an error. These relationships must be checked at load time.

A parent's name may be an empty string. In this case, the parent node is the default entry in the node palette. The default node palette entry is a group node.



9.5.1.2.1 U32: Parent Node Count

Parent Node Count is the number of parent nodes for this node. A node may have zero parents.

9.5.1.2.2 String: Parent Node Name

Each parent node is identified by the object's name.

9.5.1.2.3 F32: Parent Node Transform Matrix Element

This node holds a transform matrix indicating the position and orientation of the node relative to each parent node. There is a separate transformation matrix for each parent. The matrix is written in the alphabetic order described below:

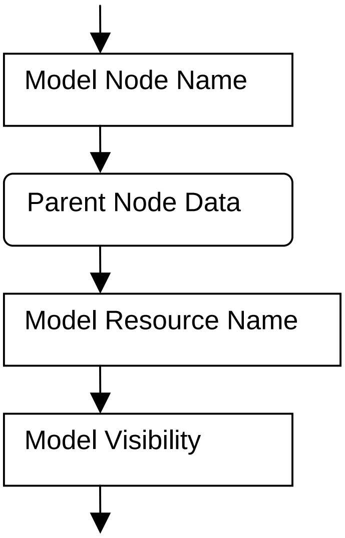
$$\begin{bmatrix} A & E & I & M \\ B & F & J & N \\ C & G & K & O \\ D & H & L & P \end{bmatrix}$$

9.5.2 Model Node (blocktype: 0xFFFFFF22)

The Model Node contains: a name, the number of parents, the parents' names, and a transform relative to each parent. The Model Node also contains the name of a model resource chain. A model node is the first modifier in a node modifier chain. The node modifier chain takes input from the model resource modifier chain that is specified by the model resource name field in the model node.

The Model Node produces the following outputs: Transform Set, View Frustum, View Size

The Model Node's outputs depend on: Transform.



9.5.2.1 String: Model Node Name

The Model Node is identified by the Model Node Name.

9.5.2.2 Parent Node Data

Described in 9.5.1.2 Parent Node Data for the group node block.

9.5.2.3 String: Model Resource Name

Model Resource Name is the name of the model resource chain used as input to the model node's modifier chain.

9.5.2.4 U32: Model Visibility

Model Visibility is used to indicate whether the front facing or back facing surface should be drawn. All other values are reserved.

0 – Not visible

1 – Front visible

2 – Back visible

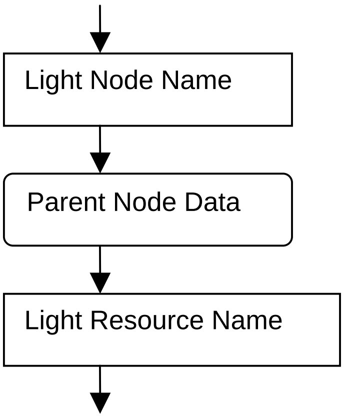
3 – Front and back visible

9.5.3 Light Node (blocktype: 0xFFFFFF23)

The Light Node contains: a name, the number of parents, the parents' names, and a transform relative to each parent. The Light Node also contains the name of a light resource. All other information needed for a light is contained in the light resource; so, the Light Node does not have any additional fields.

The Light Node produces the following outputs: Transform Set.

The Light Node's outputs have no dependencies.



9.5.3.1 String: Light Node Name

The Light Node is identified by Light Node Name.

9.5.3.2 Parent Node Data

Described in 9.5.1.2 Parent Node Data for the group node block.

9.5.3.3 String: Light Resource Name

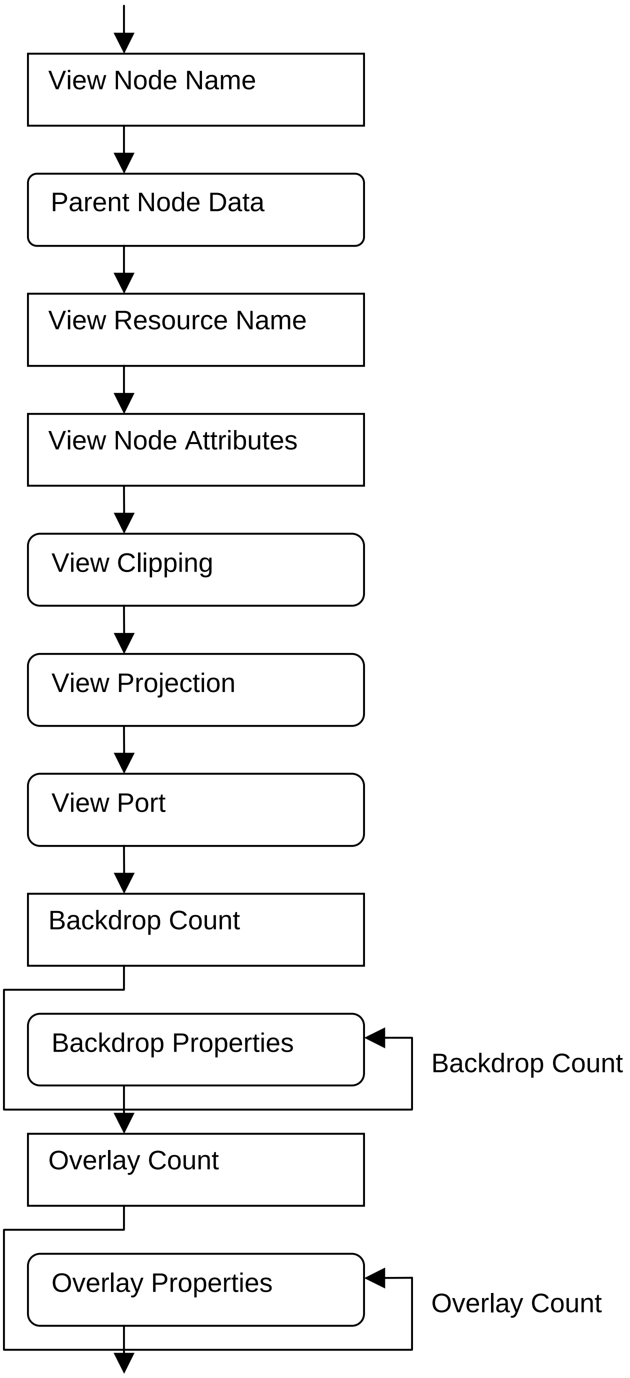
Light Resource Name identifies the light resource used by this Light Node.

9.5.4 View Node (blocktype: 0xFFFFFF24)

The View Node contains: a name, the number of parents, the parents' names, and a transform relative to each parent. The View Node also contains the name of a view resource, clipping, projection, and view port fields that are specific to this instance and define how the view is rendered on the screen. The clipping information specifies what part of the world is available to the view. The view may be rendered with one, two, or three point perspective projection or orthogonal projection. The view port fields determine where on the screen the view will be rendered.

The Group Node produces the following outputs: Transform Set.

The Group Node's outputs have no dependencies.



9.5.4.1 String: View Node Name

View Node Name identifies the View Node.

9.5.4.2 Parent Node Data

Described in 9.5.1.2 Parent Node Data for the group node block.

9.5.4.3 String: View Resource Name

View Resource Name identifies the view resource used by this View Node.

9.5.4.4 U32: View Node Attributes

View Node Attributes is a bitfield used to indicate different modes of operation of the view node. View Node Attributes are defined for projection mode and for screen position units mode. Attributes can be combined by OR operation. Only one projection mode can be selected. All other values are reserved.

0x00000000 – default attributes: three-point perspective projection and screen position units expressed in screen pixels.

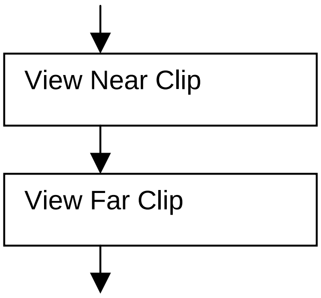
0x00000001 – screen position units: expressed as percentage of screen dimension.

0x00000002 – projection mode: orthographic projection is used by the view

0x00000004 – projection mode: two-point perspective projection is used by the view

0x00000006 – projection mode: one-point perspective projection is used by the view

9.5.4.5 View Clipping



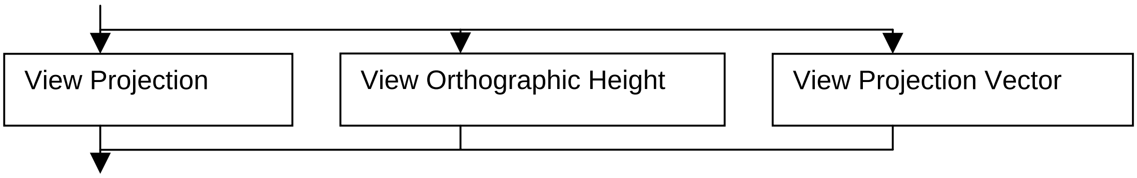
9.5.4.5.1 F32: View Near Clip

View Near Clip is the near clipping distance. Elements closer to the View Node than the near clipping distance are not drawn.

9.5.4.5.2 F32: View Far Clip

View Far Clip is the far clipping distance. Elements farther from the View Node than the far clipping distance are not drawn.

9.5.4.6 View Projection



9.5.4.6.1 F32: View Projection

View Projection is the field of view of the virtual camera in degrees. This value is only present for three-point perspective projection mode. Projection mode is defined in View Node Attributes. Details of rendering are outside the scope of this specification. A renderer would be allowed to adjust the volume of space rendered by a particular view for various purposes such as to reduce perspective distortion.

9.5.4.6.2 F32: View Orthographic Height

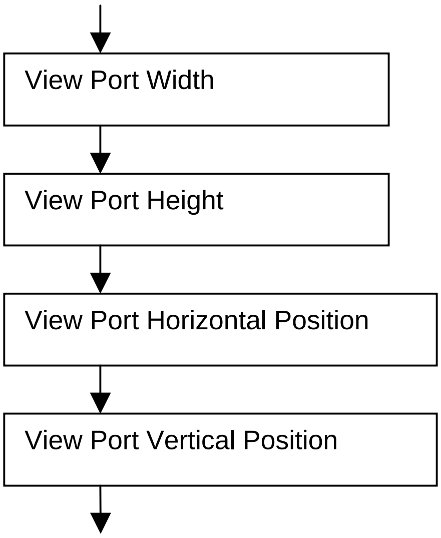
View Orthographic Height is the height of the orthographic view. This value is only present for orthographic projection mode. Projection mode is defined in View Node Attributes.

9.5.4.6.3 View Projection Vector

View Projection Vector is only present for one-point and two-point perspective projection mode. For one-point perspective projection, View Projection Vector is a vector normal to the view plane. For two-point perspective projection, View Projection Vector is a vector in the "up" direction for this view node.

9.5.4.7 View Port

The View Port describes the window in screen space in which the view will render. The units used by the View Port are defined in View Node Attributes. The View Port values are expressed either in screen pixels or as a fraction of the screen dimensions. When using screen fraction units, the View Port will occupy the entire screen if the width and height are set to 1.0 and the horizontal and vertical position are set to 0.0.



9.5.4.7.1 F32: View Port Width

View Port Width is the width of the window in which the view will render.

9.5.4.7.2 F32: View Port Height

View Port Height is the height of the window in which the view will render.

9.5.4.7.3 F32: View Port Horizontal Position

View Port Horizontal Position is the horizontal position on the screen of the window in which the view will render. Position is measured from the upper left corner of the screen.

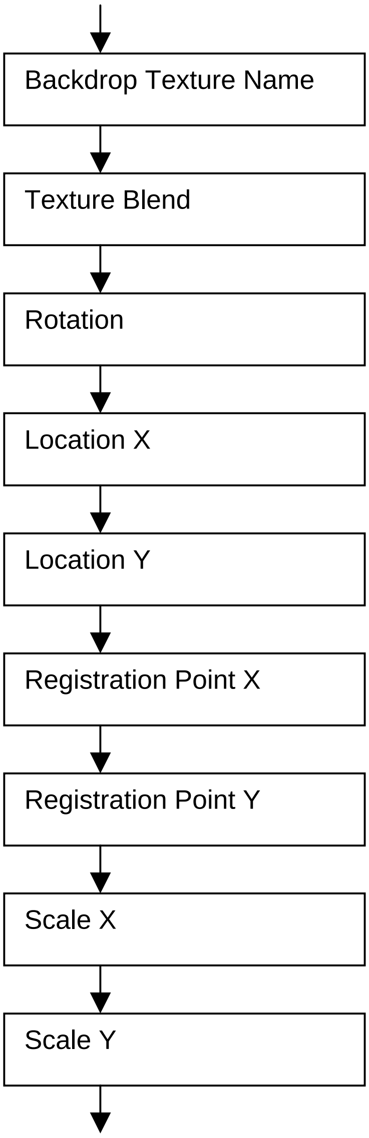
9.5.4.7.4 F32: View Port Vertical Position

View Port Vertical Position is the vertical position on the screen of the window in which the view will render. Position is measured from the upper left corner of the screen.

9.5.4.8 U32: Backdrop Count

The Backdrop Count is the number of backdrops the view has. A backdrop is a texture displayed in this view behind all objects rendered. Backdrops are displayed in order with the first backdrop displayed behind the next backdrop.

9.5.4.9 Backdrop Properties



9.5.4.9.1 String: Backdrop Texture Name

Backdrop Texture Name is the name of the texture resource to use for this backdrop.

9.5.4.9.2 F32: Texture Blend

Texture Blend is the blend factor used with the backdrop's texture.

9.5.4.9.3 F32: Rotation

The Rotation is how the texture used with the backdrop is rotated. Rotation is measured in radians, counter clockwise.

9.5.4.9.4 F32: Location X

The Location X is the backdrop's horizontal location. The position of the backdrop is measured from the upper left corner of the display to the registration point. The units used are defined in View Node Attributes.

9.5.4.9.5 F32: Location Y

The Location Y is the backdrop's vertical location. The position of the backdrop is measured from the upper left corner of the display to the registration point. The units used are defined in View Node Attributes.

9.5.4.9.6 I32: Registration Point X

Registration Point X is the horizontal registration point. The registration point of the backdrop texture is measured in texture pixels from the upper left corner of the texture.

9.5.4.9.7 I32: Registration Point Y

Registration Point Y is the vertical registration point. The registration point of the backdrop texture is measured in texture pixels from the upper left corner of the texture.

9.5.4.9.8 F32: Scale X

Scale X is a scale factor applied to the backdrop horizontally.

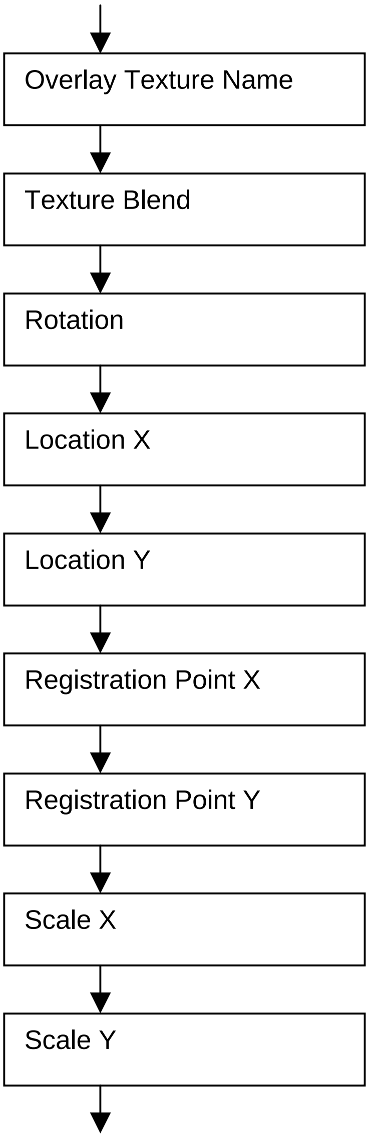
9.5.4.9.9 F32: Scale Y

Scale Y is a scale factor applied to the backdrop vertically.

9.5.4.10 U32: Overlay Count

The Overlay Count is the number of overlays used with this view. An overlay is a texture displayed in this view in front of all objects rendered. Overlays are displayed in order with the first overlay displayed behind the next overlay.

9.5.4.11 Overlay Properties



9.5.4.11.1 String: Overlay Texture Name

Overlay Texture Name is the name of the texture resource to use for this overlay.

9.5.4.11.2 F32: Texture Blend

Texture Blend is the blend factor applied to the texture used for this overlay.

9.5.4.11.3 F32: Rotation

Rotation is how much the texture is rotated. Rotation is measured in radians, counter clockwise.

9.5.4.11.4 F32: Location X

Location X is the horizontal position of the overlay. The position of the overlay is measured from the upper left corner of the display to the registration point. The units used are defined in View Node Attributes.

9.5.4.11.5 F32: Location Y

Location Y is the vertical position of the overlay. The position of the overlay is measured from the upper left corner of the display to the registration point. The units used are defined in View Node Attributes.

9.5.4.11.6 I32: Registration Point X

Registration Point X is the horizontal registration point. The registration point of the overlay texture is measured in texture pixels from the upper left corner of the texture.

9.5.4.11.7 I32: Registration Point Y

Registration Point Y is the vertical registration point. The registration point of the overlay texture is measured in texture pixels from the upper left corner of the texture.

9.5.4.11.8 F32: Scale X

Scale X is the scale factor applied to the overlay horizontally.

9.5.4.11.9 F32: Scale Y

Scale Y is the scale factor applied to the overlay vertically.

9.6 Geometry generator blocks

Geometry generator blocks contain the declarative information for creating model resource modifier chains. The model resource modifier chains serve as input to the node modifier chains.

9.6.1 CLOD Mesh Generator (blocktypes: 0xFFFFFF31; 0xFFFFFF3B; 0xFFFFFF3C)

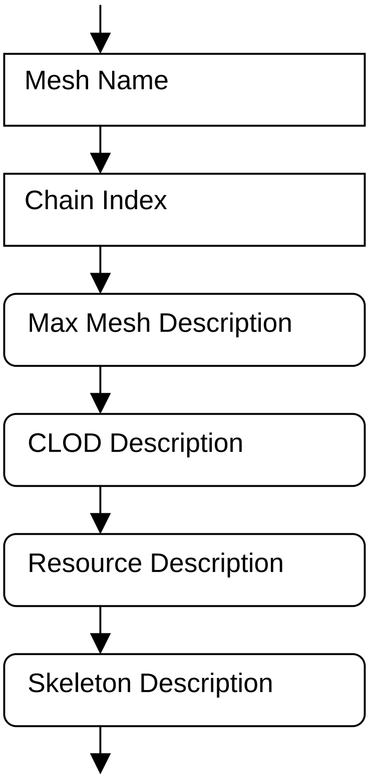
The CLOD Mesh Generator contains the data needed to create a continuous level of detail mesh. This data includes vertices, normal vectors, faces, shading lists, and level of detail information for the base mesh and updates. The information in the CLOD Mesh Generator blocks describes the author mesh. The CLOD Mesh Genarator converts the author mesh into a render mesh for display. Description of differences between the author mesh and render mesh can be found in 8.8 CLOD mesh generator.

The CLOD Mesh Generator produces the following outputs: Renderable Group, Renderable Group Bounds, Transform Set.

The CLOD Mesh Generator's outputs have no dependencies.

9.6.1.1 CLOD Mesh Declaration (blocktype: 0xFFFFFF31)

The CLOD Mesh Declaration contains the declaration information for a continuous level of detail mesh generator. The declaration information is sufficient to allocate space for the mesh data and create the mesh generator object. The mesh data is contained in following continuation blocks.



9.6.1.1.1 String: Mesh Name

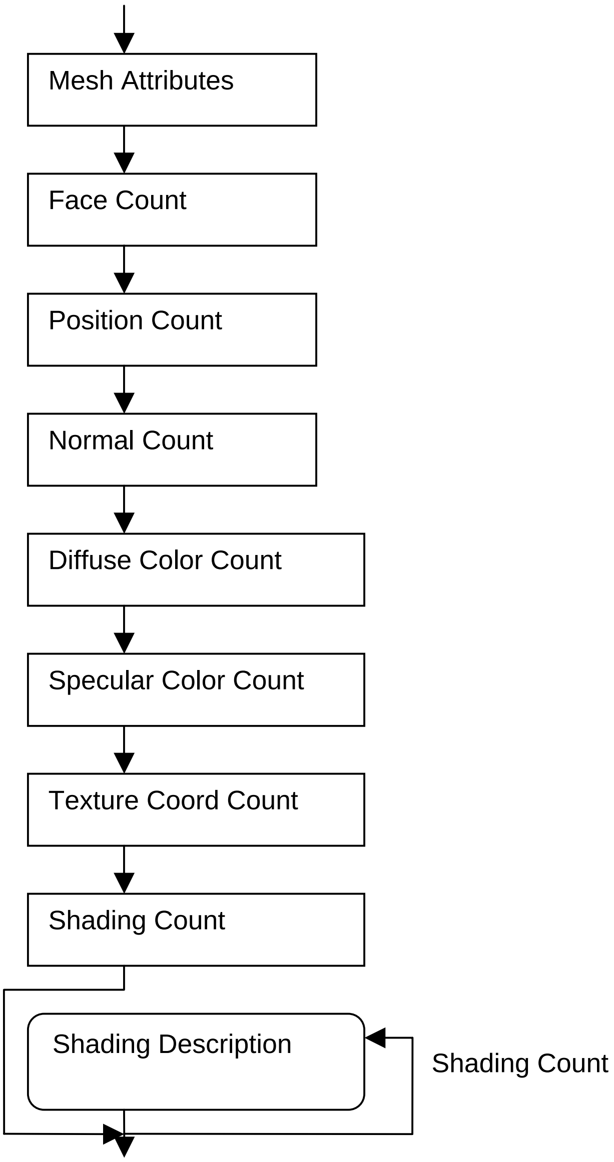
Mesh Name is the name of the CLOD mesh generator. This name is also the name of the model resource modifier chain that contains the CLOD mesh generator.

9.6.1.1.2 U32: Chain Index

Chain Index is the position of the CLOD mesh generator in the model resource modifier chain. The value of Chain Index shall zero for this blocktype.

9.6.1.1.3 Max Mesh Description

Max Mesh Description describes the size of the mesh at full resolution. Max Mesh Description can be used to allocate space for the mesh.



9.6.1.1.3.1 U32: Mesh Attributes

Mesh Attributes contains information that applies to the entire mesh. Mesh Attributes is a collection of flags. The only flag currently defined indicates the usage of per vertex normals. The flags are combined using a bitwise OR operation. All other values are reserved.

0x00000000 – Default: The faces in the mesh have a normal index at each corner.

0x00000001 – Exclude Normals: The faces in the mesh do not have a normal index at each corner.

An implementation that requires normals may generate normals for a mesh which does not have normals.

9.6.1.1.3.2 U32: Face Count

Face Count is the number of faces in the mesh.

9.6.1.1.3.3 U32: Position Count

Position Count is the number of positions in the position array.

9.6.1.1.3.4 U32: Normal Count

Normal Count is the number of normals in the normal array.

9.6.1.1.3.5 U32: Diffuse Color Count

Diffuse Color Count is the number of colors in the diffuse color array.

9.6.1.1.3.6 U32: Specular Color Count

Specular Color Count is the number of colors in the specular color array.

9.6.1.1.3.7 U32: Texture Coord Count

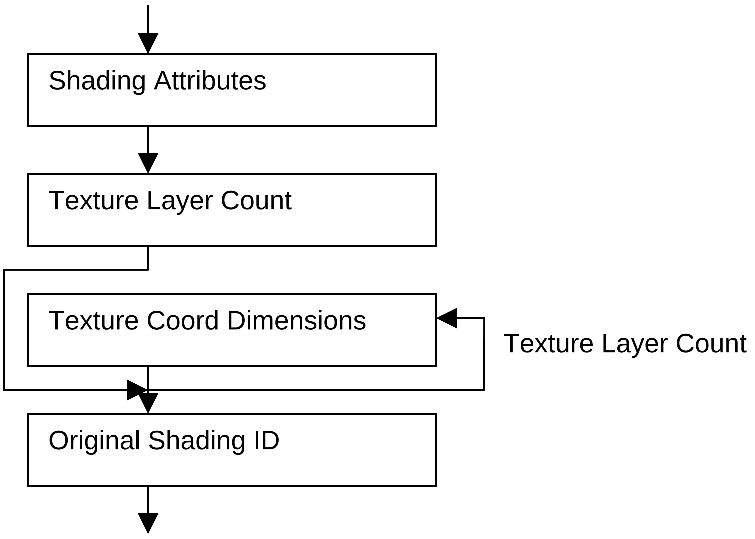
Texture Coord Count is the number of texture coordinates in the texture coordinate array.

9.6.1.1.3.8 U32: Shading Count

Shading Count is the number of shading descriptions used in the mesh. Each shading description corresponds to one shader list in the shading group.

9.6.1.1.3.9 Shading Description

Shading Description indicates which per vertex attributes, in addition to position and normal, are used by each shading list.



9.6.1.1.3.9.1 U32: Shading Attributes

Shading Attributes is a collection of flags combined using the binary OR operator. These flags are used to indicate the usage of per vertex colors. The flags are combined using a bitwise OR operation. All other values are reserved.

0x00000000 – The shader list uses neither diffuse colors nor specular colors.

0x00000001 – The shader list uses per vertex diffuse colors.

0x00000002 – The shader list uses per vertex specular colors.

0x00000003 – The shader list uses both diffuse and specular colors, per vertex.

9.6.1.1.3.9.2 U32: Texture Layer Count

Texture layer Count is the number of texture layers used by this shader list.

9.6.1.1.3.9.3 U32: Texture Coord Dimensions

Texture Coord Dimensions is the number of dimensions in the texture coordinate vector. The texture coordinate vector can have 1, 2, 3, or 4 dimensions.

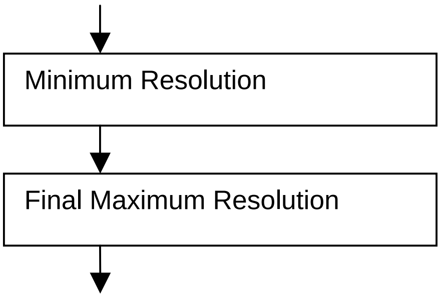
9.6.1.1.3.9.4 U32: Original Shading ID

Original Shading ID is the original shading index for this shader list. Shader lists may be re-ordered during the encode process. Unused shader lists may be removed by the encode process.

9.6.1.1.4 CLOD Description

CLOD Description describes the range of resolutions available for the continuous level of detail mesh.

Two special cases are worth noting. If the Minimum Resolution is zero, then there is no base mesh. If the Minimum Resolution is equal to the Final Maximum Resolution, then the base mesh is the entire mesh and the CLOD mechanism cannot change the resolution of the mesh.



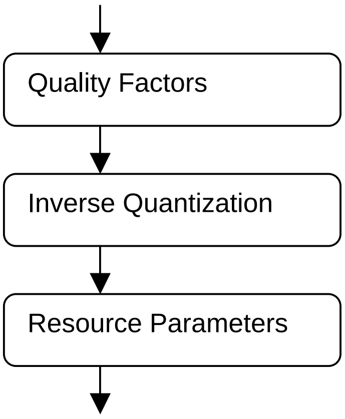
9.6.1.1.4.1 U32: Minimum Resolution

Minimum Resolution shall be the number of positions in the base mesh.

9.6.1.1.4.2 U32: Final Maximum Resolution

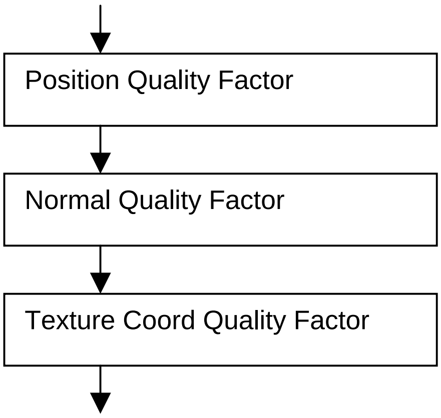
Final Maximum Resolution shall be the number of positions in the Max Mesh Description.

9.6.1.1.5 Resource Description



9.6.1.1.5.1 Quality Factors

The quality factors are for information only and are not used. The quality factors enable the user interface to provide the user with information on some of the parameters used to encode the mesh.



9.6.1.1.5.1.1 U32: Position Quality Factor

Position Quality Factor is the quality factor associated with quantization of positions.

9.6.1.1.5.1.2 U32: Normal Quality Factor

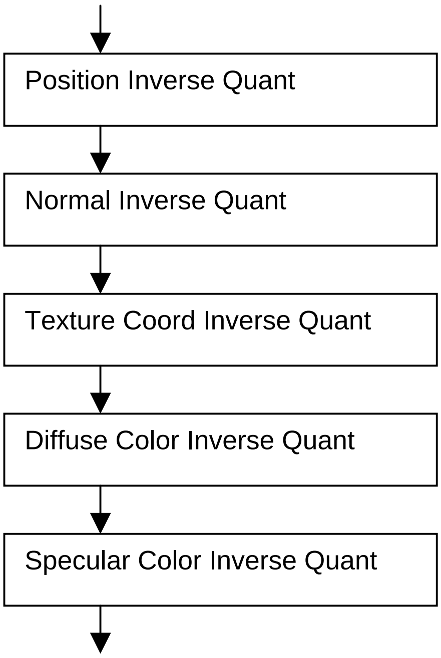
Normal Quality Factor is the quality factor associated with quantization of normal vectors.

9.6.1.1.5.1.3 U32: Texture Coord Quality Factor

Texture Coord Quality Factor is the quality factor associated with quantization of texture coordinates.

9.6.1.1.5.2 Inverse Quantization

Inverse Quantization contains the inverse quantization factors used to reconstruct floating point values that had been quantized.


Position Inverse Quant
Normal Inverse Quant
Texture Coord Inverse Quant
Diffuse Color Inverse Quant
Specular Color Inverse Quant

9.6.1.1.5.2.1 F32: Position Inverse Quant

Position Inverse Quant is the inverse quantization factor used in the reconstruction of position vectors.

9.6.1.1.5.2.2 F32: Normal Inverse Quant

Normal Inverse Quant is the inverse quantization factor used in the reconstruction of normal vectors.

9.6.1.1.5.2.3 F32: Texture Coord Inverse Quant

Texture Coord Inverse Quant is the inverse quantization factor used in the reconstruction of texture coordinates.

9.6.1.1.5.2.4 F32: Diffuse Color Inverse Quant

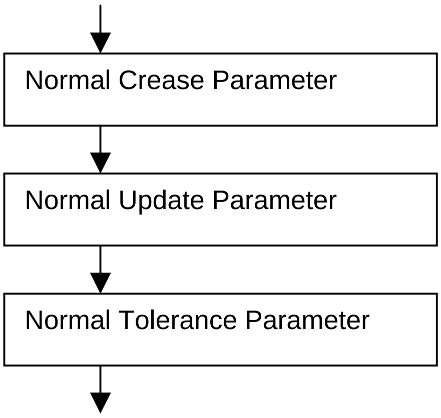
Diffuse Color Inverse Quant is the inverse quantization factor used in the reconstruction of diffuse colors.

9.6.1.1.5.2.5 F32: Specular Color Inverse Quant

Specular Color Inverse Quant is the inverse quantization factor used in the reconstruction of specular colors.

9.6.1.1.5.3 Resource Parameters

Resource Parameters control the operation of the CLOD mesh generator. The parameters defined in this section control the conversion of the mesh from Author Mesh format to Render Mesh format.



9.6.1.1.5.3.1 F32: Normal Crease Parameter

In the conversion from Author Mesh to Render Mesh, normals that are sufficiently close together are merged. The closeness of normals is measured by calculating the dot product between them. The resulting closeness measure is in the range –1.0 to +1.0. – 1.0 is farthest apart and + 1.0 is closest together.

Normals at the same position which are closer than Normal Crease Parameter are merged. In other words, if the dot product between two normals is larger than Normal Crease Parameter, then the two normals shall be merged.

Normal Crease Parameter can be used to trade off smoothing over edges against preservation of sharp edges.

9.6.1.1.5.3.2 F32: Normal Update Parameter

In decoding the Author Mesh normals, a correction is made to a predicted normal. If the corrected normal is closer to the predicted normal than Normal Update Parameter, then the normal correction can be dropped.

9.6.1.1.5.3.3 F32: Normal Tolerance Parameter

Normals which are closer together than Normal Tolerance Parameter are considered equivalent in the conversion of the Author Mesh to Render Mesh. A more compact Render Mesh can be created if more normals are allowed to be replaced by similar normals. The compactness of the Render Mesh is traded off against the accuracy or the Render Mesh normals.

9.6.1.1.6 Skeleton Description

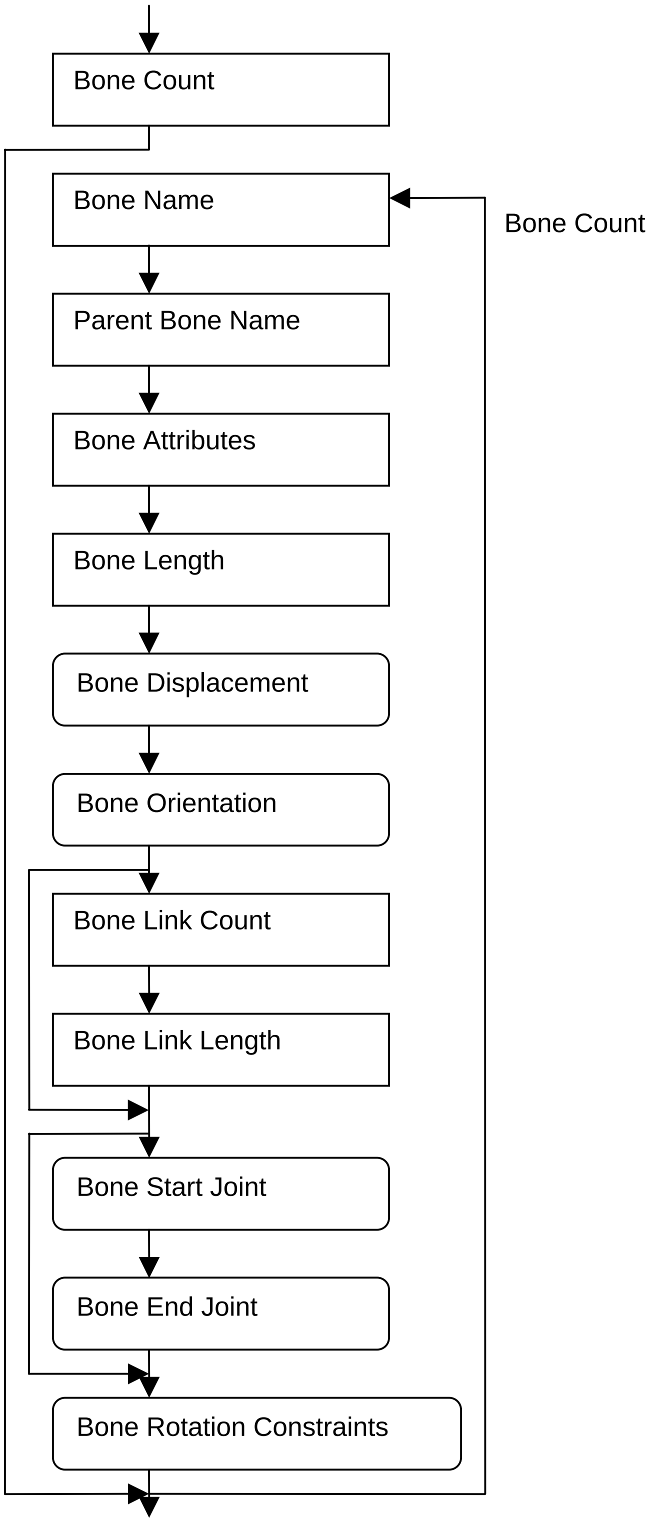
Skeleton Description provides bone structure information. If there is no bone structure associated with the generator, then the Bone Count is zero. Bones are structured in a tree hierarchy. The position and orientation of each bone is described relative to a parent bone.

The Skeleton Description is used in bones-based animation. The Animation Modifier uses the bone structure to deform geometry based on the position and orientation of the bones. The Animation Modifier also modifies the bone structure based on information in a Motion Resource.

In a future editions of this specification, it may be possible to use the Skeleton Description for additional features such as inverse kinematics and automatic bone weight generation.

Bone links are small bones that are inserted between the start of a bone and the end of its parent bone. The placement of bone links is done automatically based on the position and orientation of the bone and its parent.

Bone links and bone joint information may be useful for inverse kinematics extensions and automatic bone weight generation extensions.



9.6.1.1.6.1 U32: Bone Count

Bone Count is the number of bones associated with the mesh.

9.6.1.1.6.2 String: Bone Name

Bone Name is the name of this bone.

9.6.1.1.6.3 String: Parent Bone Name

Parent Bone Name is the name of the parent of this bone.

For the first bone, the Parent Bone Name shall be the empty string. The first bone is called the root bone. For bones after the first bone, the Parent Bone Name shall be the Bone Name of a previous bone.

9.6.1.1.6.4 U32: Bone Attributes

Bone Attributes is a collection of flags. The flags are combined using the binary OR operator. These flags are used to indicate the presence of optional link and joint sections and whether the rotational constraints are enabled. All other values are reserved.

0x00000001 – The Bone Link Count and Bone Link Length are present.

0x00000002 – The Bone Start Joint and Bone End Joint sections are present.

0x00000004 – The X Rotation Constraint is active.

0x00000008 – The X Rotation Constraint is limited.

0x00000010 – The Y Rotation Constraint is active.

0x00000020 – The Y Rotation Constraint is limited.

0x00000040 – The Z Rotation Constraint is active.

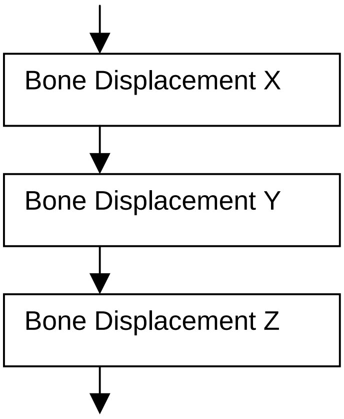
0x00000080 – The Z Rotation Constraint is limited.

9.6.1.1.6.5 F32: Bone Length

Bone Length is the length of this bone. The length of the bone is not modified by the Animation Modifier in 9.7.3.

9.6.1.1.6.6 Bone Displacement

Bone Displacement is the displacement of the start of this bone from the end of its parent bone.



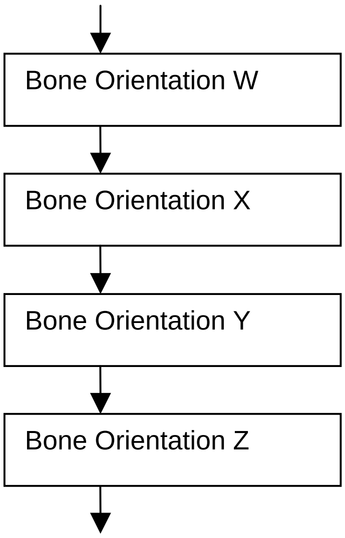
9.6.1.1.6.6.1 F32: Bone Displacement X

9.6.1.1.6.6.2 F32: Bone Displacement Y

9.6.1.1.6.6.3 F32: Bone Displacement Z

9.6.1.1.6.7 Bone Orientation

Bone Orientation is the change in orientation of this bone relative to the orientation of its parent bone. The change in orientation is expressed as a quaternion.



9.6.1.1.6.7.1 F32: Bone Orientation W

9.6.1.1.6.7.2 F32: Bone Orientation X

9.6.1.1.6.7.3 F32: Bone Orientation Y

9.6.1.1.6.7.4 F32: Bone Orientation Z

9.6.1.1.6.8 U32: Bone Link Count

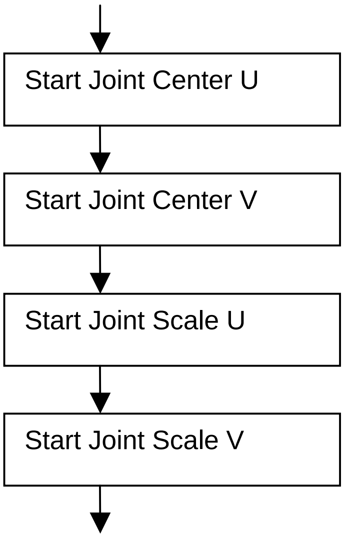
Bone Link Count is the number of bone links between the end of the parent bone and the start of this bone.

9.6.1.1.6.9 F32: Bone Link Length

Bone Link Length is the length of the bone links.

9.6.1.1.6.10 Bone Start Joint

Bone Start Joint describes an ellipse that approximates the cross-section of the geometry surrounding the bone at the start of the bone. The ellipse is oriented in the local coordinate space of the bone. Start Joint Center is a displacement of the center for the ellipse from the axis of the bone. Start Joint Scale provides major and minor axis of the ellipse.



9.6.1.1.6.10.1 F32: Start Joint Center U

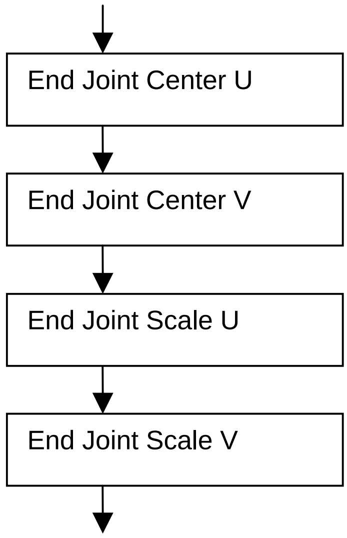
9.6.1.1.6.10.2 F32: Start Joint Center V

9.6.1.1.6.10.3 F32: Start Joint Scale U

9.6.1.1.6.10.4 F32: Start Joint Scale V

9.6.1.1.6.11 Bone End Joint

Bone End Joint describes an ellipse that approximates the cross-section of the geometry surrounding the bone at the end of the bone. The ellipse is oriented in the local coordinate space of the bone. End Joint Center is a displacement of the center for the ellipse from the axis of the bone. End Joint Scale provides major and minor axis of the ellipse.



9.6.1.1.6.11.1 F32: End Joint Center U

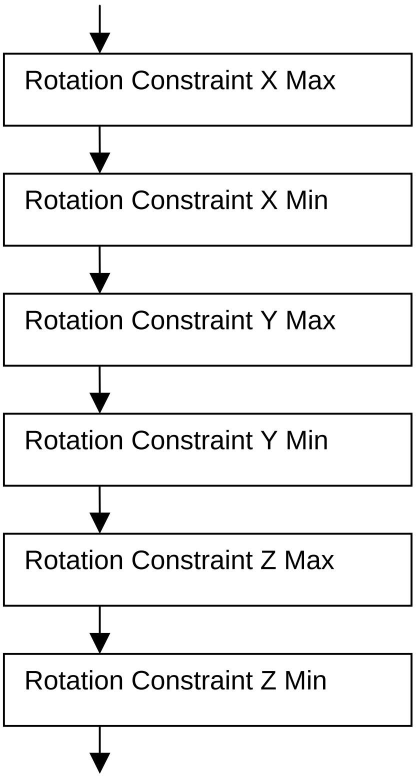
9.6.1.1.6.11.2 F32: End Joint Center V

9.6.1.1.6.11.3 F32: End Joint Scale U

9.6.1.1.6.11.4 F32: End Joint Scale V

9.6.1.1.6.12 Bone Rotation Constraints

An inverse kinematics extension could use these bone rotation constraints when updating the bone positions and orientations. The Animation Modifier does not use these Bone Rotation Constraints.


Rotation Constraint X Max
Rotation Constraint X Min
Rotation Constraint Y Max
Rotation Constraint Y Min
Rotation Constraint Z Max
Rotation Constraint Z Min

9.6.1.1.6.12.1 F32: Rotation Constraint X Max

9.6.1.1.6.12.2 F32: Rotation Constraint X Min

9.6.1.1.6.12.3 F32: Rotation Constraint Y Max

9.6.1.1.6.12.4 F32: Rotation Constraint Y Min

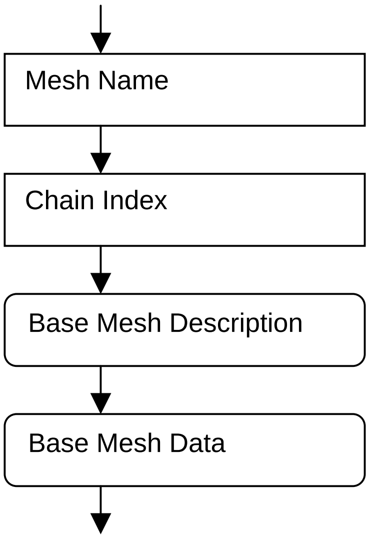
9.6.1.1.6.12.5 F32: Rotation Constraint Z Max

9.6.1.1.6.12.6 F32: Rotation Constraint Z Min

9.6.1.2 CLOD Base Mesh Continuation (blocktype: 0xFFFFFF3B)

The CLOD Base Mesh Continuation block contains base mesh information for a continuous level of detail mesh generator. The base mesh is the minimum LOD mesh. The base mesh does not contain resolution updates. As a result, a CLOD resolution controller cannot reduce the mesh resolution to less than the size of the base mesh (other than reducing the mesh to zero resolution). The base mesh is not quantized.

The CLOD Base Mesh Continuation block is a continuation type block. The CLOD Base Mesh Continuation block is only present if Minimum Resolution is greater than zero.



9.6.1.2.1 String: Mesh Name

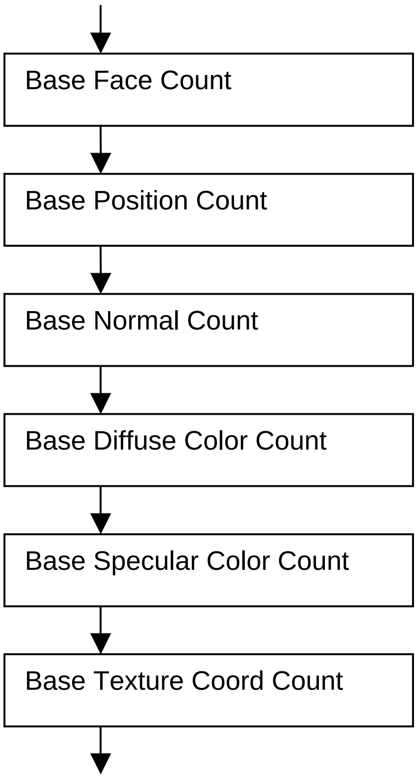
Mesh Name is the name of the CLOD mesh generator. This name is also the name of the model resource modifier chain that contains the CLOD mesh generator.

9.6.1.2.2 U32: Chain Index

Chain Index is the position of the CLOD mesh generator in the model resource modifier chain. The value of Chain Index shall be zero for this blocktype.

9.6.1.2.3 Base Mesh Description

Base Mesh Description describes the size of the mesh at minimum resolution. Base Mesh Description indicates the portion of space allocated for the mesh that is used by the base mesh. The elements of the base mesh occupy the first part (lowest index) of each of the various mesh arrays.



9.6.1.2.3.1 U32: Base Face Count

Base Face Count is the number of faces in the base mesh.

9.6.1.2.3.2 U32: Base Position Count

Base Position Count is the number of positions used by the base mesh in the position array.

9.6.1.2.3.3 U32: Base Normal Count

Base Normal Count is the number of normals used by the base mesh in the normal array.

9.6.1.2.3.4 U32: Base Diffuse Color Count

Base Diffuse Color Count is the number of colors used by the base mesh in the diffuse color array.

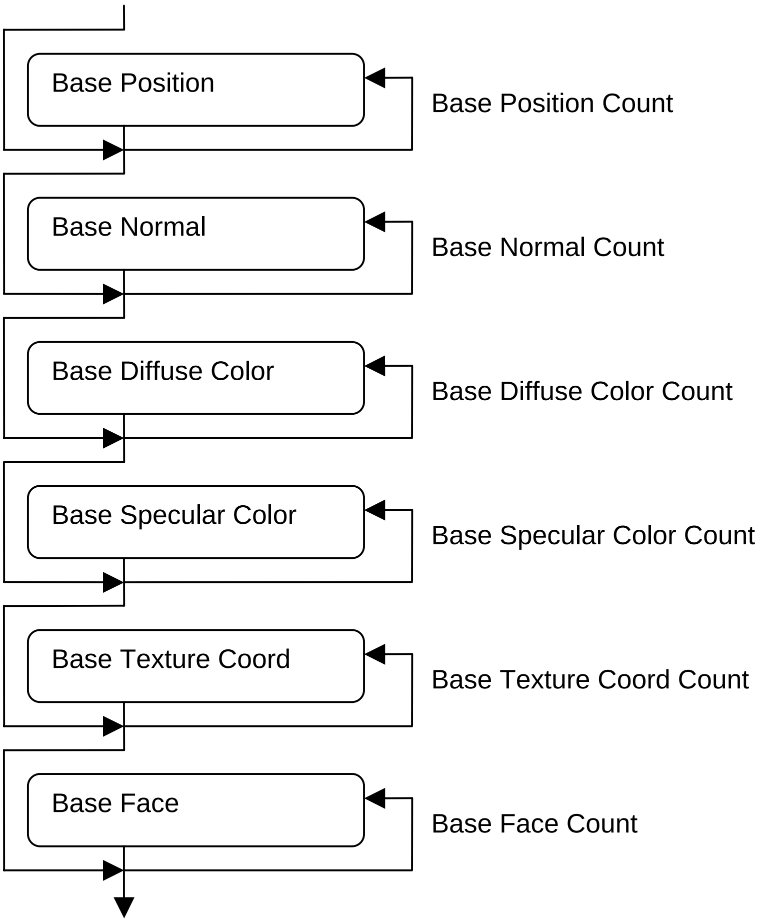
9.6.1.2.3.5 U32: Base Specular Color Count

Base Specular Color Count is the number of colors used by the base mesh in the specular color array.

9.6.1.2.3.6 U32: Base Texture Coord Count

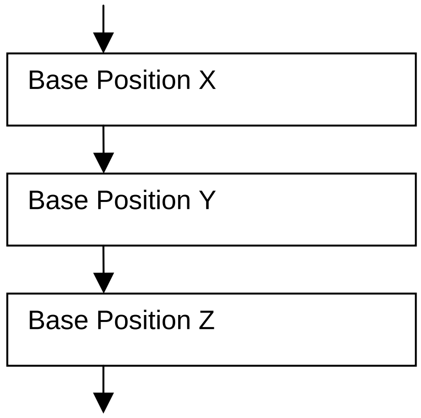
Base Texture Coord Count is the number of texture coordinates used by the base mesh in the texture coordinate array.

9.6.1.2.4 Base Mesh Data



9.6.1.2.4.1 Base Position

Base Position is a 3D position in the position array.



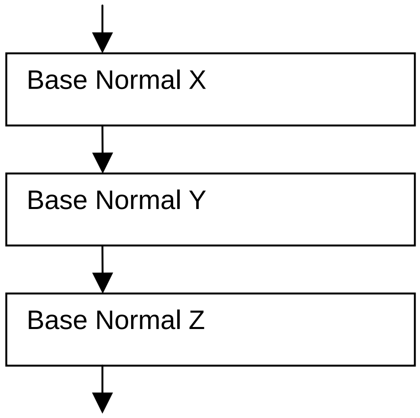
9.6.1.2.4.1.1 F32: Base Position X

9.6.1.2.4.1.2 F32: Base Position Y

9.6.1.2.4.1.3 F32: Base Position Z

9.6.1.2.4.2 Base Normal

Base Normal is a 3D normal in the normal array.



9.6.1.2.4.2.1 F32: Base Normal X

9.6.1.2.4.2.2 F32: Base Normal Y

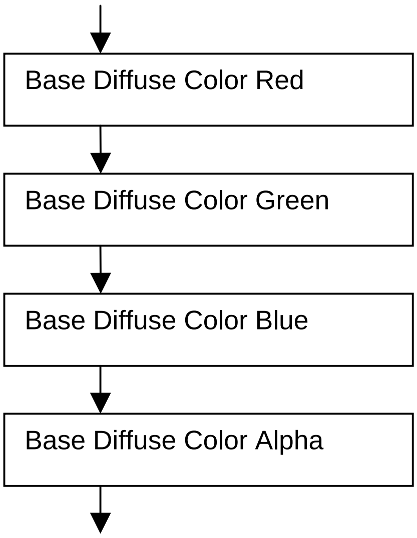
9.6.1.2.4.2.3 F32: Base Normal Z

9.6.1.2.4.3 Base Diffuse Color

Base Diffuse Color is an RGBA color in the diffuse color array.

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.



9.6.1.2.4.3.1 F32: Base Diffuse Color Red

9.6.1.2.4.3.2 F32: Base Diffuse Color Green

9.6.1.2.4.3.3 F32: Base Diffuse Color Blue

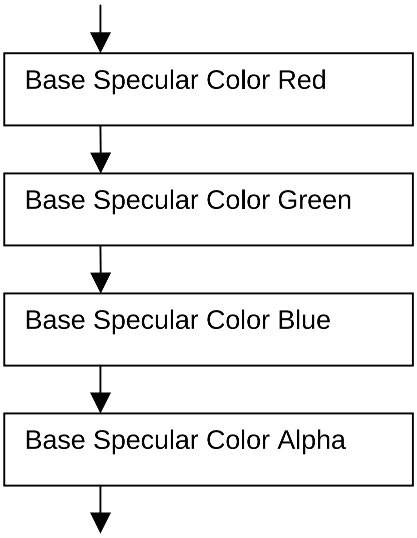
9.6.1.2.4.3.4 F32: Base Diffuse Color Alpha

9.6.1.2.4.4 Base Specular Color

Base Specular Color is an RGBA color in the specular color array.

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.



9.6.1.2.4.4.1 F32: Base Specular Color Red

9.6.1.2.4.4.2 F32: Base Specular Color Green

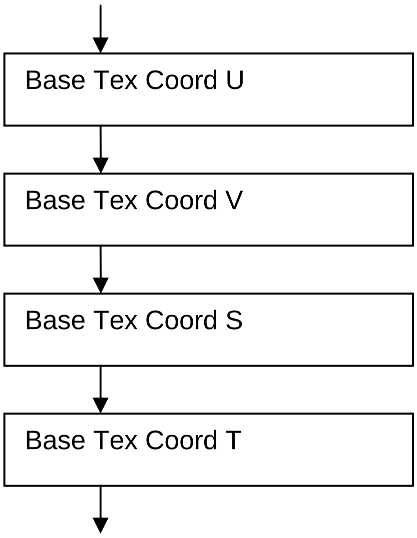
9.6.1.2.4.4.3 F32: Base Specular Color Blue

9.6.1.2.4.4.4 F32: Base Specular Color Alpha

9.6.1.2.4.5 Base Texture Coord

Base Texture Coord is a 4D texture coordinate in the texture coordinate array.

The shading list description may define a texture coordinate layer to have 1, 2, 3 or 4 dimension texture coordinates. For 1D texture coordinate layers, only the U value is used. For 2D texture coordinate layers, the U and V values are used. For 3D texture coordinate layers, the U, V ,and S layers are used. For 4D texture coordinate layers, the U, V, S and T layers are used.



9.6.1.2.4.5.1 F32: Base Tex Coord U

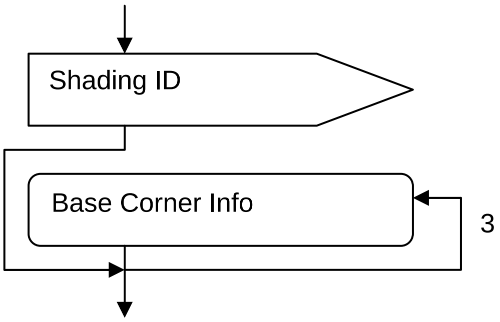
9.6.1.2.4.5.2 F32: Base Tex Coord V

9.6.1.2.4.5.3 F32: Base Tex Coord S

9.6.1.2.4.5.4 F32: Base Tex Coord T

9.6.1.2.4.6 Base Face

Base Face is a face in the base mesh. The face contains an index into the shading list description array and indices into the various mesh arrays for each corner.

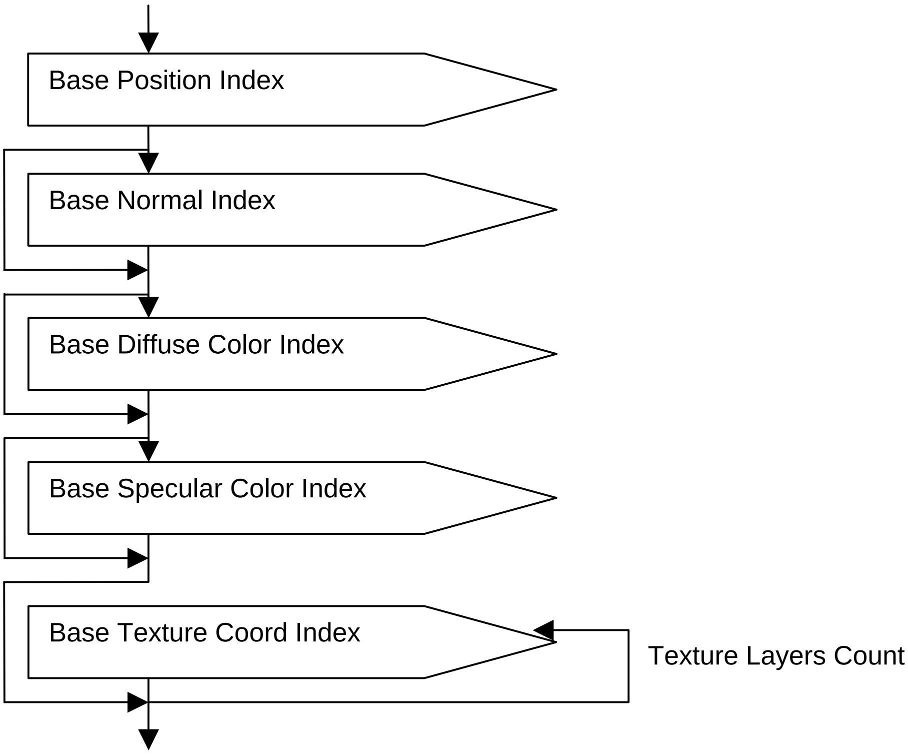


9.6.1.2.4.6.1 U32 [cShading]: Shading ID

Shading ID is the index of the shading list descriptions used for this face. The Shading List Description array is defined in the CLOD Mesh Declaration block.

9.6.1.2.4.6.2 Base Corner Info

Base Corner Info contains the indices into the various mesh arrays for a corner of a face in the base mesh. The indices are limited to the sizes in Base Mesh Description.



9.6.1.2.4.6.2.1 U32 [rBasePositionCount]: Base Position Index

Base Position Index must be less than Base Position Count in the Base Mesh Description.

9.6.1.2.4.6.2.2 U32 [rBaseNormalCount]: Base Normal Index

Base Normal Index must be less than Base Normal Count in the Base Mesh Description. Base Normal Index is not present if 9.6.1.1.3.1 Mesh Attributes in the Max Mesh Description indicates Exclude Normals.

9.6.1.2.4.6.2.3 U32 [rBaseDiffColorCnt]: Base Diffuse Color Index

Base Diffuse Color Index must be less than Base Diffuse Color Count in Base Mesh Description. Base Diffuse Color Index is present only if shading list description indicated by Shading ID indicates diffuse colors are used.

9.6.1.2.4.6.2.4 U32 [rBaseSpecColorCnt]: Base Specular Color Index

Base Specular Color Index must be less than Base Specular Color Count in Base Mesh Description. Base Specular Color Index is present only if shading description indicated by Shading ID indicates specular colors are used.

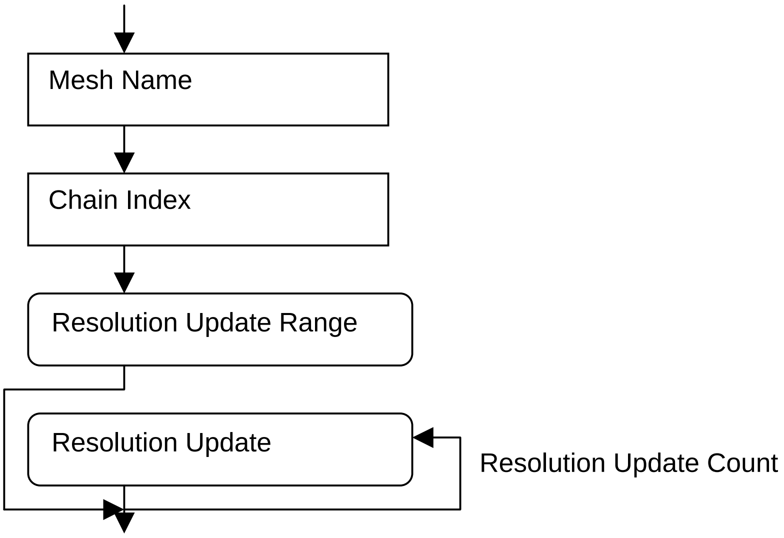
9.6.1.2.4.6.2.5 U32 [rBaseTexCoordCnt]: Base Texture Coord Index

Base Texture Coord Index must be less than Base Texture Coord Count in Base Mesh Description. Texture Layer Count in the shading description indicated by Shading ID determines the number of times Base Texture Coord Index in repeated at this corner.

9.6.1.3 CLOD Progressive Mesh Continuation (blocktype: 0xFFFFFF3C)

The CLOD Mesh Progressive Continuation block contains progressive mesh information for a continuous level of detail mesh generator.

The CLOD Mesh Progressive Continuation block is a continuation type block. The CLOD Mesh Progressive Continuation block is present only if Final Maximum Resolution is greater than Minimum Resolution.



9.6.1.3.1 String: Mesh Name

Mesh Name is the name of the CLOD mesh generator. This name is also the name of the model resource modifier chain that contains the CLOD mesh generator.

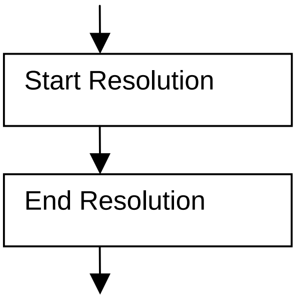
9.6.1.3.2 U32: Chain Index

Chain Index is the position of the CLOD mesh generator in the model resource modifier chain. The value of Chain Index shall be zero for this blocktype.

9.6.1.3.3 Resolution Update Range

Resolution Update Range specifies the range of progressive mesh vertex updates provided in this continuation block.

This continuation block contains CLOD mesh information for positions from (Start Resolution) to (End Resolution – 1). The total number of positions added by this block is Resolution Update Count = End Resolution – Start Resolution.



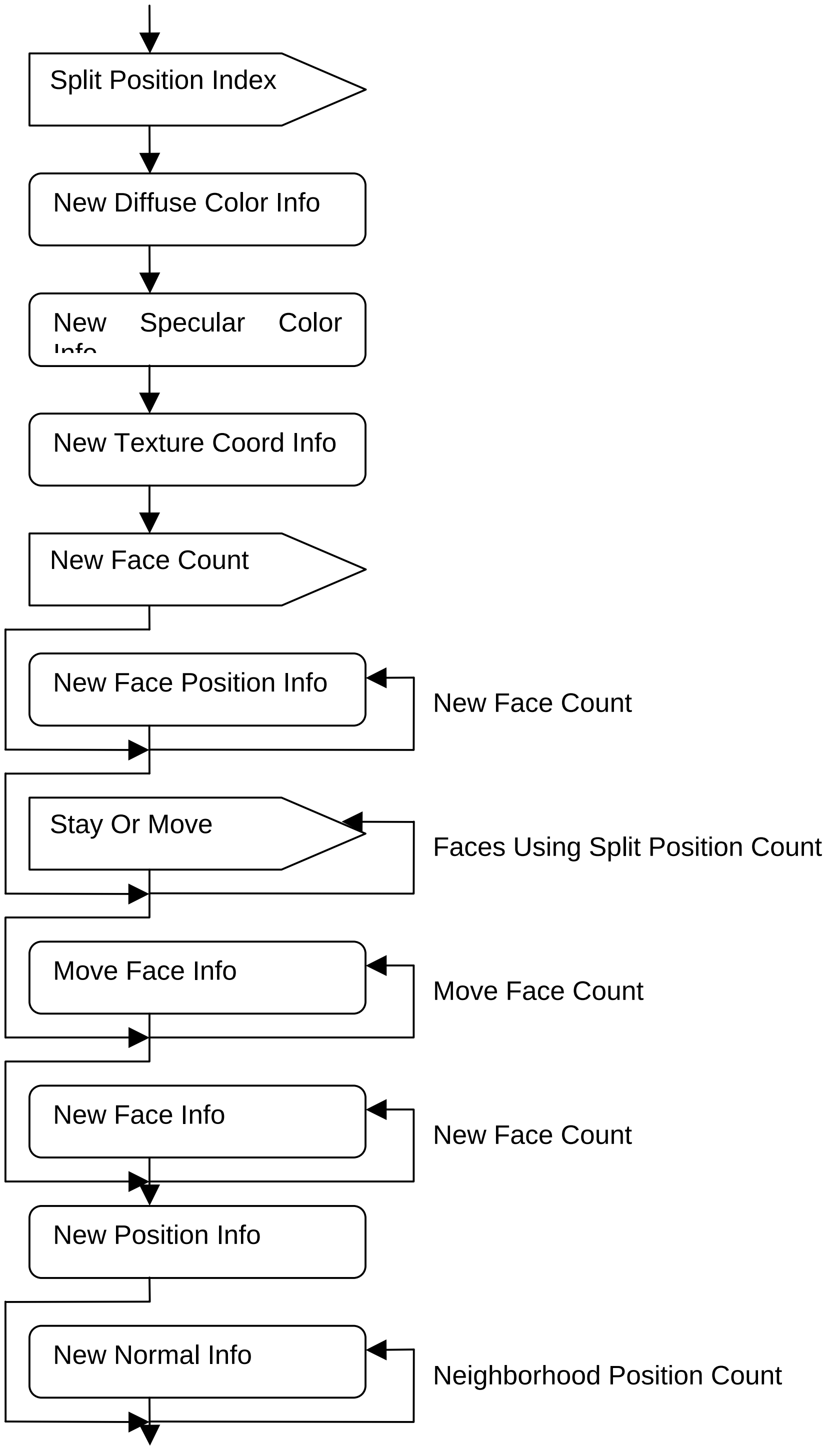
9.6.1.3.3.1 U32: Start Resolution

Start Resolution is the index of the first position added by this block.

9.6.1.3.3.2 U32: End Resolution

End Resolution is one more than the index of the last position added by this block.

9.6.1.3.4 Resolution Update

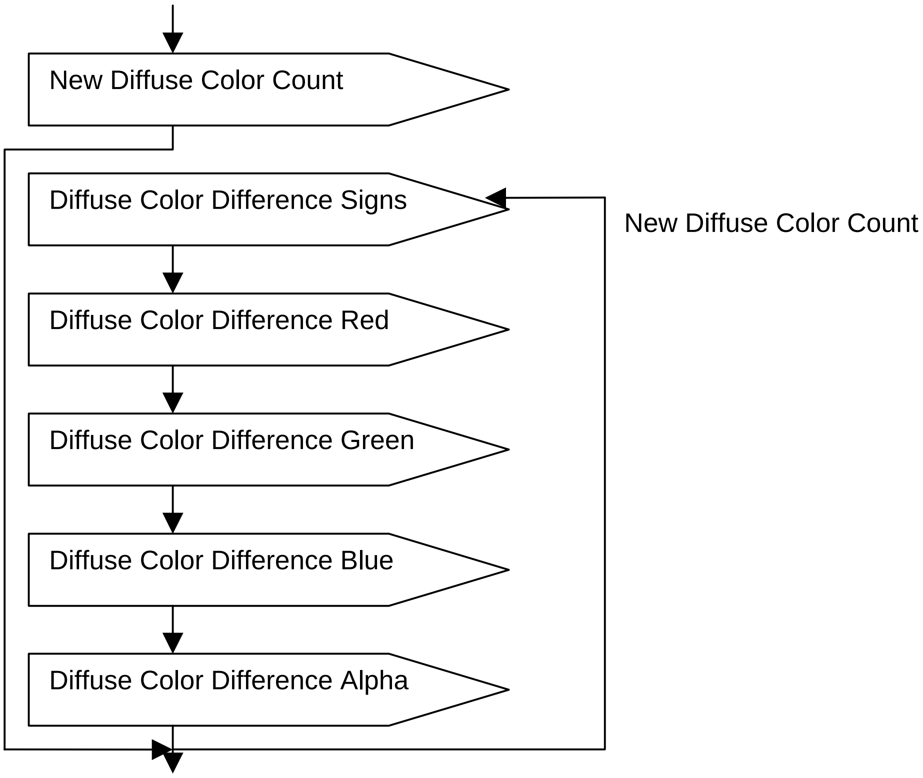


9.6.1.3.4.1 U32 [rCurrentPositionCount]: Split Position Index

Split Position Index is the index of the position to be split by this Resolution Update. Each Resolution Update adds one new position to the position array. Split Position Index will be less than the current position count. Each new face added by a Resolution Update will use the split position and the new position. Each face updated by a Resolution Update will change the split position to the new position. The new position is predicted based on the split position. The method for selecting the split position index is implementation dependent.

9.6.1.3.4.2 New Diffuse Color Info

New Diffuse Color Info describes new color values added to the diffuse color array of the mesh in this resolution update. The prediction for the new diffuse color value is calculated as the average of all diffuse color values used at the split position.


New Diffuse Color Count
Diffuse Color Difference Signs
New Diffuse Color Count
Diffuse Color Difference Red
Diffuse Color Difference Green
Diffuse Color Difference Blue
Diffuse Color Difference Alpha

9.6.1.3.4.2.1 U32[cDiffuseCount]: NewDiffuse Color Count

New Diffuse Color Count is the number of new color values added in this resolution update.

9.6.1.3.4.2.2 U8 [cDiffuseColorSign]: Diffuse Color Difference Signs

Diffuse Color Difference Signs is a collection of sign bits for the prediction differences.

0x01 – Sign bit for Diffuse Color Difference Red

0x02 – Sign bit for Diffuse Color Difference Green

0x04 – Sign bit for Diffuse Color Difference Blue

0x08 – Sign bit for Diffuse Color Difference Alpha

9.6.1.3.4.2.3 U32[cColorDiffR]: Diffuse Color Difference Red

Reconstructed Color red component is calculated as

Reconstructed Red = InverseQuant(predicted red,(Diffuse Color Difference Signs & 0x01),

Diffuse Color Difference Red, Diffuse Color Inverse Quant).

9.6.1.3.4.2.4 U32 [cColorDiffG]: Diffuse Color Difference Green

Reconstructed Color green component is calculated as

Reconstructed Green = InverseQuant(predicted green, ((Diffuse Color Difference Signs & 0x02) >> 1),

Diffuse Color Difference Green, Diffuse Color Inverse Quant).

9.6.1.3.4.2.5 U32 [cColorDiffB]: Diffuse Color Difference Blue

Reconstructed Color blue component is calculated as

Reconstructed Blue = InverseQuant(predicted blue, ((Diffuse Color Difference Signs & 0x04) >> 2),

Diffuse Color Difference Blue, Diffuse Color Inverse Quant).

9.6.1.3.4.2.6 U32 [cColorDiffA]: Diffuse Color Difference Alpha

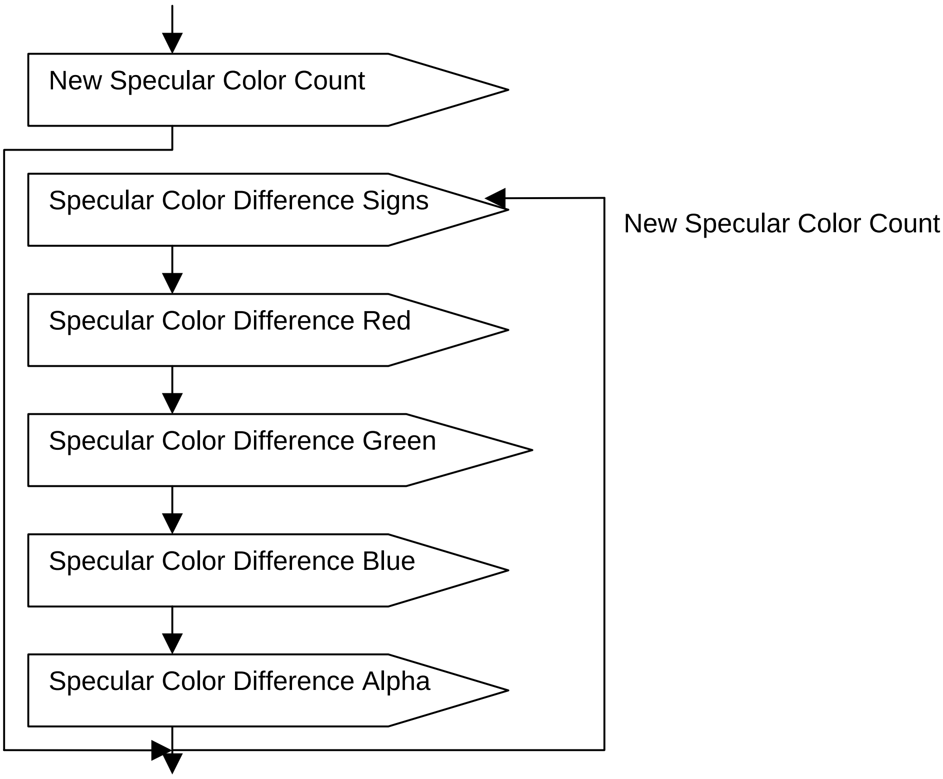
Reconstructed Color alpha component is calculated as

Reconstructed Alpha = InverseQuant(predicted alpha, ((Diffuse Color Difference Signs & 0x08) >> 3)

Diffuse Color Difference Alpha, Diffuse Color Inverse Quant).

9.6.1.3.4.3 New Specular Color Info

New Specular Color Info describes new color values added to the specular color array of the mesh in this resolution update. The prediction for the new specular color value is calculated as the average of all specular color values used at the split position.


New Specular Color Count
Specular Color Difference Signs
New Specular Color Count
Specular Color Difference Red
Specular Color Difference Green
Specular Color Difference Blue
Specular Color Difference Alpha

9.6.1.3.4.3.1 U32[cSpecularCount]: New Specular Color Count

New Specular Color Count is the number of new color values added in this resolution update.

9.6.1.3.4.3.2 U8 [cSpecularColorSign]: Specular Color Difference Signs

Specular Color Difference Signs is a collection of sign bits for the prediction differences.

0x01 – Sign bit for Specular Color Difference Red

0x02 – Sign bit for Specular Color Difference Green

0x04 – Sign bit for Specular Color Difference Blue

0x08 – Sign bit for Specular Color Difference Alpha

9.6.1.3.4.3.3 U32[cColorDiffR]: Specular Color Difference Red

Reconstructed Color red component is calculated as

Reconstructed Red = InverseQuant(predicted red, (Specular Color Difference Signs & 0x01),

Specular Color Difference Red, Specular Color Inverse Quant).

9.6.1.3.4.3.4 U32 [cColorDiffG]: Specular Color Difference Green

Reconstructed Color green component is calculated as

Reconstructed Green = InverseQuant(predicted green, ((Specular Color Difference Signs & 0x02) >> 1)),

Specular Color Difference Green, Specular Color Inverse Quant).

9.6.1.3.4.3.5 U32 [cColorDiffB]: Specular Color Difference Blue

Reconstructed Color blue component is calculated as

Reconstructed Blue = InverseQuant(predicted blue, ((Specular Color Difference Signs & 0x04) >> 2)),

Specular Color Difference Blue, Specular Color Inverse Quant).

9.6.1.3.4.3.6 U32 [cColorDiffA]: Specular Color Difference Alpha

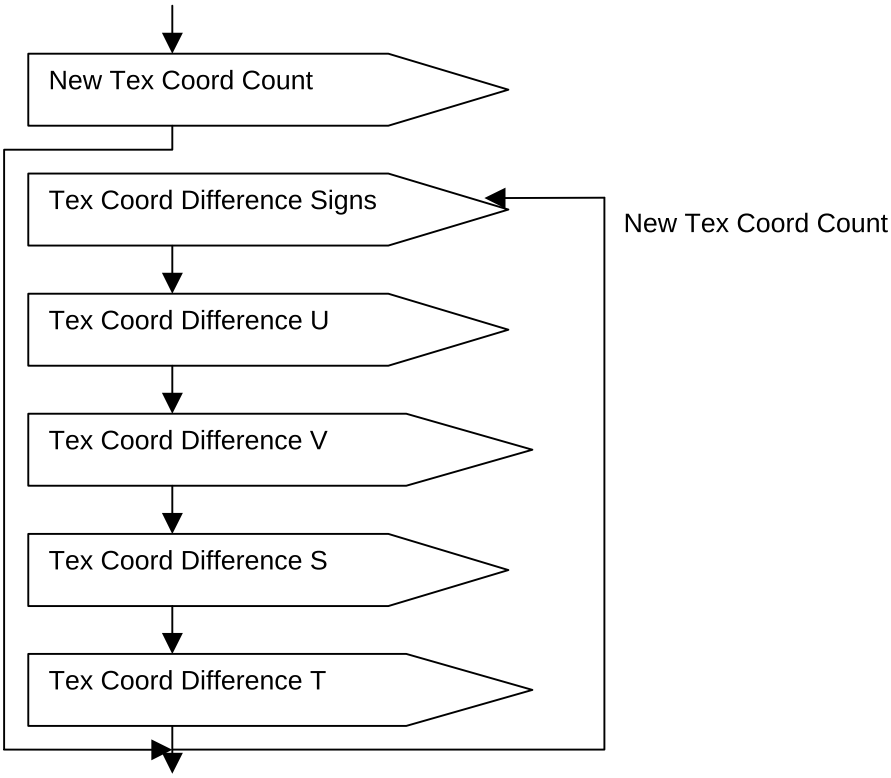
Reconstructed Color alpha component is calculated as

Reconstructed Alpha = InverseQuant(predicted alpha, ((Specular Color Difference Signs & 0x08) >> 3)),

Specular Color Difference Alpha, Specular Color Inverse Quant).

9.6.1.3.4.4 New Texture Coord Info

New Texture Coord Info describes new texture coordinate values added to the texture coordinate array of the mesh in this resolution update. The prediction for the new texture coordinate value is calculated as the average of all texture coordinates used at the split position in the first layer.



9.6.1.3.4.4.1 U32[cTexCoordCount]: New Tex Coord Count

New Tex Coord Count is the number of new texture coordinate values added in this resolution update.

9.6.1.3.4.4.2 U8 [cTexCoordSign]: Tex Coord Difference Signs

Tex Coord Difference Signs is a collection of sign bits for the prediction differences.

0x01 – Sign bit for Texture Coord Difference U

0x02 – Sign bit for Texture Coord Difference V

0x04 – Sign bit for Texture Coord Difference S

0x08 – Sign bit for Texture Coord Difference T

9.6.1.3.4.4.3 U32 [cTexCDiffU]: Texture Coord Difference U

The reconstructed texture coordinate U is calculated as

> Reconstructed TexCoord U = InverseQuant(predicted Tex Coord U, (Tex Coord Signs & 0x01),
>
> Texture Coord Difference U, Texture Coord Inverse Quant).

9.6.1.3.4.4.4 U32 [cTexCDiffV]: Texture Coord Difference V

The reconstructed texture coordinate V is calculated as

> Reconstructed TexCoord V = InverseQuant(predicted Tex Coord V, ((Tex Coord Signs & 0x02) >> 1),
>
> Texture Coord Difference V, Texture Coord Inverse Quant).

9.6.1.3.4.4.5 U32 [cTexCDiffS]: Texture Coord Difference S

The reconstructed texture coordinate S is calculated as

Reconstructed TexCoord S = InverseQuant(predicted Tex Coord S, ((Tex Coord Signs & 0x04) >> 2),

Texture Coord Difference S, Texture Coord Inverse Quant).

9.6.1.3.4.4.6 U32 [cTexCDiffT]: Texture Coord Difference T

The reconstructed texture coordinate T is calculated as

Reconstructed TexCoord T = InverseQuant(predicted Tex Coord, ((Tex Coord Signs & 0x08) >> 3),

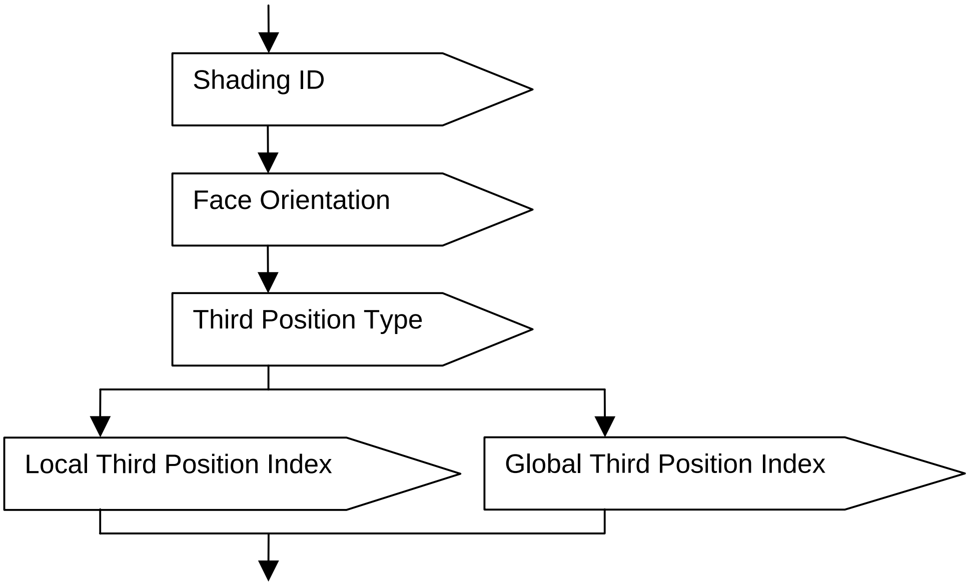
Texture Coord Difference T, Texture Coord Inverse Quant).

9.6.1.3.4.5 U32 [cFaceCnt]: New Face Count

New Face Count is the number of new faces added to the mesh by this Resolution Update.

9.6.1.3.4.6 New Face Position Info

New Face Position Info describes a new face to be added to the mesh. One of the corners of the new face will use the Split Position and another of the corners will use the New Position.



9.6.1.3.4.6.1 U32 [cShading]: Shading ID

Shading ID is the index of the shading list used for this face. The Shading Description array is defined in the CLOD Mesh Declaration block.

9.6.1.3.4.6.2 U8 [cFaceOrnt]: Face Orientation

Face Orientation refers to the winding order of the face.

0x00 – Left Orientation: Split Position; New Position; Third Position

0x01 – Right Orientation: New Position; Split Position; Third Position

9.6.1.3.4.6.3 U8 [cThrdPosType]: Third Position Type

Third Position Type indicates whether the Third Position Index that follows is an index into the full position array or a smaller local position array.

0x00 – Local Third Position Index

0x01 – Global Third Position Index

9.6.1.3.4.6.4 U32 [cLocal3rdPos]: Local Third Position Index

The local position array is generated by adding all positions used by faces that also use the Split Position. Each position is added only once to the local position array. The local position array contains indices into the full position array. The local position array is sorted with the smaller values first.

9.6.1.3.4.6.5 U32 [rCurrentPositionCount]: Global Third Position Index

Global Third Position Index is an index into the full position array. Current Position Count is the number of positions in the full position array.

9.6.1.3.4.7 U8 [cStayMove+StayMovePrediction]: Stay Or Move

For each face that was using the Split Position, Stay Or Move specifies if that face should continue to use the Split Position or be updated to use the New Position.

0x00 – Stay; Continue to use the Split Position

0x01 – Move; Update face to use the New Position

The compression context depends on whether the face is predicted to stay or move. Valid values for StayMovePrediction are:

0 – No prediction

1 – Predict stay from use of third position by face

2 – Predict move from use of third position by face

3 – Predict stay from prediction used for neighboring face

4 – Predict move from prediction used for neighboring face

For faces that use the split position and one third position from a new face: if the corner winding order puts the split position before the third position, then predict stay if the new face orientation was right or predict move if the new face orientation was left; if the corner winding order puts the split position after the third position, then predict stay if the new face orientation was left or predict move if the new face orientation was right.

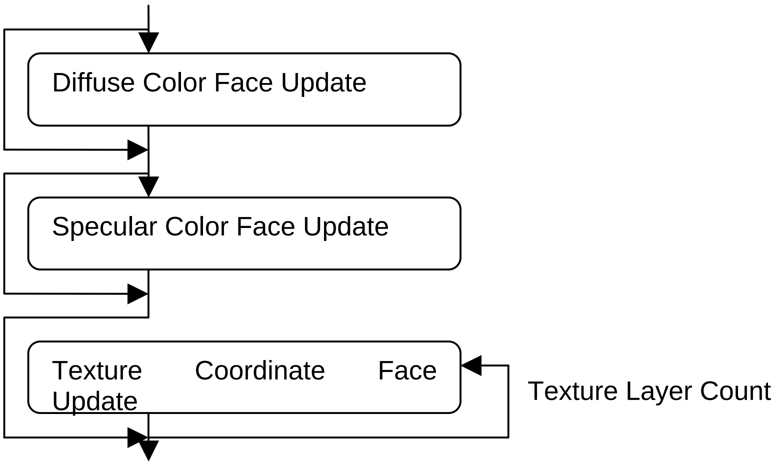
For faces that use the split position and do not use a third position from a new face: if a neighboring face has been predicted stay or move, then use that prediction for this face.

If there is a conflict where one neighboring face is predicted stay and another neighboring face is predicted move, then make no prediction.

For faces where the preceding prediction rules do not apply, make no prediction.

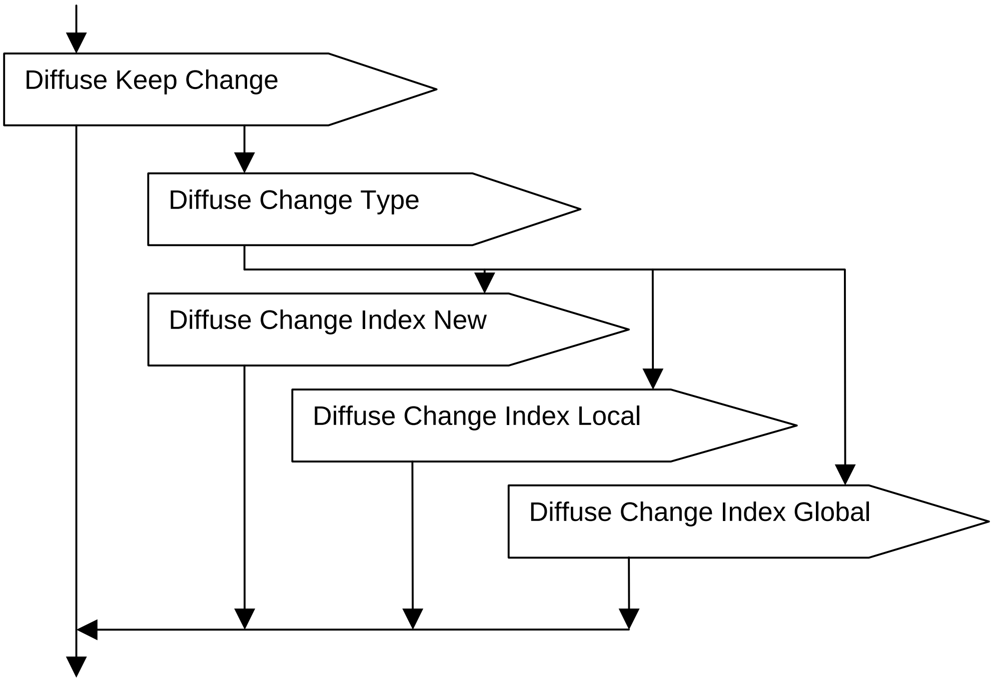
9.6.1.3.4.8 Move Face Info

The move faces are the faces for which one of the corners changes from using the split position to using the new position. For each face of the move faces, the other properties at that corner may also change. Move Face Info describes how those corners of move faces should be updated for vertex color and texture coordinate properties.


Diffuse Color Face Update
Specular Color Face Update
Texture Coordinate Face Update
Texture Layer Count

9.6.1.3.4.8.1 Diffuse Color Face Update

Diffuse Color Face Update is present only if the shading description for this face indicates that the face has diffuse colors at the corners of the face.



9.6.1.3.4.8.1.1 U8 [cDiffuseKeepChange]: Diffuse Keep Change

For each face with diffuse colors and a split position that is moving to the New Position, Diffuse Keep Change indicates whether the diffuse color at the same corner as the split position should also change.

0x00 – Keep; The diffuse color should not change.

0x01 – Change; The diffuse color should change. The new value for the diffuse color index can be found in the following change index.

9.6.1.3.4.8.1.2 U8 [cDiffuseChangeType]: Diffuse Change Type

Diffuse Change Type indicates the type of change index that follows.

0x01 – New;

0x02 – Local;

0x03 – Global;

9.6.1.3.4.8.1.3 U32 [cDiffuseChangeIndexNew]: Diffuse Change Index New

Diffuse Change Index New is an index into the list of new diffuse colors for this resolution update as described in 9.6.1.3.4.2 New Diffuse Color Info.

9.6.1.3.4.8.1.4 U32 [cDiffuseChangeIndexLocal]: Diffuse Change Index Local

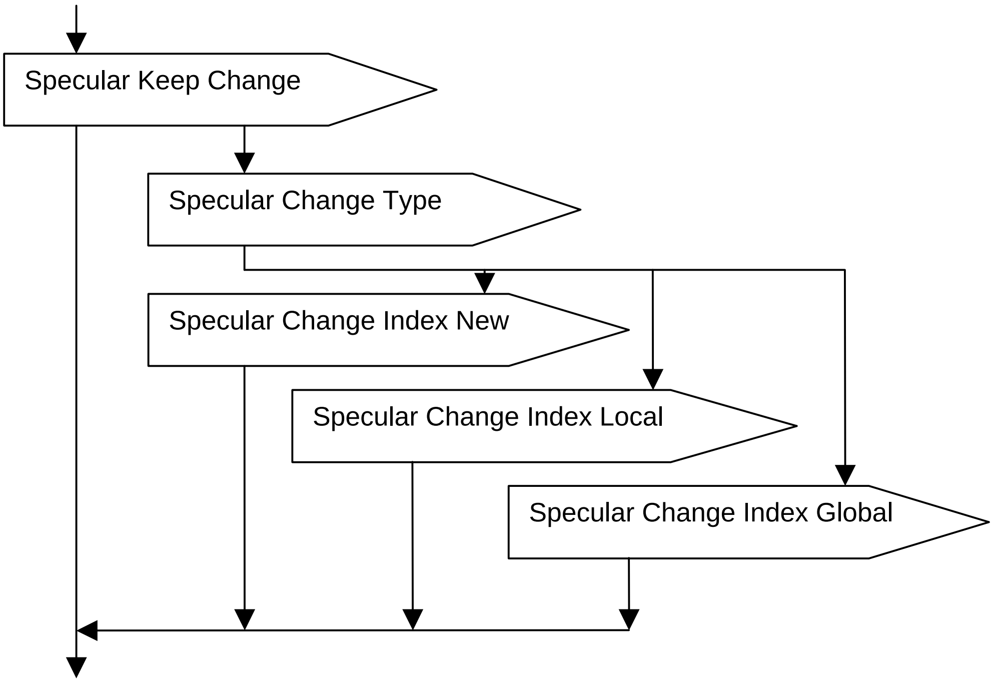
Diffuse Change Index Local is an index into the list of diffuse color indices used at the split position. Larger indices appear first in the that list.

9.6.1.3.4.8.1.5 U32 [cDiffuseChangeIndexGlobal]: Diffuse Change Index Global

Diffuse Change Index Global is an index into the full diffuse color pool.

9.6.1.3.4.8.2 Specular Color Face Update

Specular Color Face Update is present only if the shading description for this face indicates that the face has specular colors at the corners of the face.


Specular Keep Change
Specular Change Type
Specular Change Index New
Specular Change Index Local
Specular Change Index Global

9.6.1.3.4.8.2.1 U8 [cSpecularKeepChange]: Specular Keep Change

For each face with specular colors and a split position that is moving to the New Position, Specular Keep Change indicates whether the specular color at the same corner as the split position should also change.

0x00 – Keep; The specular color should not change.

0x01 – Change; The specular color should change. The new value for the specular color index can be found in the following change index.

9.6.1.3.4.8.2.2 U8 [cSpecularChangeType]: Specular Change Type

Specular Change Type indicates the type of change index that follows.

0x01 – New;

0x02 – Local;

0x03 – Global;

9.6.1.3.4.8.2.3 U32 [cSpecularChangeIndexNew]: Specular Change Index New

Specular Change Index New is an index into the list of new specular colors for this resolution update as described in 9.6.1.3.4.3 New Specular Color Info.

9.6.1.3.4.8.2.4 U32 [cSpecularChangeIndexLocal]: Specular Change Index Local

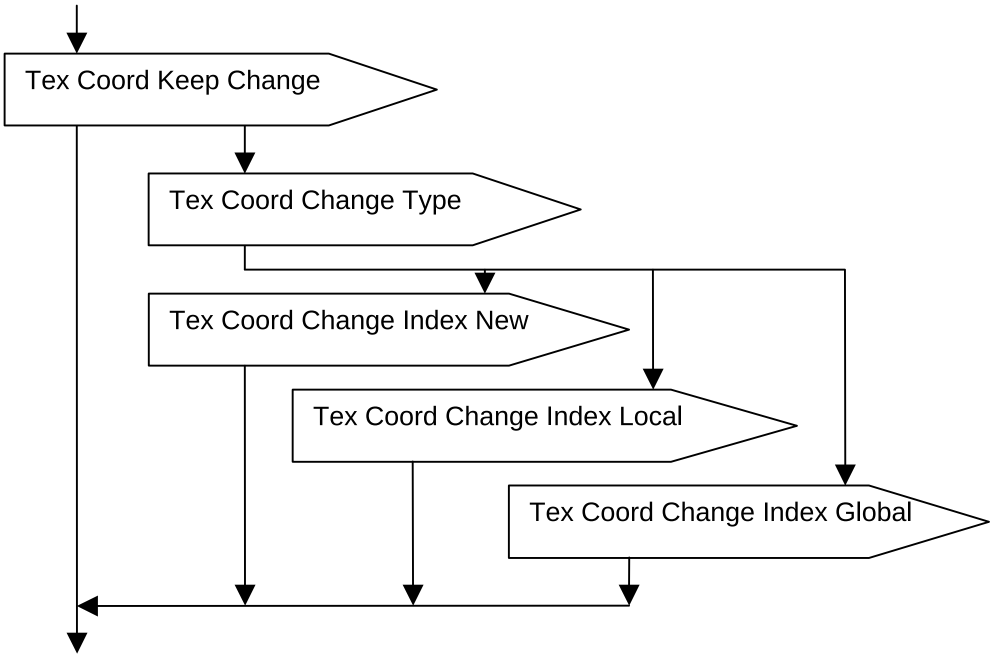
Specular Change Index Local is an index into the list of specular color indices used at the split position. Larger indices appear first in the that list.

9.6.1.3.4.8.2.5 U32 [cSpecularChangeIndexGlobal]: Specular Change Index Global

Specular Change Index Global is an index into the full specular color pool.

9.6.1.3.4.8.3 Texture Coordinate Face Update

Texture Coordinate Face Update is repeated once for each texture layer for the move face. If there are no texture layers, then Texture Coordinate Face Update is not present.


Tex Coord Keep Change
Tex Coord Change Type
Tex Coord Change Index New
Tex Coord Change Index Local
Tex Coord Change Index Global

9.6.1.3.4.8.3.1 U8 [cTCKeepChange]: Tex Coord Keep Change

For each face with texture coordinates and a split position that is moving to the New Position, Tex Coord Keep Change indicates whether the texture coordinate at the same corner as the split position should also change.

0x00 – Keep; The texture coordinate should not change.

0x01 – Change; The texture coordinate should change. The new value for the texture coordinate can be found in the following change index.

9.6.1.3.4.8.3.2 U8 [cTCChangeType]: Tex Coord Change Type

Tex Coord Change Type indicates the type of change index that follows.

0x01 – New;

0x02 – Local;

0x03 – Global;

9.6.1.3.4.8.3.3 U32 [cTCChangeIndexNew]: Tex Coord Change Index New

Tex Coord Change Index New is an index into the list of new texture coordinates for this resolution update as described in 9.6.1.3.4.4 New Texture Coord Info.

9.6.1.3.4.8.3.4 U32 [cTCChangeIndexLocal]: Tex Coord Change Index Local

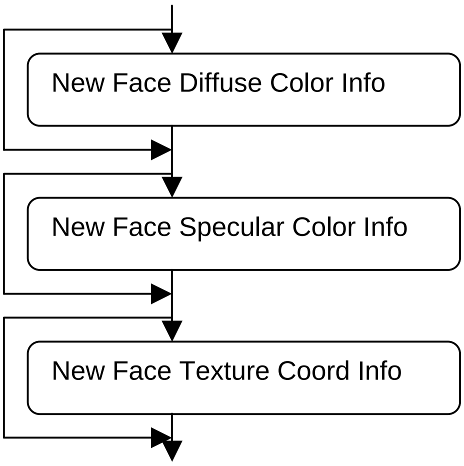
Tex Coord Change Index Local is an index into the list of texture coordinate indices used at the split position at this texture layer. Larger indices appear first in the that list.

9.6.1.3.4.8.3.5 U32 [cTCChangeIndexGlobal]: Tex Coord Change Index Global

Tex Coord Change Index Global is an index into the full texture coordinate pool.

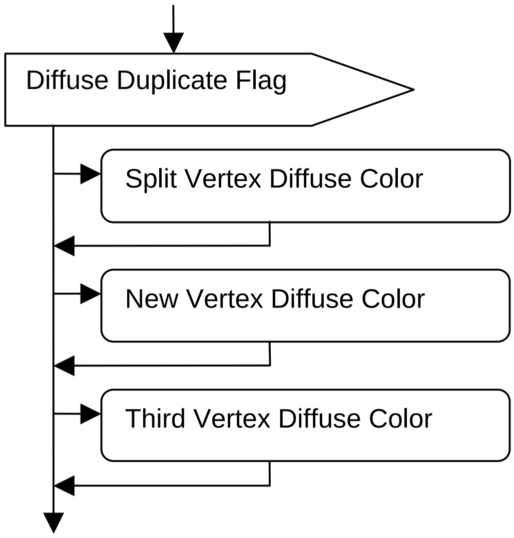
9.6.1.3.4.9 New Face Info

New Face Info completes the description of the new faces to be added to the mesh. The description was started in 9.6.1.3.4.6 New Face Position Info. The presence of the diffuse, specular or texture coordinate face info is determined by the Shading Description indicated by the Shading ID for this new face.


New Face Diffuse Color Info
New Face Specular Color Info
New Face Texture Coord Info

9.6.1.3.4.9.1 New Face Diffuse Color Info

New Face Diffuse Color Info indicates which colors from the diffuse color pool are used at each corner of this face. New Face Diffuse Color Info is only present if the shading list identified by Shading ID uses diffuse color coordinates. One of the Shading Attributes flags indicates the presence of diffuse color coordinates.



9.6.1.3.4.9.1.1 U8 [cColorDup]: Diffuse Duplicate Flag

Diffuse Duplicate Flag is a set of flags that indicates if the index for the color at a particular corner is the same as the corresponding index from the previous diffuse face. If the flag is set (one), then the most recently used color is used again. If the flag is not set (zero), then a color index is used to indicate the color to be used from the diffuse color pool. All other values are reserved.

0x00 – Split Vertex uses color indicated by the color index.

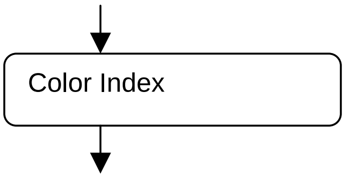
0x01 – Split Vertex uses color used at previous diffuse split vertex.

0x02 – New Vertex uses color used at previous diffuse new vertex.

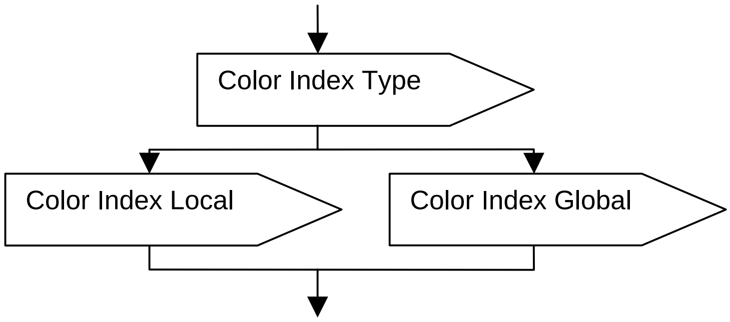
0x04 – Third Vertex uses color used at previous diffuse third vertex.

9.6.1.3.4.9.1.2 Split Vertex Diffuse Color

Split Vertex Diffuse Color is present only if Diffuse Duplicate Flag indicates the Split Vertex does not use a duplicate color. This color index uses the list of diffuse color indices used at the split position for the local index list. This color index uses the diffuse color pool.



9.6.1.3.4.9.1.2.1 Color Index



9.6.1.3.4.9.1.2.1.1 U8[cColorIndexType]: Color Index Type

Color Index Type indicates whether the following index is an index into the complete color pool or an index into a smaller local list of color indices.

0x02 – Local

0x03 – Global

9.6.1.3.4.9.1.2.1.2 U32[cColorIndexLocal]: Color Index Local

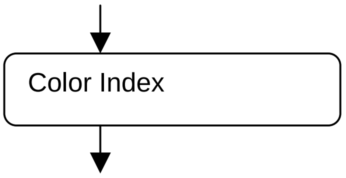
Color Index Local is an index into a local list of colors. The indices in the list are sorted with the larger indices first.

9.6.1.3.4.9.1.2.1.3 U32[cColorIndexGlobal]: Color Index Global

Color Index Global is an index into the complete color pool.

9.6.1.3.4.9.1.3 New Vertex Diffuse Color

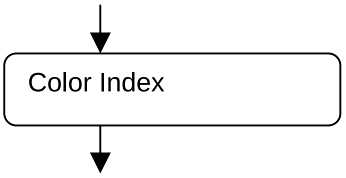
New Vertex Diffuse Color is present only if Diffuse Duplicate Flag indicates the New Vertex does not use a duplicate color. This color index uses the list of diffuse color indices used at the split position for the local index list. This color index uses the diffuse color pool for the complete color pool.



Details on the color index format are in 9.6.1.3.4.9.1.2.1 Color Index.

9.6.1.3.4.9.1.4 Third Vertex Diffuse Color

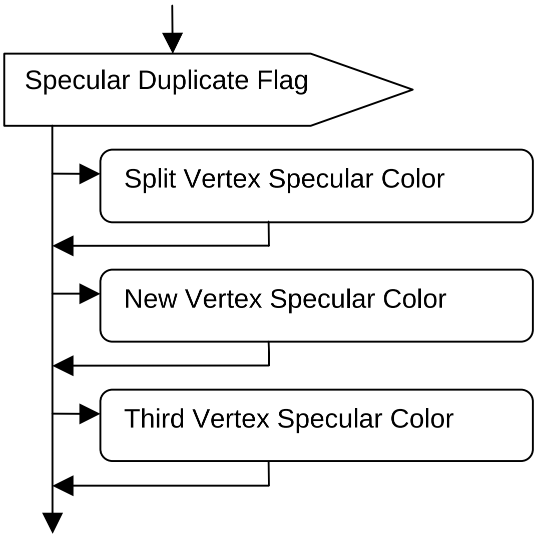
Third Vertex Diffuse Color is present only if Diffuse Duplicate Flag indicates the Third Vertex does not use a duplicate color. This color index uses the list of diffuse color indices used at the third position for the local index list. This color index uses the diffuse color pool for the complete color pool.



Details on the color index format are in 9.6.1.3.4.9.1.2.1 Color Index.

9.6.1.3.4.9.2 New Face Specular Color Info

New Face Specular Color Info indicates which colors from the specular color pool are used at each corner of this face. New Face Specular Color Info is only present if the shading list identified by Shading ID uses specular color coordinates. One of the Shading Attributes flags indicates the presence of specular color coordinates.


Specular Duplicate Flag
Split Vertex Specular Color
New Vertex Specular Color
Third Vertex Specular Color

9.6.1.3.4.9.2.1 U8 [cColorDup]: Specular Duplicate Flag

Specular Duplicate Flag is a set of flags that indicates if the index for the color at a particular corner is the same as the corresponding index from the previous specular face. If the flag is set (one), then the most recently used color is used again. If the flag is not set (zero), then a color index is used to indicate the color to be used from the specular color pool. All other values are reserved.

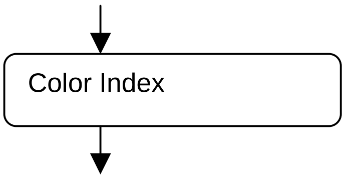
0x01 – Split Vertex uses color used at previous specular split vertex.

0x02 – New Vertex uses color used at previous specular new vertex.

0x04 – Third Vertex uses color used at previous specular third vertex.

9.6.1.3.4.9.2.2 Split Vertex Specular Color

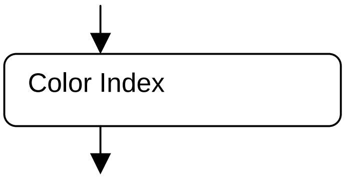
Split Vertex Specular Color is present only if Specular Duplicate Flag indicates the Split Vertex does not use a duplicate color. This color index uses the list of specular color indices used at the split position for the local index list. This color index uses the specular color pool for the complete color pool.



Details on the color index format are in 9.6.1.3.4.9.1.2.1 Color Index.

9.6.1.3.4.9.2.3 New Vertex Specular Color

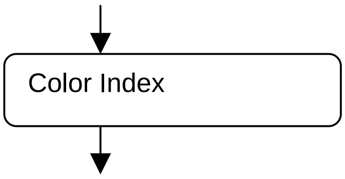
New Vertex Specular Color is present only if Specular Duplicate Flag indicates the New Vertex does not use a duplicate color. This color index uses the list of specular color indices used at the split position for the local index list. This color index uses the specular color pool for the complete color pool.



Details on the color index format are in 9.6.1.3.4.9.1.2.1 Color Index.

9.6.1.3.4.9.2.4 Third Vertex Specular Color

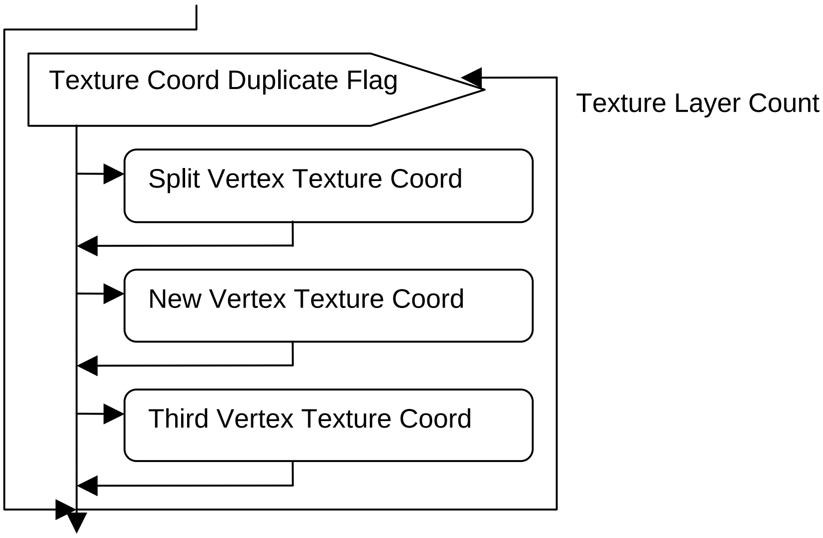
Third Vertex Specular Color is present only if Specular Duplicate Flag indicates the Third Vertex does not use a duplicate color. This color index uses the list of specular color indices used at the third position for the local index list. This color index uses the specular color pool for the complete color pool.



Details on the color index format are in 9.6.1.3.4.9.1.2.1 Color Index.

9.6.1.3.4.9.3 New Face Texture Coord Info

New Face Texture Coord Info indicates which texture coordinates from the texture coordinate pool are used at each corner of this face for each texgture coordinate layer. New Face Texture Coord Info is only present if the shading list identified by Shading ID uses texture coordinates. A texture layer count greater than zero indicates the presence of texture coordinates.


Texture Coord Duplicate Flag
Texture Layer Count
Split Vertex Texture Coord
New Vertex Texture Coord
Third Vertex Texture Coord

9.6.1.3.4.9.3.1 U8 [cTexCDup]: Texture Coord Duplicate Flag

Texture Coord Duplicate Flag is a set of flags that indicates if the index for the texture coordinagte at a particular corner is the same as the corresponding index from the previous corner. If the flag is set (one), then the most recently used texture coordinate is used again. If the flag is not set (zero), then a texture coordinate index is used to indicate the texture coordinate to be used from the texture coordinate pool. All other values are reserved.

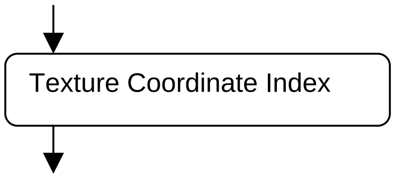
0x01 – Split Vertex uses previous texture coordinate used at a split vertex in this mesh.

0x02 – New Vertex uses previous texture coordinate used at a new vertex in this mesh.

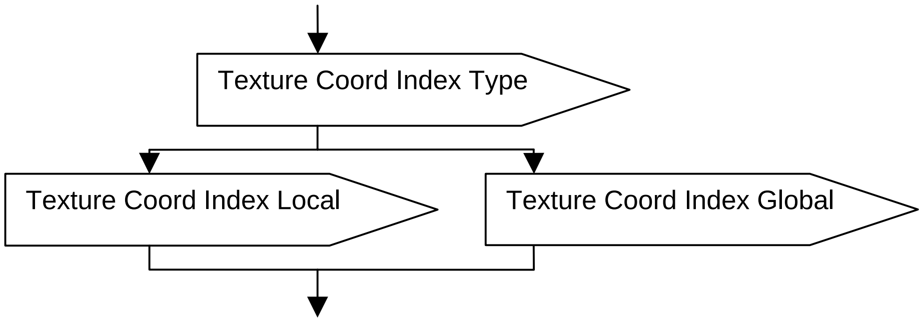
0x04 – Third Vertex uses previous texture coordinate used at a third vertex in this mesh.

9.6.1.3.4.9.3.2 Split Vertex Texture Coordinate

Split Vertex Texture Coordinate is present only if Texture Coord Duplicate Flag indicates the Split Vertex does not use a duplicate texture coordinate. This texture coordinate index uses the list of texture coordinate indices used at the split position for the local index list.



9.6.1.3.4.9.3.2.1 Texture Coordinate Index



9.6.1.3.4.9.3.2.1.1 U8[cTexCIndexType]: Texture Coord Index Type

Texture Coord Index Type indicates whether the following index is an index into the complete texture coordinate pool or an index into a smaller local list of texture coordinate indices.

0x02 – Local

0x03 – Global

9.6.1.3.4.9.3.2.1.2 U32[cColorIndexLocal]: Texture Coord Index Local

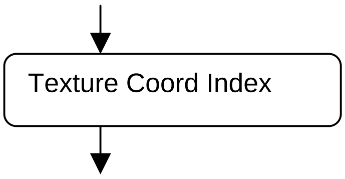
Texture Coord Index Local is an index into a local list of texture coordinates. The indices in the list are sorted with the larger indices first.

9.6.1.3.4.9.3.2.1.3 U32[cColorIndexGlobal]: Texture Coord Index Global

Texture Coord Index Global is an index into the complete texture coordinate pool.

9.6.1.3.4.9.3.3 New Vertex Texture Coord

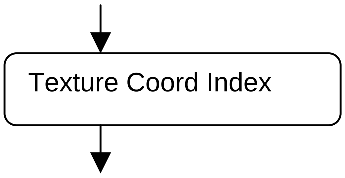
New Vertex Texture Coord is present only if Texture Coord Duplicate Flag indicates the New Vertex does not use a duplicate texture coordinate. This texture coordinate index uses the list of texture coordinates indices used at the split position for the current texture layer for the local index list.



Details on the texture coordinate index format are in 9.6.1.3.4.9.3.2.1 Texture Coord Index.

9.6.1.3.4.9.3.4 Third Vertex Texture Coord

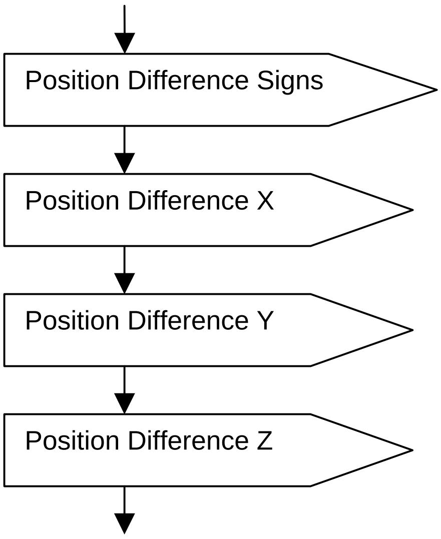
Third Vertex Texture Coord is present only if Texture Coord Duplicate Flag indicates the Third Vertex does not use a duplicate texture coordinate. This texture coordinate index uses the list of texture coordinate indices used at the third position for the current texture layer for the local index list. This color index uses the diffuse color pool for the complete color pool.



Details on the texture coordinate index format are in 9.6.1.3.4.9.3.2.1 Texture Coord Index.

9.6.1.3.4.10 New Position Info

New Position Info describes the position added to the mesh during this resolution upate. The new position value is predicted as the split position value.



9.6.1.3.4.10.1 U8 [cPosDiffSign]: Position Difference Signs

Position Difference Signs is a collection of sign bits for the prediction differences.

0x01 – Sign bit for Position Difference X

0x02 – Sign bit for Position Difference Y

0x04 – Sign bit for Position Difference Z

9.6.1.3.4.10.2 U32 [cPosDiffX]: Position Difference X

Reconstructed Position X coordinate is calculated as

Reconstructed Position X = InverseQuant(split position X,(Position Difference Signs & 0x01),

Position Difference X, Position Inverse Quant).

9.6.1.3.4.10.3 U32 [cPosDiffY]: Position Difference Y

Reconstructed Position Y coordinate is calculated as

Reconstructed Position Y = InverseQuant(split position Y,((Position Difference Signs & 0x02)>>1),

Position Difference Y, Position Inverse Quant).

9.6.1.3.4.10.4 U32 [cPosDiffZ]: Position Difference Z

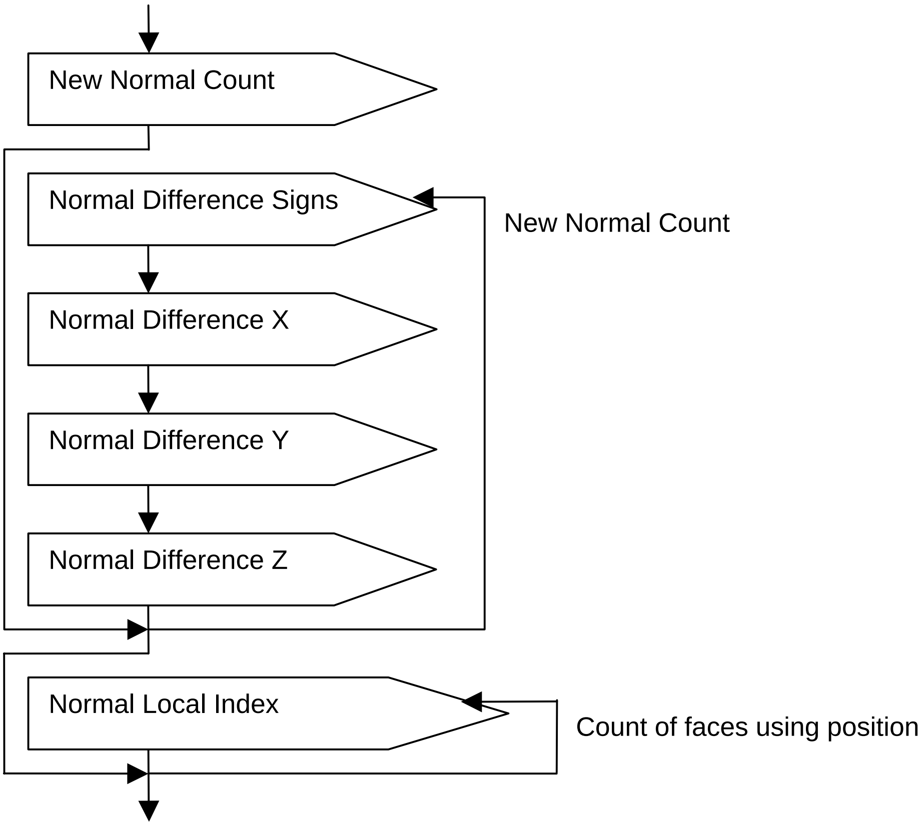
Reconstructed Position Z coordinate is calculated as

Reconstructed Position Z = InverseQuant(split position Z,((Position Difference Signs & 0x04)>>2),

Position Difference Z, Position Inverse Quant).

9.6.1.3.4.11 New Normal Info

New Normal Info is repeated once for each position in the neighbourhood of the new position. Positions with higher index values are handled first. This neighbourhood includes the new position. New Normal Info describes new normals added to the normal pool. New Normal Info also specifies which normal should be used by each corner that uses that position. New Normal Info is not present if 9.6.1.1.3.1 Mesh Attributes in the Max Mesh Description indicates Exclude Normals.



9.6.1.3.4.11.1 U32 [cNormlCnt]: New Normal Count

New Normal Count is the number of normals added to the normal array. An array of predicted normals is generated and the difference from the predictions is quantized and encoded in the following sections. To generate the array of predicted normals, start by putting the face normal for each face that uses this position into an array. While the size of this array is larger than New Normal Count, merge the two normals that are closest. Merging normals is done using a weighted spherical-linear average where each normal is weighted by the number of original face normals that it includes.

9.6.1.3.4.11.2 U8 [cDiffNormalSign]: Normal Difference Signs

Normal Difference Signs is a collection of sign bits for the prediction differences.

0x01 – Sign bit for Normal Difference X

0x02 – Sign bit for Normal Difference Y

0x04 – Sign bit for Normal Difference Z

9.6.1.3.4.11.3 U32 [cDiffNormalX]: Normal Difference X

Reconstructed Normal X coordinate is calculated as

Reconstructed Normal X = InverseQuant(predicted normal X,(Normal Difference Signs & 0x01),
Normal Difference X, Normal Inverse Quant).

9.6.1.3.4.11.4 U32 [cDiffNormalY]: Normal Difference Y

Reconstructed Normal Y coordinate is calculated as

Reconstructed Normal Y = InverseQuant(predicted normal Y,((Normal Difference Signs & 0x02)>>1),

Normal Difference Y, Normal Inverse Quant).

9.6.1.3.4.11.5 U32 [cDiffNormalZ]: Normal Difference Z

Reconstructed Normal Z coordinate is calculated as

Reconstructed Normal Z = InverseQuant(predicted normal Z,((Normal Difference Signs & 0x04)>>2),

Normal Difference Z, Normal Inverse Quant).

9.6.1.3.4.11.6 U32 [cNormlIdx]: Normal Local Index

For each face using the position, the face shall use a New Normal from the new normal array at the corner that is using the position. Normal Local Index specifies which of the new normals should be used. The face may be a new face added during this resolution update or may be a face that already existed in the mesh.

9.6.2 Point Set (blocktypes: 0xFFFFFF36; 0xFFFFFF3E)

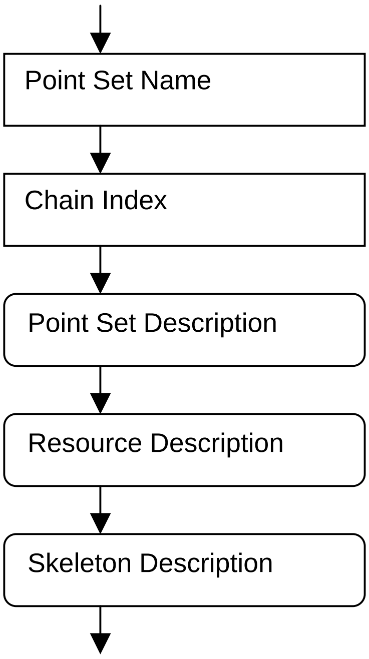
The Point Set generator contains the data needed to represent a set of points.

The Point Set produces the following outputs: Renderable Group, Renderable Group Bounds, Transform Set.

The Point Set's outputs have no dependencies.

9.6.2.1 Point Set Declaration (blocktype: 0xFFFFFF36)

The Point Set Declaration contains the declaration information for a point set generator. The declaration information is sufficient to allocate space for the point set data and create the point set generator object. The point set data is contained in following continuation blocks.



9.6.2.1.1 String: Point Set Name

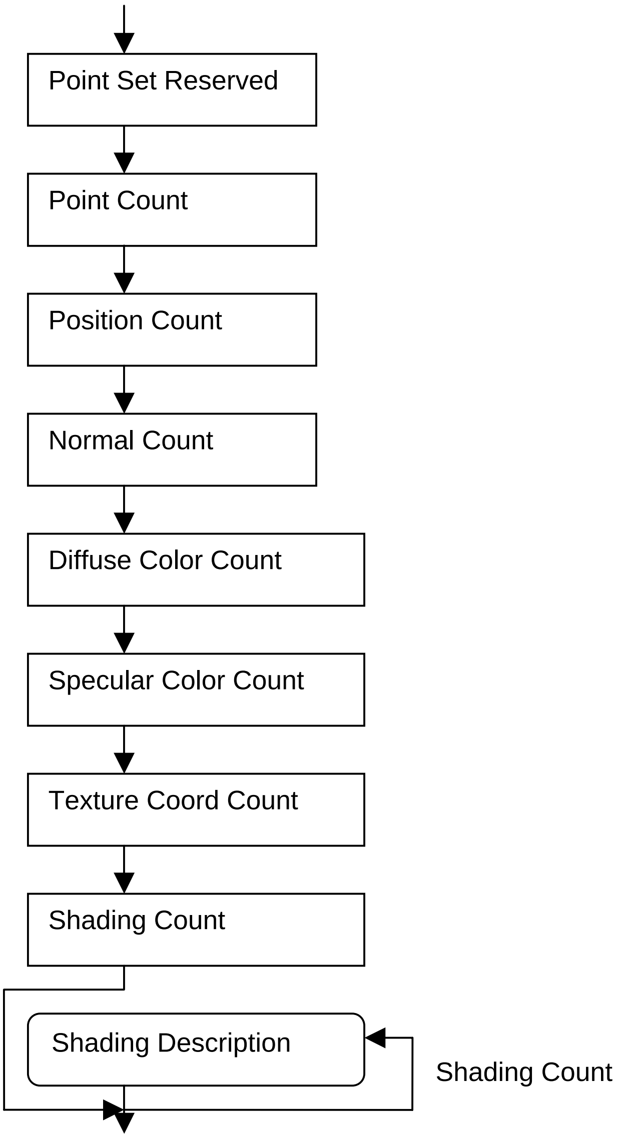
Point Set Name is the name of the point set generator. This name is also the name of the model resource modifier chain that contains the point set generator.

9.6.2.1.2 U32: Chain Index

Chain Index is the position of the point set generator in the model resource modifier chain.The value of Chain Index shall zero for this blocktype.

9.6.2.1.3 Point Set Description

Point Set Description describes the size of the point set. Point Set Description can be used to allocate space for the point set.



9.6.2.1.3.1 U32: Point Set Reserved

Point Set Reserved is a reserved field and shall have the value 0.

9.6.1.1.3.2 U32: Point Count

Point Count is the number of points in the point set.

9.6.2.1.3.3 U32: Position Count

Position Count is the number of positions in the position array.

9.6.2.1.3.4 U32: Normal Count

Normal Count is the number of normals in the normal array.

9.6.2.1.3.5 U32: Diffuse Color Count

Diffuse Color Count is the number of colors in the diffuse color array.

9.6.2.1.3.6 U32: Specular Color Count

Specular Color Count is the number of colors in the specular color array.

9.6.2.1.3.7 U32: Texture Coord Count

Texture Coord Count is the number of texture coordinates in the texture coordinate array.

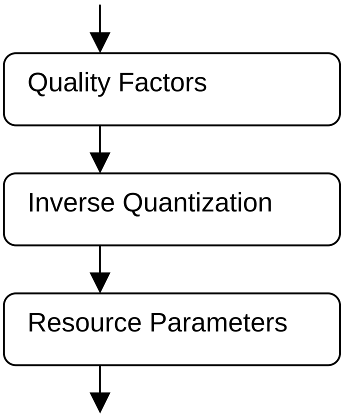
9.6.2.1.3.8 U32: Shading List Count

Shading List Count Count is the number of materials used in the point set.

9.6.2.1.3.9 Shading Description

Shading Description indicates which per vertex attributes, in addition to position and normal, are used by each shading list. Details are covered in 9.6.1.1.3.9 Shading Description.

9.6.2.1.4 Resource Description



9.6.2.1.4.1 Quality Factors

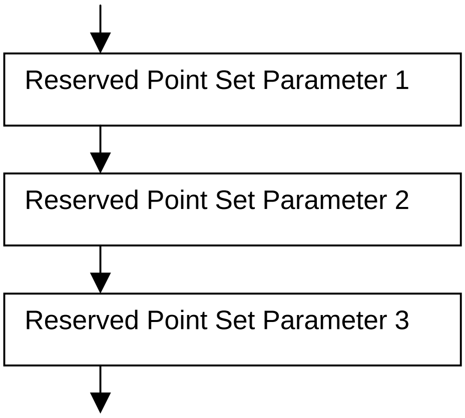
The quality factors are defined in 9.6.1.1.5.1 Quality Factors.

9.6.2.1.4.2 Inverse Quantization

Inverse quantization is defined in 9.6.1.1.5.2 Inverse Quantization.

9.6.2.1.4.3 Resource Parameters

Resource Parameters control the operation of the point set generator. The parameters defined in this section control the creation of the renderable point set. These parameters are reserved for future definition.


Reserved Point Set Parameter 1
Reserved Point Set Parameter 2
Reserved Point Set Parameter 3

9.6.2.1.4.3.1 U32: Reserved Point Set Parameter 1

Reserved Point Set Parameter 1 shall have the value 0.

9.6.2.1.4.3.2 U32: Reserved Point Set Parameter 2

Reserved Point Set Parameter 2 shall have the value 0.

9.6.2.1.4.3.3 U32: Reserved Point Set Parameter 3

Reserved Point Set Parameter 3 shall have the value 0.

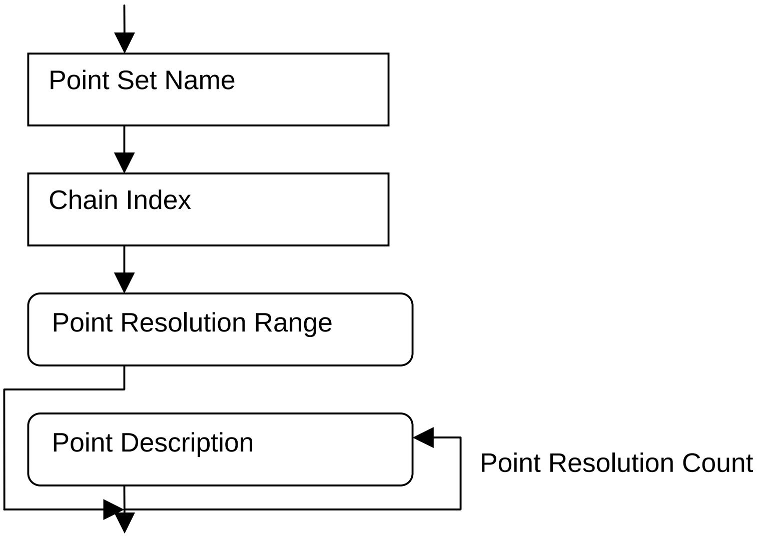
9.6.2.1.5 Skeleton Description

Skeleton Description provides bone structure information. Definition of Skeleton Description is in 9.6.1.1.6 Skeleton Description.

9.6.2.2 Point Set Continuation (blocktype: 0xFFFFFF3E)

The Point Set Continuation contains point data for a point set generator.

The Point Set Continuation block is a continuation type block.



9.6.2.2.1 String: Point Set Name

Point Set Name is the name of the point set generator. This name is also the name of the model resource modifier chain that contains the point set generator.

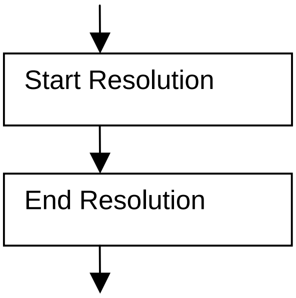
9.6.2.2.2 U32: Chain Index

Chain Index is the position of the point set generator in the model resource modifier chain. The value of Chain Index shall zero for this blocktype.

9.6.2.2.3 Point Resolution Range

Point Resolution Range specifies the range of point description data provided in this continuation block.

This continuation block contains point description data for positions from (Start Resolution) to (End Resolution – 1). The total number of positions added by this block is Point Resolution Count = End Resolution – Start Resolution.



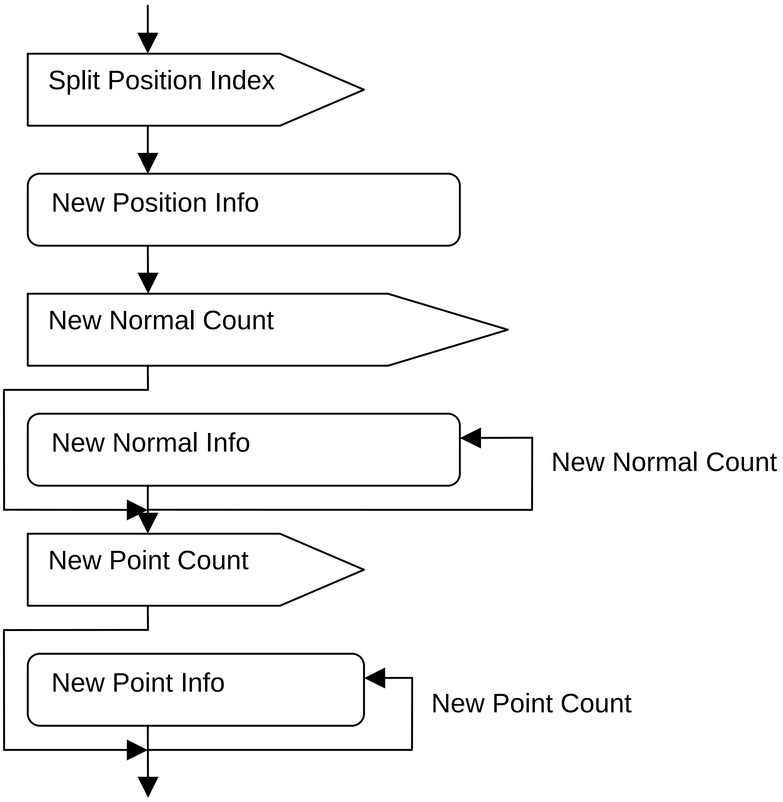
9.6.2.2.3.1 U32: Start Resolution

Start Resolution is the index of the first position added by this block.

9.6.2.2.3.2 U32: End Resolution

End Resolution is one more than the index of the last position added by this block.

9.6.2.2.4 Point Description



9.6.2.2.4.1 U32 [rCurrentPositionCount]: Split Position Index

Each Point Description adds one new position to the position array. Split Position Index is the index of the position of the points used as a prediction reference by this Point Description. Split Position Index will be less than the current position count.

9.6.2.2.4.2 New Position Info

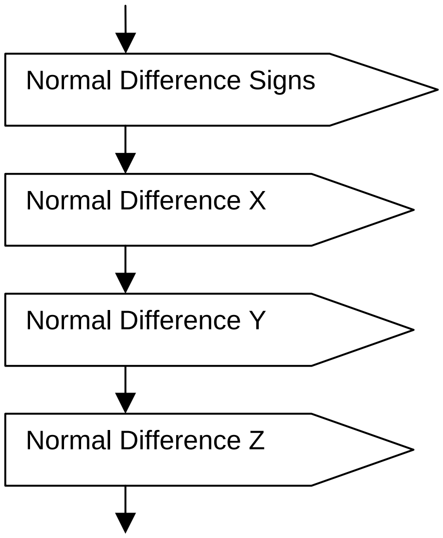
The new position is predicted as the split position. New Position Info is defined above in 9.6.1.3.4.10 New Position Info.

9.6.2.2.4.3 U32 [cNormlCnt]: New Normal Count

New Normal Count is the number of normals added to the normal array for use by points at this position.

9.6.2.2.4.4 New Normal Info

The normals are predicted as the spherical-linear average of the normals used by points at the split position.



9.6.2.2.4.4.1 U8 [cDiffNormalSign]: Normal Difference Signs

Normal Difference Signs is a collection of sign bits for the prediction differences.

0x01 – Sign bit for Normal Difference X

0x02 – Sign bit for Normal Difference Y

0x04 – Sign bit for Normal Difference Z

9.6.2.2.4.4.2 U32 [cDiffNormal]: Normal Difference X

Reconstructed Normal X coordinate is calculated as

Reconstructed Normal X = InverseQuant(predicted normal X,(Normal Difference Signs & 0x01),

Normal Difference X, Normal Inverse Quant).

9.6.2.2.4.4.3 U32 [cDiffNormal]: Normal Difference Y

Reconstructed Normal Y coordinate is calculated as

Reconstructed Normal Y = InverseQuant(predicted normal Y,((Normal Difference Signs & 0x02)>>1),

Normal Difference Y, Normal Inverse Quant).

9.6.2.2.4.4.4 U32 [cDiffNormal]: Normal Difference Z

Reconstructed Normal Z coordinate is calculated as

Reconstructed Normal Z = InverseQuant(predicted normal Z,((Normal Difference Signs & 0x04)>>2),

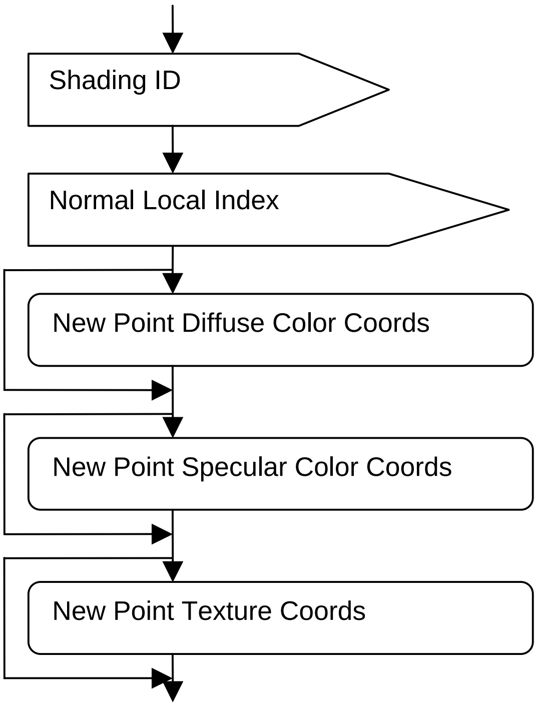
Normal Difference Z, Normal Inverse Quant).

9.6.2.2.4.5 U32 [cPointCnt]: New Point Count

New Point Count is the number of new points added to the point set by this Point Description.

9.6.2.2.4.6 New Point Info

New Point Info describes a new point to be added to the point set. The position index of the point is the current position count.


Shading ID
Normal Local Index
New Point Diffuse Color Coords
New Point Specular Color Coords
New Point Texture Coords

9.6.2.2.4.6.1 U32 [cShading]: Shading ID

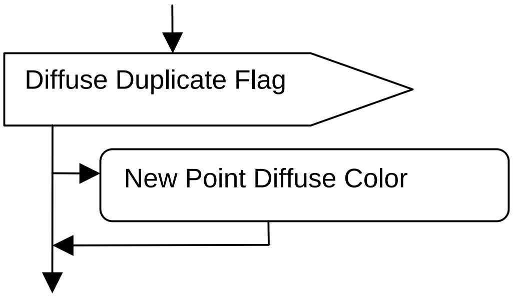
Shading ID is the index of the shading description used for this point. The Shading Description array is defined in the Point Set Declaration block.

9.6.2.2.4.6.2 U32 [cNormlIdx]: Normal Local Index

The new point shall use a normal the New Normal Info array for this point. Normal Local Index specifies which of the new normals should be used.

9.6.2.2.4.6.3 New Point Diffuse Color Coords

New Point Diffuse Color Coords is only present if the shading list identified by Shading ID uses diffuse color coordinates. One of the Shading Attributes flags indicates the presence of diffuse color coordinates.



9.6.2.2.4.6.3.1 U8 [cDiffDup]: Diffuse Duplicate Flag

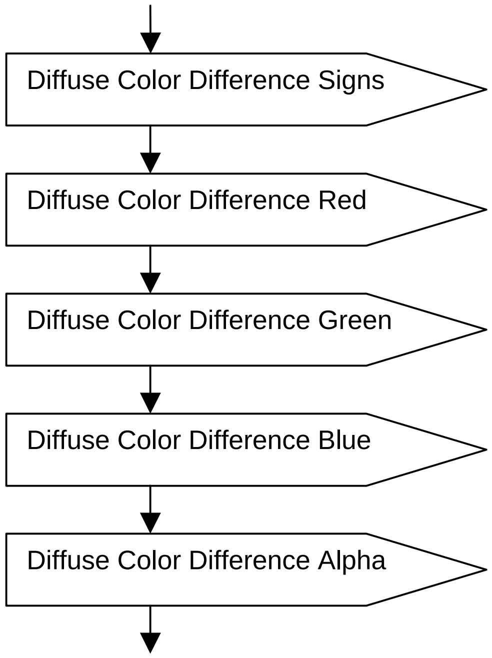
Diffuse Duplicate Flag is a set of flags that indicates if a new color is added to the color pool or if the most recently added color is used again. If the flag is set (one), then the most recently added color is used again. If the flag is not set (zero), then a new color is added to the diffuse color pool. All other values are reserved.

0x02 – New point uses duplicate color

9.6.2.2.4.6.3.2 New Point Diffuse Color

New Point Diffuse Color is present only if Diffuse Duplicate Flag indicates the new point does not use a duplicate color.

The New Point Diffuse Color is predicted as the average of the diffuse colors used by all points that use the Split Position.


Diffuse Color Difference Signs
Diffuse Color Difference Red
Diffuse Color Difference Green
Diffuse Color Difference Blue
Diffuse Color Difference Alpha

9.6.2.2.4.6.3.2.1 U8 [cDiffuseColorSign]: Diffuse Color Difference Signs

Diffuse Color Difference Signs is a collection of sign bits for the prediction differences.

0x01 – Sign bit for Diffuse Color Difference Red

0x02 – Sign bit for Diffuse Color Difference Green

0x04 – Sign bit for Diffuse Color Difference Blue

0x08 – Sign bit for Diffuse Color Difference Alpha

9.6.2.2.4.6.3.2.2 U32[cColorDiffR]: Diffuse Color Difference Red

Reconstructed Color red component is calculated as

Reconstructed Red = InverseQuant(predicted red,(Diffuse Color Difference Signs & 0x01),

Diffuse Color Difference Red, Diffuse Color Inverse Quant).

9.6.2.2.4.6.3.2.3 U32 [cColorDiffG]: Diffuse Color Difference Green

Reconstructed Color green component is calculated as

Reconstructed Green = InverseQuant(predicted green, ((Diffuse Color Difference Signs & 0x02) >> 1),

Diffuse Color Difference Green, Diffuse Color Inverse Quant).

9.6.2.2.4.6.3.2.4 U32 [cColorDiffB]: Diffuse Color Difference Blue

Reconstructed Color blue component is calculated as

Reconstructed Blue = InverseQuant(predicted blue, ((Diffuse Color Difference Signs & 0x04) >> 2),

Diffuse Color Difference Blue, Diffuse Color Inverse Quant).

9.6.2.2.4.6.3.2.5 U32 [cColorDiffA]: Diffuse Color Difference Alpha

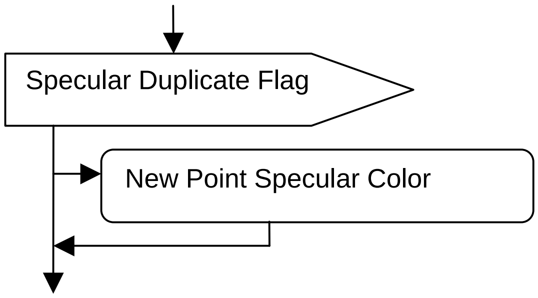
Reconstructed Color alpha component is calculated as

Reconstructed Alpha = InverseQuant(predicted alpha, ((Diffuse Color Difference Signs & 0x08) >> 3)

Diffuse Color Difference Alpha, Diffuse Color Inverse Quant).

9.6.2.2.4.6.4 New Point Specular Color Coords

New Point Specular Color Coords is only present if the shading list identified by Shading ID uses specular color coordinates. One of the Shading Attributes flags indicates the presence of specular color coordinates.



9.6.2.2.4.6.4.1 U8 [cSpecDup]: Specular Duplicate Flag

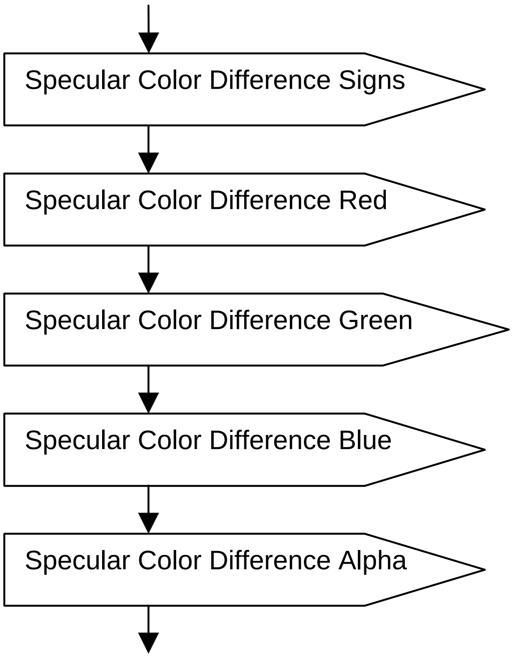
Specular Duplicate Flag is a set of flags that indicates if a new color is added to the color pool or if the most recently added color is used again. If the flag is set (one), then the most recently added color is used again. If the flag is not set (zero), then a new color is added to the specular color pool. All other values are reserved.

0x02 – New point uses duplicate color

9.6.2.2.4.6.4.2 New Point Specular Color

New Point Specular Color is present only if Specular Duplicate Flag indicates the new point does not use a duplicate color.

The New Point Specular Color is predicted as the average of the specular colors used at all points that use the Split Position.


Specular Color Difference Signs
Specular Color Difference Red
Specular Color Difference Green
Specular Color Difference Blue
Specular Color Difference Alpha

9.6.2.2.4.6.4.2.1 U8 [cSpecularColorSign]: Specular Color Difference Signs

Specular Color Difference Signs is a collection of sign bits for the prediction differences.

0x01 – Sign bit for Specular Color Difference Red

0x02 – Sign bit for Specular Color Difference Green

0x04 – Sign bit for Specular Color Difference Blue

0x08 – Sign bit for Specular Color Difference Alpha

9.6.2.2.4.6.4.2.2 U32[cColorDiffR]: Specular Color Difference Red

Reconstructed Color red component is calculated as

Reconstructed Red = InverseQuant(predicted red,(Specular Color Difference Signs & 0x01),

Specular Color Difference Red, Specular Color Inverse Quant).

9.6.2.2.4.6.4.2.3 U32 [cColorDiffG]: Specular Color Difference Green

Reconstructed Color green component is calculated as

Reconstructed Green = InverseQuant(predicted green, ((Specular Color Difference Signs & 0x02) >> 1),

Specular Color Difference Green, Specular Color Inverse Quant).

9.6.2.2.4.6.4.2.4 U32 [cColorDiffB]: Specular Color Difference Blue

Reconstructed Color blue component is calculated as

Reconstructed Blue = InverseQuant(predicted blue, ((Specular Color Difference Signs & 0x04) >> 2),

Specular Color Difference Blue, Specular Color Inverse Quant).

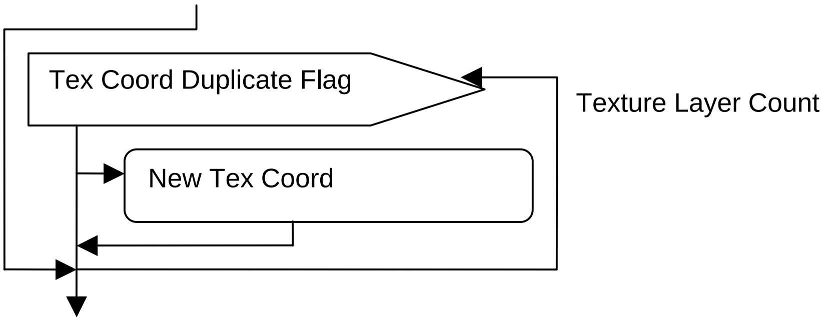
9.6.2.2.4.6.4.2.5 U32 [cColorDiffA]: Specular Color Difference Alpha

Reconstructed Color alpha component is calculated as

Reconstructed Alpha = InverseQuant(predicted alpha, ((Specular Color Difference Signs & 0x08) >> 3)

Specular Color Difference Alpha, Specular Color Inverse Quant)

9.6.2.2.4.6.5 New Point Texture Coords



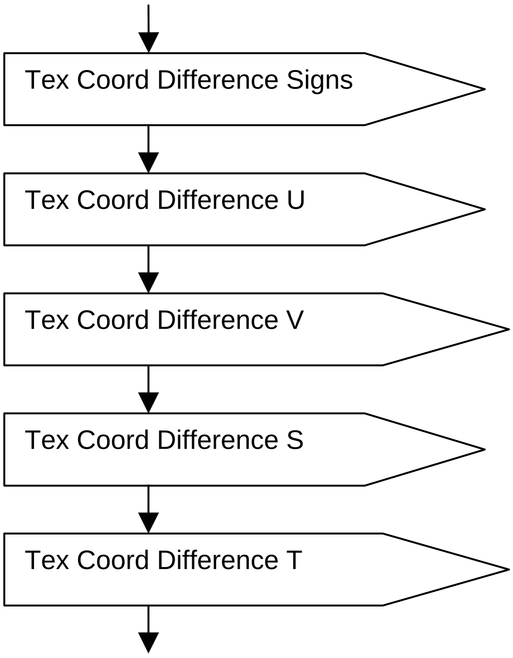
9.6.2.2.4.6.5.1 U8 [cTexCDup]: Tex Coord Duplicate Flag

Tex Coord Duplicate Flag is a set of flags that indicates if a new texture coordinate is added to the texture coordinate pool or if the most recently added texture coordinate is used again. If the flag is set (one), then the most recently added texture coordinate is used again. If the flag is not set (zero), then a new texture coordinate is added to the texture coordinate pool. All other values are reserved.

0x02 – New point uses duplicate texture coordinate

9.6.2.2.4.6.5.2 New Tex Coord

The New Tex Coord is predicted as the average of the texture coordinates at the same layer used by all points using the split position.



9.6.2.2.4.6.5.2.1 U8 [cTexCoordSign]: Tex Coord Difference Signs

Tex Coord Difference Signs is a collection of sign bits for the prediction differences.

0x01 – Sign bit for Texture Coord Difference U

0x02 – Sign bit for Texture Coord Difference V

0x04 – Sign bit for Texture Coord Difference S

0x08 – Sign bit for Texture Coord Difference T

9.6.2.2.4.6.5.2.2 U32 [cTexCDiffU]: Texture Coord Difference U

The reconstructed texture coordinate U is calculated as

> Reconstructed TexCoord U = InverseQuant(predicted Tex Coord U, (Tex Coord Signs & 0x01),
>
> Texture Coord Difference U, Texture Coord Inverse Quant).

9.6.2.2.4.6.5.2.3 U32 [cTexCDiffV]: Texture Coord Difference V

The reconstructed texture coordinate V is calculated as

> Reconstructed TexCoord V = InverseQuant(predicted Tex Coord V, ((Tex Coord Signs & 0x02) >> 1)),
>
> Texture Coord Difference V, Texture Coord Inverse Quant).

9.6.2.2.4.6.5.2.4 U32 [cTexCDiffS]: Texture Coord Difference S

The reconstructed texture coordinate S is calculated as

Reconstructed TexCoord S = InverseQuant(predicted Tex Coord S, ((Tex Coord Signs & 0x04) >> 2),

Texture Coord Difference S, Texture Coord Inverse Quant).

9.6.2.2.4.6.5.2.5 U32 [cTexCDiffT]: Texture Coord Difference T

The reconstructed texture coordinate T is calculated as

Reconstructed TexCoord T = InverseQuant(predicted Tex Coord, ((Tex Coord Signs & 0x08) >> 3),

Texture Coord Difference T, Texture Coord Inverse Quant).

9.6.3 Line Set (blocktypes: 0xFFFFFF37; 0xFFFFFF3F)

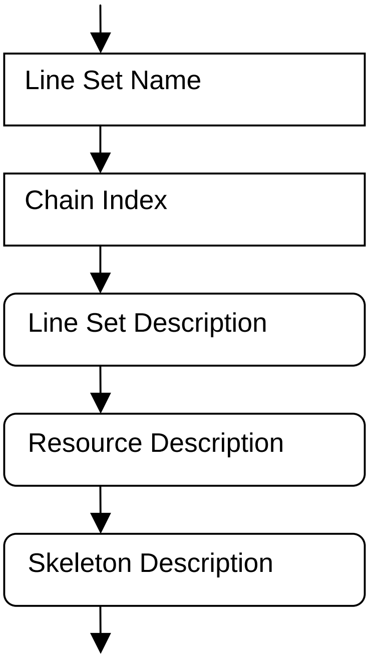
The Line Set generator contains the data needed to represent a set of lines.

The Line Set produces the following outputs: Renderable Group, Renderable Group Bounds, Transform Set.

The Line Set's outputs have no dependencies.

9.6.3.1 Line Set Declaration (blocktype: 0xFFFFFF37)

The Line Set Declaration contains the declaration information for a line set generator. The declaration information is sufficient to allocate space for the line set data and create the line set generator object. The line set data is contained in following continuation blocks.



9.6.3.1.1 String: Line Set Name

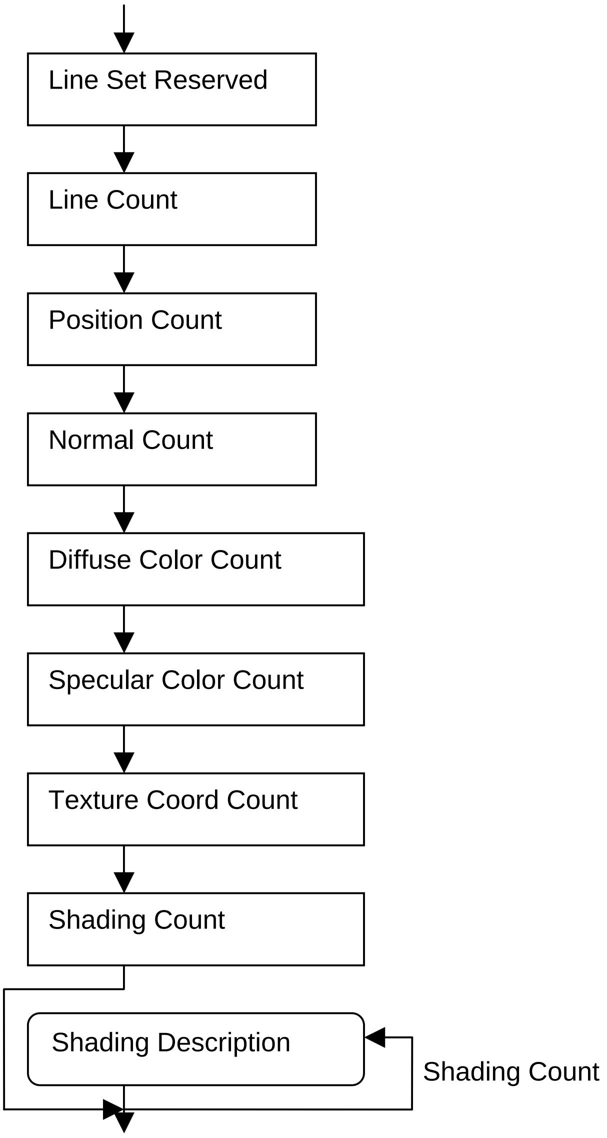
Line Set Name is the name of the line set generator. This name is also the name of the model resource modifier chain that contains the line set generator.

9.6.3.1.2 U32: Chain Index

Chain Index is the position of the line set generator in the model resource modifier chain. The value of Chain Index shall zero for this blocktype.

9.6.3.1.3 Line Set Description

Line Set Description describes the size of the line set. Line Set Description can be used to allocate space for the line set.



9.6.3.1.3.1 U32: Line Set Reserved

Line Set Reserved is a reserved field and shall have the value 0.

9.6.3.1.3.2 U32: Line Count

Line Count is the number of line segments in the line set.

9.6.3.1.3.3 U32: Position Count

Position Count is the number of positions in the position array.

9.6.3.1.3.4 U32: Normal Count

Normal Count is the number of normals in the normal array.

9.6.3.1.3.5 U32: Diffuse Color Count

Diffuse Color Count is the number of colors in the diffuse color array.

9.6.3.1.3.6 U32: Specular Color Count

Specular Color Count is the number of colors in the specular color array.

9.6.3.1.3.7 U32: Texture Coord Count

Texture Coord Count is the number of texture coordinates in the texture coordinate array.

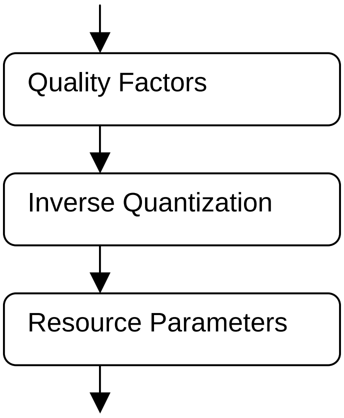
9.6.3.1.3.8 U32: Shading Count

Shading Count is the number of shading descriptions used in the line set.

9.6.3.1.3.9 Shading Description

Shading Description indicates which per vertex attributes, in addition to position and normal, are used by each shading list. Details are provided above in 9.6.1.1.3.9 Shading Description.

9.6.3.1.4 Resource Description



9.6.3.1.4.1 Quality Factors

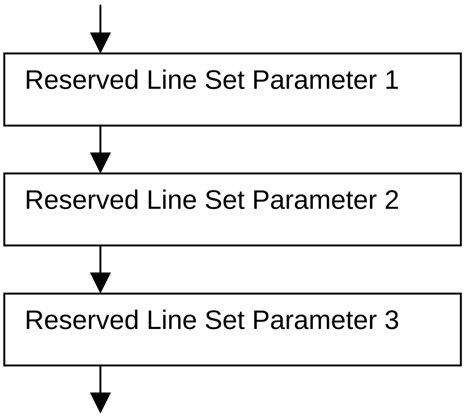
The quality factors are defined in 9.6.1.1.5.1 Quality Factors.

9.6.3.1.4.2 Inverse Quantization

Inverse quantization is defined in 9.6.1.1.5.2 Inverse Quantization.

9.6.3.1.4.3 Resource Parameters

Resource Parameters control the operation of the line set generator. The parameters defined in this section control the creation of the renderable line group. These parameters are reserved for future definition.


Reserved Line Set Parameter 1
Reserved Line Set Parameter 2
Reserved Line Set Parameter 3

9.6.3.1.4.3.1 U32: Reserved Line Set Parameter 1

Reserved Line Set Parameter 1 shall have the value 0.

9.6.3.1.4.3.2 U32: Reserved Line Set Parameter 2

Reserved Line Set Parameter 2 shall have the value 0.

9.6.3.1.4.3.3 U32: Reserved Line Set Parameter 3

Reserved Line Set Parameter 3 shall have the value 0.

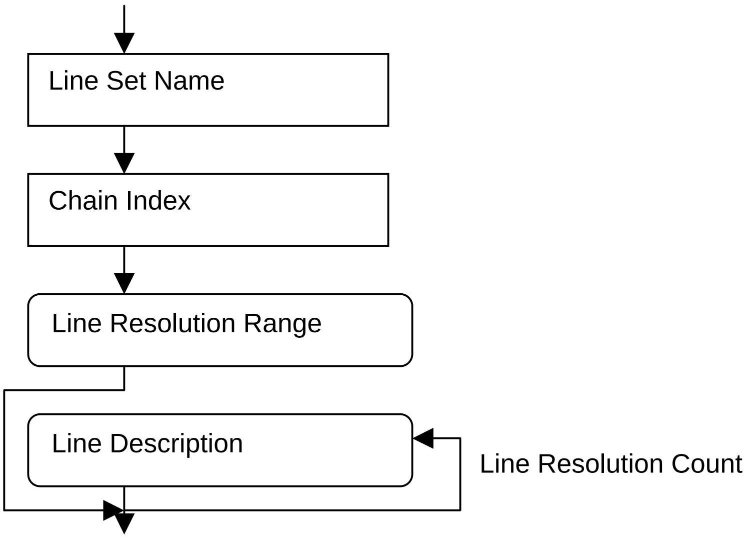
9.6.3.1.5 Skeleton Description

Skeleton Description provides bone structure information. Details are provided in 9.6.1.1.6 Skeleton Description.

9.6.3.2 Line Set Continuation (blocktype: 0xFFFFFF3F)

The Line Set Continuation contains data for a line set generator.

The Line Set Continuation block is a continuation type block.



9.6.3.2.1 String: Line Set Name

Line Set Name is the name of the line set generator. This name is also the name of the model resource modifier chain that contains the line set generator.

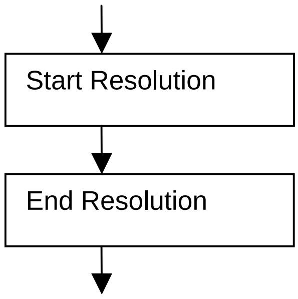
9.6.3.2.2 U32: Chain Index

Chain Index is the position of the line set generator in the model resource modifier chain. The value of Chain Index shall zero for this blocktype.

9.6.3.2.3 Line Resolution Range

Line Resolution Range specifies the range of line description data provided in this continuation block.

This continuation block contains line description data for positions from (Start Resolution) to (End Resolution – 1). The total number of positions added by this block is Point Resolution Count = End Resolution – Start Resolution.



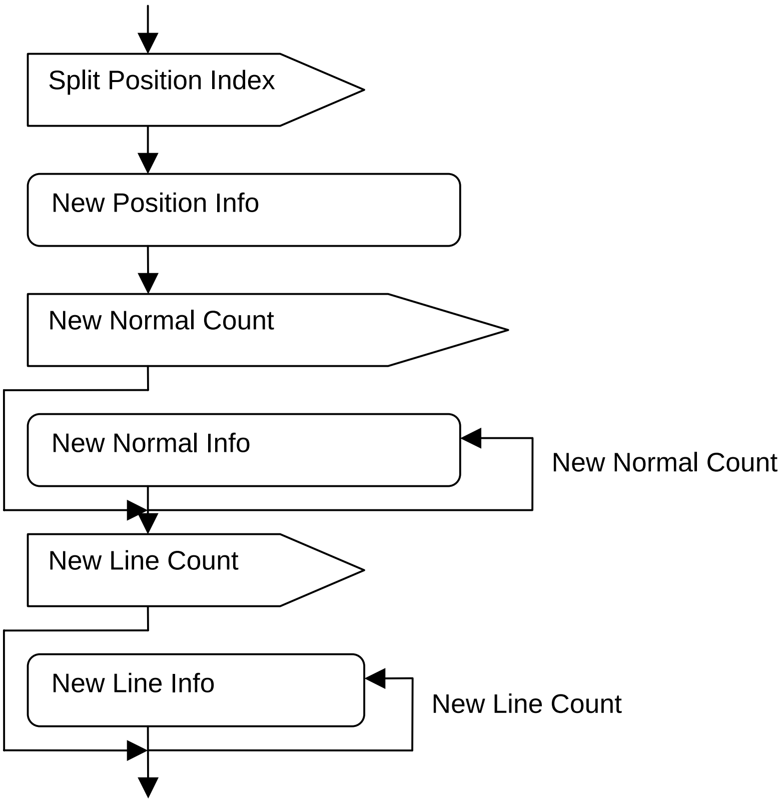
9.6.3.2.3.1 U32: Start Resolution

Start Resolution is the index of the first position added by this block.

9.6.3.2.3.2 U32: End Resolution

End Resolution is one more than the index of the last position added by this block.

9.6.3.2.4 Line Description



9.6.3.2.4.1 U32 [rCurrentPositionCount]: Split Position Index

Each Line Description adds one new position to the position array. Split Position Index is the index of the position used as a prediction reference by this Line Description. Split Position Index will be less than the current position count.

9.6.3.2.4.2 New Position Info

The new position is predicted as the split position. New Position Info is defined above in 9.6.1.3.4.10 New Position Info.

9.6.3.2.4.3 U32 [cNormlCnt]: New Normal Count

New Normal Count is the number of normals added to the normal array for use by lines using the new position.

9.6.3.2.4.4 New Normal Info

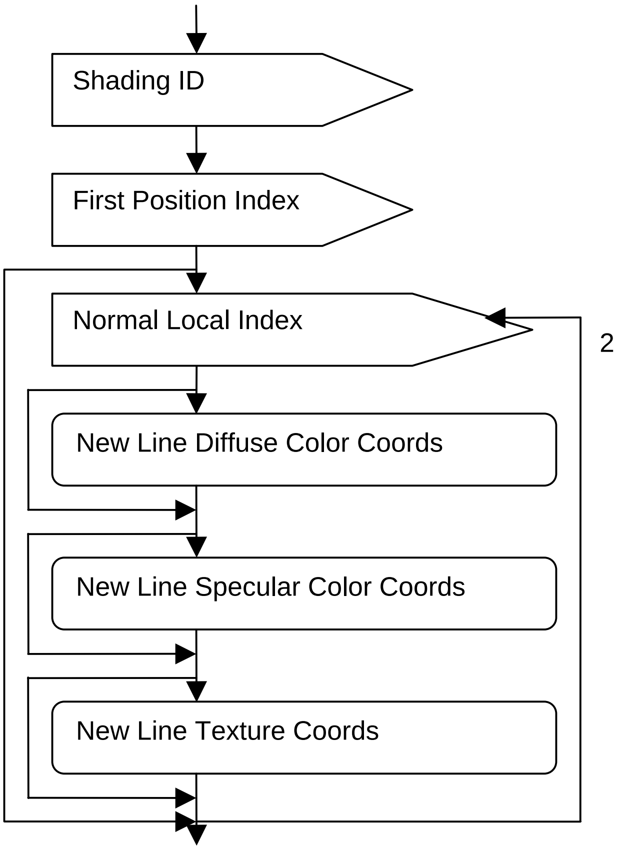
The normals are predicted as the average of the normals used at the split position. New Normal Info is defined in 9.6.2.2.4.4 New Normal Info.

9.6.3.2.4.5 U32 [cLineCnt]: New Line Count

New Line Count is the number of new lines added to the line set by this Line Description.

9.6.3.2.4.6 New Line Info

New Line Info describes a new line segment to be added to the line set. The normal, color and texture coordinate information is given first for the first end of the line segment and then for the second end of the line segment.



9.6.3.2.4.6.1 U32 [cShading]:Shading ID

Shading ID is the index of the shading description used for this line segment. The Shading Description array is defined in the Line Set Declaration block.

9.6.3.2.4.6.2 U32 [rCurrentPositionCount]: First Position Index

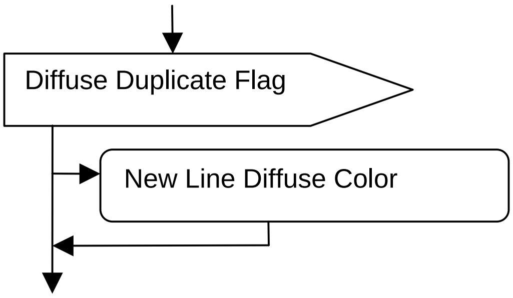
First Position Index is the index of the first end of the line segment. The index of the second end of the line segment is the current position count.

9.6.3.2.4.6.3 U32 [cNormlIdx]: Normal Local Index

The new line segment shall use a normal in the New Normal Info array for this line. Normal Local Index specifies which of the new normals should be used.

9.6.3.2.4.6.4 New Line Diffuse Color Coords

New Line Diffuse Color Coords is only present if the shading list identified by Shading List ID uses diffuse color coordinates. One of the Shading Attributes flags indicates the presence of diffuse color coordinates.



9.6.3.2.4.6.4.1 U8 [cDiffDup]: Diffuse Duplicate Flag

Diffuse Duplicate Flag is a set of flags that indicates if a new color is added to the color pool or if the most recently added color is used again. If the flag is set (one), then the most recently added color is used again. If the flag is not set (zero), then a new color is added to the diffuse color pool. All other values are reserved.

0x02 – New line segment end uses duplicate color

9.6.3.2.4.6.4.2 New Line Diffuse Color

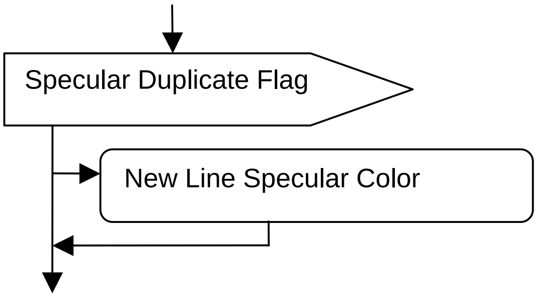
New Line Diffuse Color is present only if Diffuse Duplicate Flag indicates the new line does not use a duplicate color.

The New Line Diffuse Color is predicted as the average of the diffuse colors used by all line segment ends that use the Split Position.

The formatting for New Line Diffuse Color is the same as for 9.6.2.2.4.6.3.2 New Point Diffuse Color.

9.6.3.2.4.6.5 New Line Specular Color Coords

New Line Specular Color Coords is only present if the shading list identified by Shading ID uses specular color coordinates. One of the Shading Attributes flags indicates the presence of specular color coordinates.



9.6.3.2.4.6.5.1 U8 [cSpecDup]: Specular Duplicate Flag

Specular Duplicate Flag is a set of flags that indicates if a new color is added to the color pool or if the most recently added color is used again. If the flag is set (one), then the most recently added color is used again. If the flag is not set (zero), then a new color is added to the specular color pool. All other values are reserved.

0x02 – New line segment end uses duplicate color

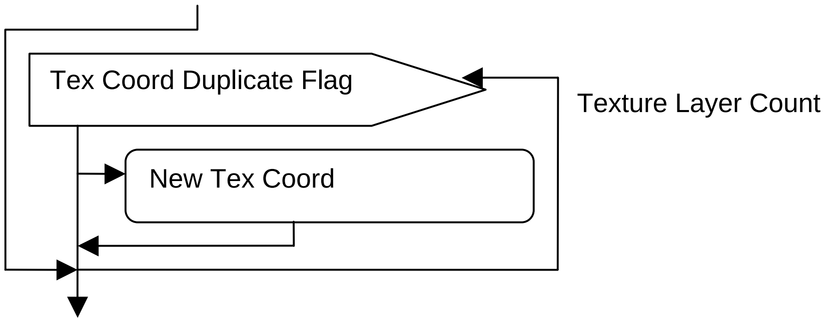
9.6.3.2.4.6.5.2 New Line Specular Color

New Line Specular Color is present only if Specular Duplicate Flag indicates the new line segment end does not use a duplicate color.

The New Line Specular Color is predicted as the average of the specular colors used at all line segment ends that use the Split Position.

The formatting for New Line Specular Color is the same as for 9.6.2.2.4.6.4.2 New Point Specular Color.

9.6.3.2.4.6.6 New Line Texture Coords



9.6.3.2.4.6.6.1 U8 [cTexCDup]: Tex Coord Duplicate Flag

Tex Coord Duplicate Flag is a set of flags that indicates if a new texture coordinate is added to the texture coordinate pool or if the most recently added texture coordinate is used again. If the flag is set (one), then the most recently added texture coordinate is used again. If the flag is not set (zero), then a new texture coordinate is added to the texture coordinate pool. All other values are reserved.

0x02 – New line segment end uses duplicate texture coordinate

9.6.3.2.4.6.6.2 New Tex Coord

The New Tex Coord is predicted as the average of the texture coordinates at the same layer used by all line segment ends using the split position. The formatting for New Tex Coord is the same as for 9.6.2.2.4.6.5.2 New Tex Coord.

9.7 Modifier blocks

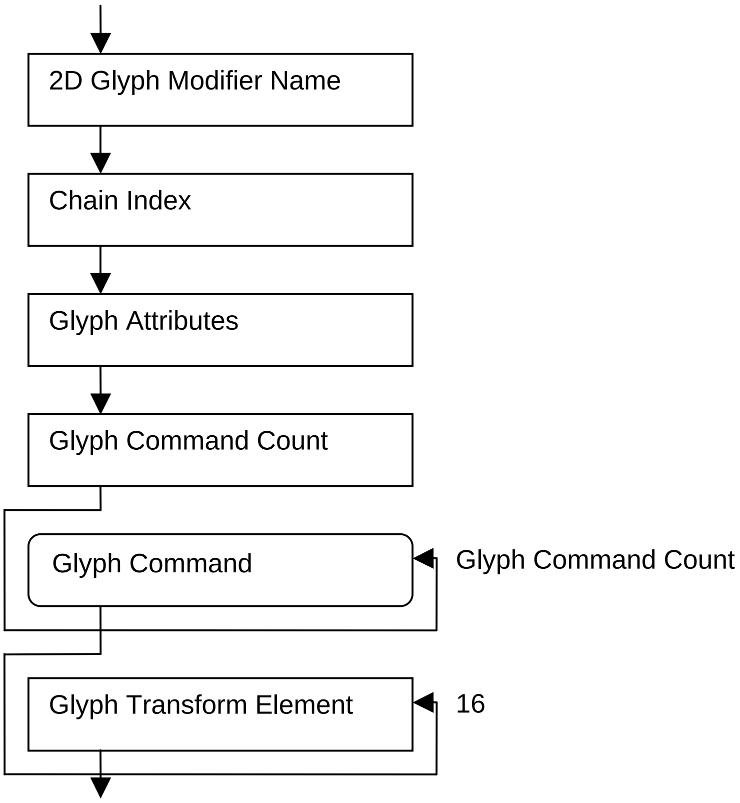
Modifier blocks contain the information necessary to create certain modifiers that can be added to a modifier chain. Note that the declaration blocks for modifiers must be contained within a modifier chain block.

9.7.1 2D Glyph Modifier (blocktype: 0xFFFFFF41)

The 2D Glyph Modifier contains information used to create a 2D shape. The shape is defined by a number of control points and parameters that define how to connect the points. The shape consists of a sequence of individual glyphs called a glyph string. Each glyph in the glyph string is defined by a sequence of drawing commands.

The 2D Glyph Modifier produces the following outputs: Renderable Group, and Renderable Group Bounds.

The 2D Glyph Modifier’s outputs depend on: Transfrom Set and View Transform.



9.7.1.1 String: 2D Glyph Modifier Name

2D Glyph Modifier Name is the string used to identify this 2D Glyph Modifier.

9.7.1.2 U32: Chain Index

Chain Index is the position of this modifier in the modifier chain.

9.7.1.3 U32: Glyph Attributes

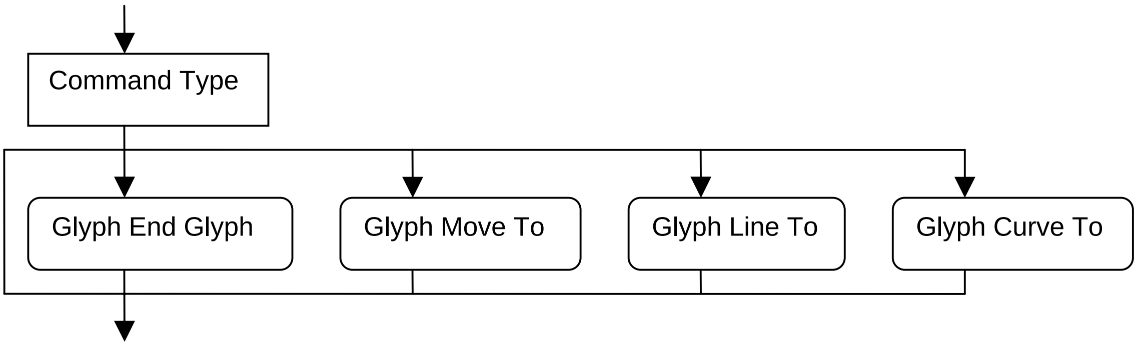
Glyph Attributes is a bit field containing information about the Glyph. The bit field is combined using a bitwise or. Other values are reserved. Valid values are:

0x00000001: Billboard: the glyph should be oriented to the view.

9.7.1.4 U32: Glyph Command Count

Glyph Command Count is the number of commands used to create this glyph.

9.7.1.5 Glyph Command



9.7.1.5.1 U32: Command Type

Valid Glyph Commands are:

0: STARTGLYPHSTRING — Start a sequence of glyph symbols. The glyphs symbols included in this sequence are defined in the subsequent commands until the next ENDGLYPHSTRING command.

1: ENDGLYPHSTRING — End a sequence of glyph symbols.

2: STARTGLYPH — Start a glyph. The glyph will be defined by the subsequent commands until the next ENDGLYPH command.

3: ENDGLYPH — End the current glyph definition.

4: STARTPATH — Start a new path to be drawn. The path is defined by the subsequent commands until the next ENDPATH command.

5: ENDPATH — End the current path.

8: MOVETO — Move the current drawing position.

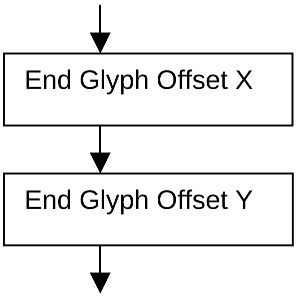
9: LINETO — Draw a line from the current drawing position to the new position.

10: CURVETO — Draw a curve from the current drawing position to the new position. The curve shape is determined by two control points.

ENDGLYPH, MOVETO, LINETO, and CURVETO require addition information described below. The other commands do not require any additional parameters.

9.7.1.5.2 Glyph End Glyph

Glyph End Glyph completes the current glyph and moves the starting point for the next glyph by the offset vector.



9.7.1.6.2.1 F32: End Glyph Offset X

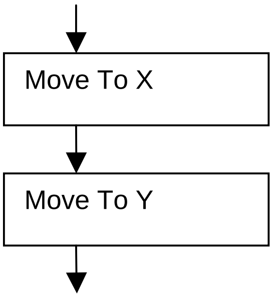
End Glyph Offset X is the horizontal offset between the starting point for this glyph and the starting point for next glyph.

9.7.1.6.2.2 F32: End Glyph Offset Y

End Glyph Offset Y is the vertical offset between the starting point for this glyph and the starting point for the next glyph.

9.7.1.5.3 Glyph Move To

The Glyph Move To command moves the active point without drawing.



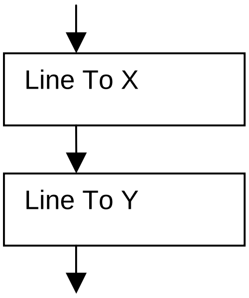
9.7.1.6.3.1 F32: Move To X

Move To X is the new horizontal position of the active point.

9.7.1.6.3.2 F32: Move To Y

Move To Y is the new vertical position of the active point.

9.7.1.5.4 Glyph Line To



The Glyph Line To command draws a line to the specified point.

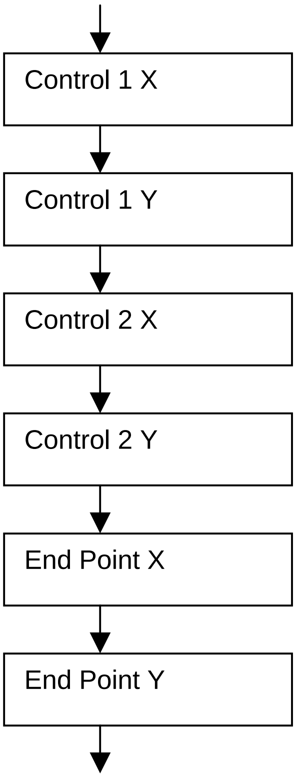
9.7.1.6.4.1 F32: Line To X

Line To X is the horizontal position of the end point of the line.

9.7.1.6.4.2 F32: Line To Y

Line To Y is the vertical position of the end point of the line.

9.7.1.5.5 Glyph Curve To



The Glyph Curve To command draws a curve to the specified point. The control points are used to determine the curve.

9.7.1.6.5.1 F32: Control 1 X

Control 1 X is the horizontal position of the first control point.

9.7.1.6.5.2 F32: Control 1 Y

Control 1 Y is the vertical position of the first control point.

9.7.1.6.5.3 F32: Control 2 X

Control 2 X is the horizontal position of the second control point.

9.7.1.6.5.4 F32: Control 2 Y

Control 2 Y is the vertical position of the second control point.

9.7.1.6.5.5 F32: End Point X

End Point X is the horizontal position of the end point of the curve.

9.7.1.6.5.6 F32: End Point Y

End Point Y is the vertical position of the end point of the curve.

9.7.1.6 F32: Glyph Transform Element

The Glyph Transform Elements make up the Transform that is applied to the glyph modifier after drawing to place it in the 3D world. The matrix is written in the alphabetic order described below:

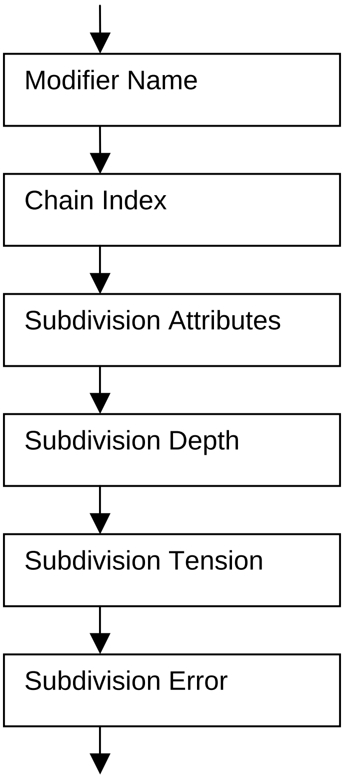
$$\begin{bmatrix} A & E & I & M \\ B & F & J & N \\ C & G & K & O \\ D & H & L & P \end{bmatrix}$$

9.7.2 Subdivision Modifier (blocktype: 0xFFFFFF42)

The Subdivision Modifier increases the resolution of a shape by dividing polygons into smaller polygons. The Subdivision Modifier block contains parameters that control the performance and appearance of the output of the subdivision algorithm.

The Subdivision Modifier produces the following outputs: Renderable Group Bounds.

The Subdivision Modifier's outputs depend on: Renderable Group, Transform Set, View Transform, and View Frustum.



9.7.2.1 String: Modifier Name

Modifier Name is the name of this Subdivision Modifier. Modifier Name is also the name of the object being modified and the name of the modifier chain that contains this modifier.

9.7.2.2 U32: Chain Index

Chain Index indicates the position of this modifier in the modifier chain.

9.7.2.3 U32: Subdivision Attributes

Subdivision Attributes is a collection of flags. The flags are combined using the binary OR operator. All other values are reserved.

0x00000001 – Enabled: The subdivision modifier is enabled.

0x00000002 – Adaptive: The subdivision modifier should use adaptive subdivision.

Uniform subdivision is used unless the adaptive subdivision flag is set. Uniform division divides all of the polygons the same number of times. Adaptive subdivision divides the polygons based on the model and if the polygons are visible.

9.7.2.4 U32: Subdivision Depth

Subdivision Depth is the maximum number of levels of subdivision.

9.7.2.5 F32: Subdivision Tension

Subdivision Tension is the tension value used for adaptive subdivision.

9.7.2.6 F32: Subdivision Error

Subdivision Error is the value of the screen space error metric. This value is used for adaptive subdivision.

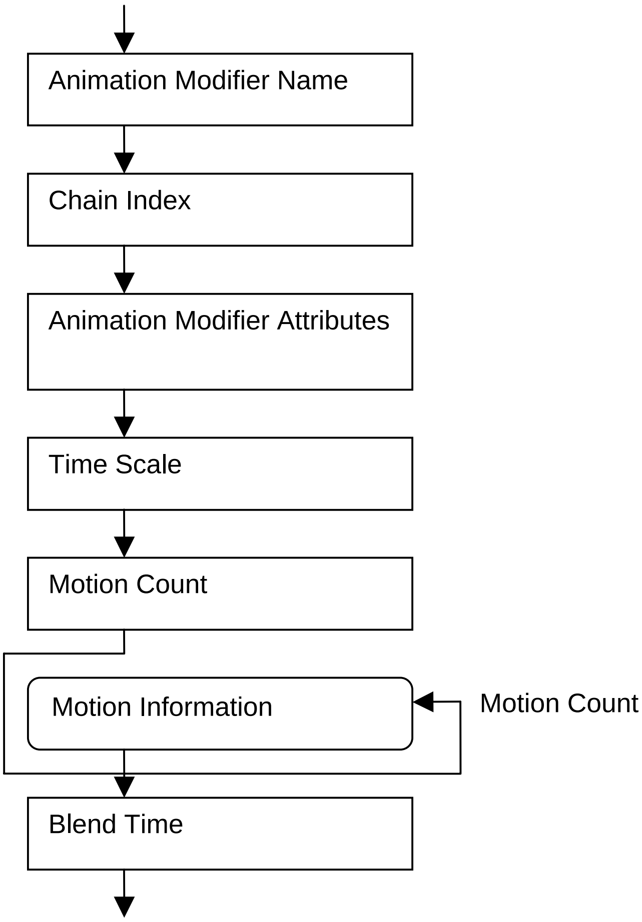
9.7.3 Animation Modifier (blocktype: 0xFFFFFF43)

The Animation Modifier block describes parameters for animating a node or a renderable group. These parameters indicate which motion resources should be used and how they should be applied. The animation modifer modifies the transforms for nodes and the transformations of bones relative to their parent bones. The hierarchy of bones is called a skeleton and is defined in the Skeleton Description of the geometry generator. The animation modifier uses the skeleton and bone weights defined by a bone weight modifier to change the positions and normals in the renderable group.

The animation modifier block is limited to modifying transformations of nodes and bones and modifying positions and normals based on the changes in the transformations. There are many other types of information that the animation modifier does not animate.

The Animation Modifier produces the following outputs: Transform Set, Renderable Group, and Skeleton.

The Animation Modifier’s outputs depend on: Transform Set, Simulation Time, Skeleton, Bone Weights, and Renderable Group.



9.7.3.1 String: Animation Modifier Name

Animation Modifier Name is the string that is used to identify this animation modifier. This is also the name of the modifier chain that contains this modifier.

9.7.3.2 U32: Chain Index

Chain Index is the position of this modifier in a modifier chain.

9.7.3.3 U32:Animation Modifier Attributes

Animation Modifier Attributes is a bit field that holds state information for this animation modifier. The values are combined using a bitwise OR operation. All other values are reserved.

0x00000001: Animation should start when possible.

0x00000002: The root bone is locked. The node’s root bone’s transform does not change as a result of the animation.

0x00000004: Playing a single track.

0x00000008: The bones’ transtorms should transition smoothly from one motion to the next during the animation.

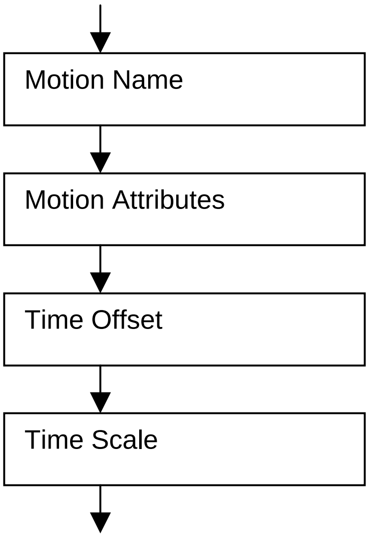
9.7.3.4 F32: Time Scale

Time Scale is a scaling value for the times of the motions.

9.7.3.5 U32: Motion Count

Motion Count is the number of motion resources referenced by this modifier. If the Motion Count is zero, the Animation Modifier will use the default motion.

9.7.3.6 Motion Information



9.7.3.6.1 String: Motion Name

Motion Name is a string that identifies a motion resource.

9.7.3.6.2 U32: Motion Attributes

Motion Attributes is a bit field of flags about the animation modifier. The values are combined with a bitwise or. Other values are reserved.

0x00000001: Loop: determines whether this motion repeats.

0x00000002: Sync: determines if all of the motion resources playing concurrently should end at the same time.

9.7.3.6.3 F32: Time Offset

Time Offset is the number of milliseconds to offset the start time of the motion.

9.7.3.6.4 F32: Time Scale

Time Scale is a scaling factor for the time of this motion resource for this animation modifier.

9.7.3.7 F32: Blend Time

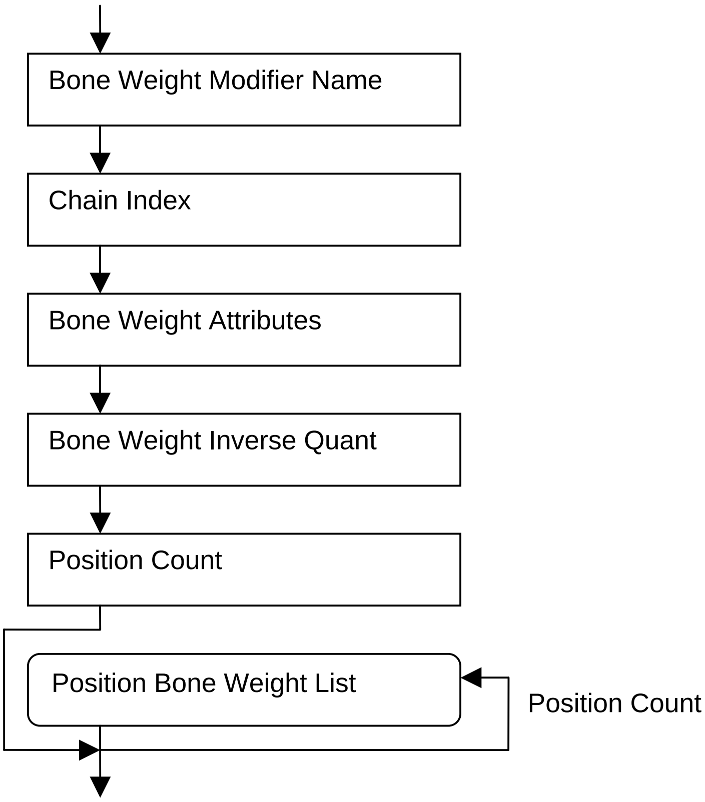
The Blend Time specifies the amount of time in milliseconds used when blending between motions.

9.7.4 Bone Weight Modifier (blocktype: 0xFFFFFF44)

The Bone Weight Modifier block describes a set of bone weights that can be added to a modifier chain. The animation modifier uses the bone weights in combination with the skeleton to animate the positions in a renderable group (mesh, point set, or line set). The normals are also changed by the animation modifier.

The Bone Weight Modifier produces the following outputs: Bone Weights.

The Bone Weight Modifier’s outputs have no dependencies.


Bone Weight Modifier Name
Chain Index
Bone Weight Attributes
Bone Weight Inverse Quant
Position Count
Position Bone Weight List
Position Count

9.7.4.1 String: Bone Weight Modifier Name

Bone Weight Modifier Name is the name of the modifier chain to which these bone weight should be added.

9.7.4.2 U32: Chain Index

Chain Index is the position of this modifier in a modifier chain.

9.7.4.3 U32: Bone Weight Attributes

The bone weights may be applied to the type of geometry specified by he Bone Weight Attributes. All other values are reserved.

0x00000001 – these bone weights are for a mesh

0x00000002 – these bone weights are for a line set

0x00000004 – these bone weights are for a point set

9.7.4.4 F32: Bone Weight Inverse Quant

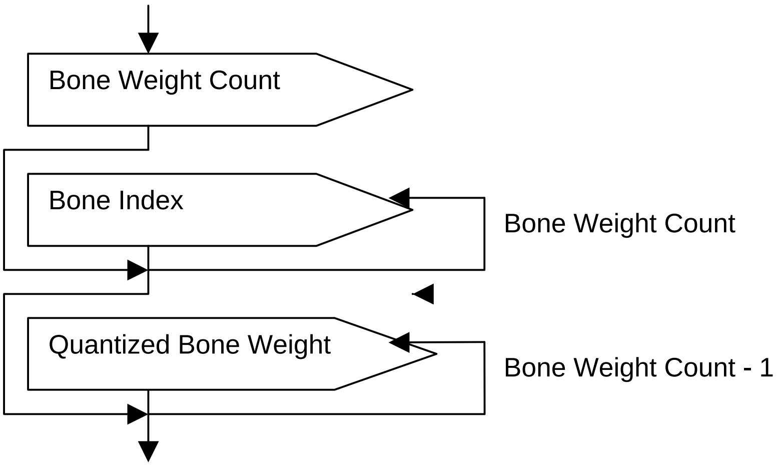
The bone weight inverse quant is the inverse quantization factor for the bone weights below.

9.7.4.5 U32: Position Count

Position Count is the number of positions for which bone weights are provided by this modifier.

9.7.4.6 Position Bone Weight List

Position Bone Weight List indicates which bones have a non-zero influence at this position.The reconstructed bone weights at this position should sum to +1.0. The bone weights cannot be negative.



9.7.4.6.1 U32 [cBoneWeightCnt]: Bone Weight Count

Bone Weight Count is the number of bones which have influence at this position.

9.7.4.6.2 U32 [cBoneIdx]: Bone Index

Bone Index is the index of the bone in the skeleton that has influence at this position. Bone Index is present only if Bone Weight Count is greater than zero.

9.7.4.6.3 U32 [cQntBoneWeight]: Quantized Bone Weight

Quantized Bone Weight is the quantized bone weight value. Quantized Bone Weight is present only if Bone Weight Count is greater than one.

For other than the last bone weight value, the reconstructed bone weight value is calculated as:

(reconstructed bone weight) = (Quantized Bone Weight) * (Bone Weight Inverse Quant)

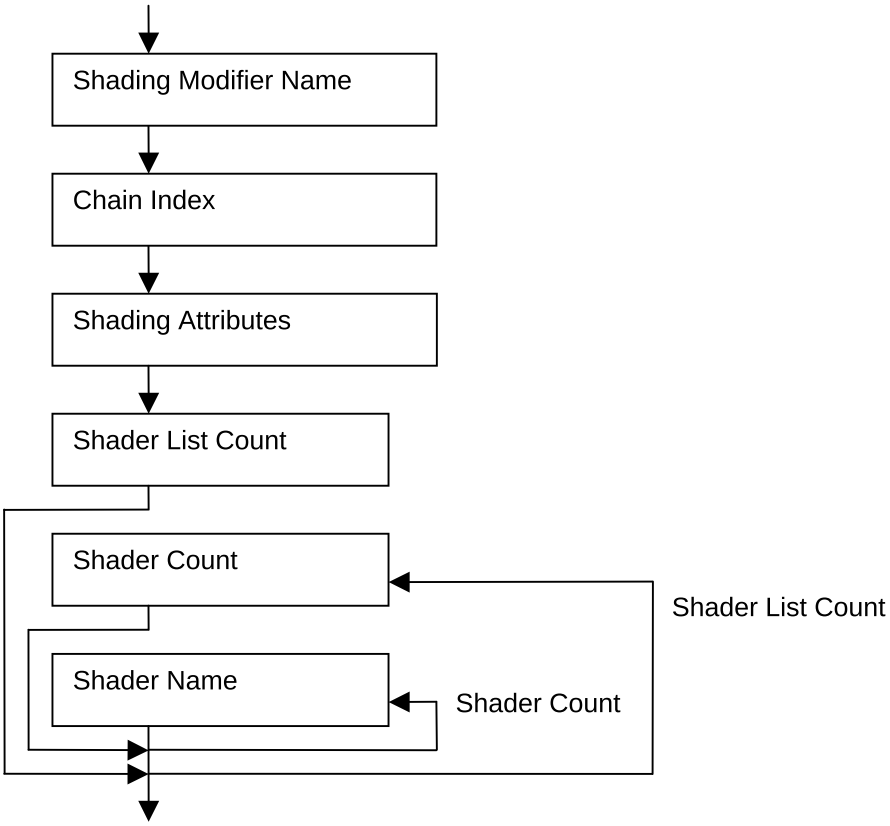
The last bone weight value is reconstructed by subtracting the sum of all the other reconstructed bone weight values from +1.0. The sum of all the bone weights at this position will be +1.0.

9.7.5 Shading Modifier (blocktype: 0xFFFFFF45)

The Shading Modifier block describes the shading group that is used in the drawing of a renderable group. The shading modifier replaces the shading group associated with a renderable group.

The Shading Modifier produces the following outputs: Renderable Group.

The Shading Modifier’s outputs depend on: Renderable Group.



9.7.5.1 String: Shading Modifier Name

The Shading Modifier Name identifies this shading modifier. Shading Modifier Name is also the name of the modifier chain that contains this modifier.

9.7.5.2 U32: Chain Index

Chain Index indicates the position of this modifier in the modifier chain.

9.7.5.3 U32: Shading Attributes

Shading Attributes is a collection of flags. The flags are combined using the binary OR operator. Other attributes are reserved.

0x00000001 – Mesh: the shading group is applied to the renderable mesh group.

0x00000002 – Line: the shading group is applied to the renderable line group.

0x00000004 – Point : the shading group is applied to the renderable point group.

9.7.5.4 U32: Shader List Count

Shader List Count is the number of shader lists in the shading group. Each shader list is associated with a renderable element in the associated renderable group.

If the number of shader lists exceeds the number of renderable elements, the excess shader lists have no effect. If the number of shader lists is less than the number of renderable elements, the excess renderable elements shall be associated with a shader list containing one shader and that one shader shall be the default shader.

9.7.5.5 U32: Shader Count

Shader Count is the number of shaders in the shader list.

9.7.5.6 String: Shader Name

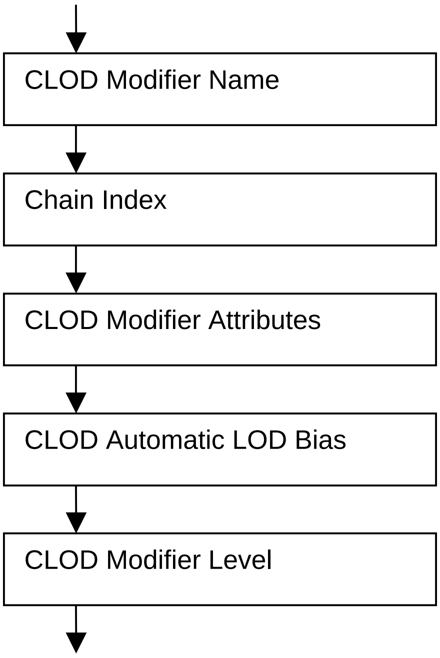
Each shader in the shader list is identified by Shader Name. Shader Name refers to a shader in the shader resource palette.

9.7.6 CLOD Modifier (blocktype: 0xFFFFFF46)

The CLOD Modifier adjusts the level of detail in the renderable meshes in the data packet. The CLOD Modifier block contains parameters for how the level of detail should be adjusted.

The CLOD Modifier produces the following outputs: Renderable Group.

The CLOD Modifier's outputs depend on: Renderable Group, Renderable Group Bounds, Transform Set, View Transform, View Frustum, View Size,



9.7.6.1 String: CLOD Modifier Name

CLOD Modifier Name is the name of the modifier chain to which the CLOD Modifier should be added.

9.7.6.2 U32: Chain Index

Chain Index is the position of this modifier in a modifier chain.

9.7.6.3 U32: CLOD Modifier Attributes

0x00000000 – Default attributes (automatic LOD control is disabled)

0x00000001 – automatic level of detail control.

If the automatic level of detail control bit is set, the level of detail of the model should be determined automatically at runtime. The calculation of the level of detail is implementation specific, but may be adjusted based on a target rendering frame rate or the size of the model on screen. All other values are reserved.

9.7.6.4 F32: CLOD Automatic Level of Detail Bias

The CLOD Modifier Automatic Level of Detail Bias is used when the level of detail of geometry is to be determined at runtime. The range of bias is 0.0 to 1.0. When calculating the level of detail used, the runtime should set a higher level of detail for larger values of this bias.

9.7.6.5 F32: CLOD Modifier Level

The range for CLOD Modifier Level is 0.0 to 1.0.

The CLOD Modifier adjusts the resolution of the renderable meshes.

The target resolution is determined by multiplying the CLOD Modifier Level by the maximum resolution of the author mesh. If the target resolution is less than the minimum resolution, then the resolution will be adjusted to the minimum resolution.

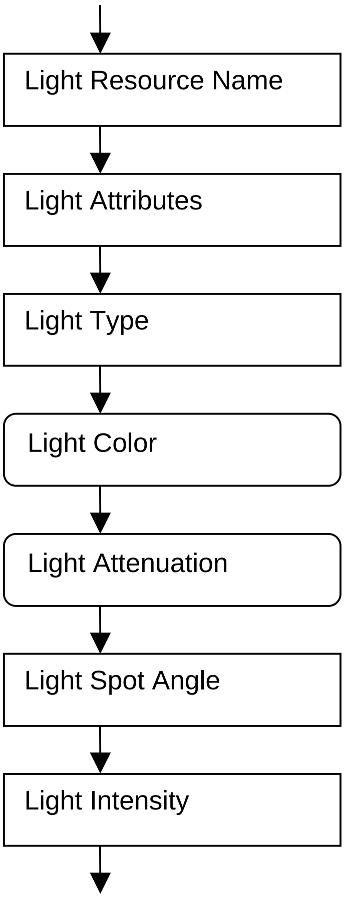
If the automatic LOD control is enabled, then the automatic LOD control overrides the CLOD Modifier Level specified in this block.

9.8 Resource blocks

Resource blocks contain the declarative information for resources. The resources can then be referenced by nodes to create specific instances during rendering.

9.8.1 Light Resource (blocktype: 0xFFFFFF51)

The Light Resource contains information regarding the type of light, color, attenuation, and intensity. Some of the fields are not used for all light types. Unused fields will not affect the appearance of the scene. For example, ambient lights do not use attenuation.



9.8.1.1 String: Light Resource Name

Light Resource Name is the name used to identify this Light Resource.

9.8.1.2 U32: Light Attributes

Light Attributes is a collection of flags. The flags are combined using the binary OR operator.

Other values are reserved.

0x00000001 – Light Enabled; the light is used.

0x00000002 – Specular; the light provides specular highlights.

0x00000004 – Spot Decay; the spot light has a smooth edge and not a hard edge cutoff.

9.8.1.3 U8: Light Type

Light Type is the type of this Light Resource.

0x00 – Ambient; Light provides uniform non-directional light to the scene.

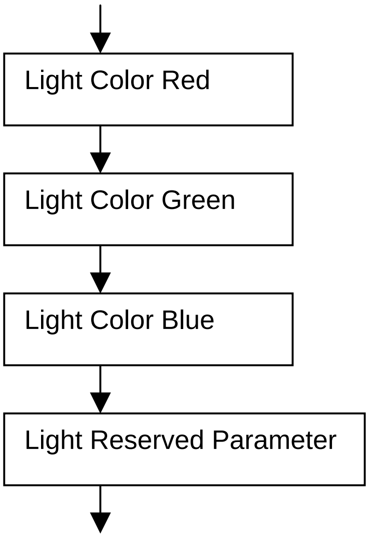
0x01 – Directional; Light provides uniform directional light to the scene.

0x02 – Point; Light is emitted from a specific point in the scene.

0x03 – Spot; Like point light, but constrained to specific directions.

9.8.1.4 Light Color

Light Color is the color of the Light Resource.



9.8.1.4.1 F32: Light Color Red

Light Color Red is the red component of the Light Color. The normal range of color component values is 0.0 (darkest) to 1.0 (brightest). Values outside this range are allowed.

9.8.1.4.2 F32: Light Color Green

Light Color Green is the green component of the Light Color. The normal range of color component values is 0.0 (darkest) to 1.0 (brightest). Values outside this range are allowed.

9.8.1.4.3 F32: Light Color Blue

Light Color Blue is the blue component of the Light Color. The normal range of color component values is 0.0 (darkest) to 1.0 (brightest). Values outside this range are allowed.

9.8.1.4.4 F32: Light Reserved Parameter

Light Color Reserved Parameter is a reserved field and shall have the value 1.0. This value shall not be used by a loader.

9.8.1.5 Light Attenuation

Light Attenuation is a vector of attenuation factors. Lights that are of type Point or Spot will light objects based on the distance from the object’s vertices to the light’s position. The formula for this attenuation is

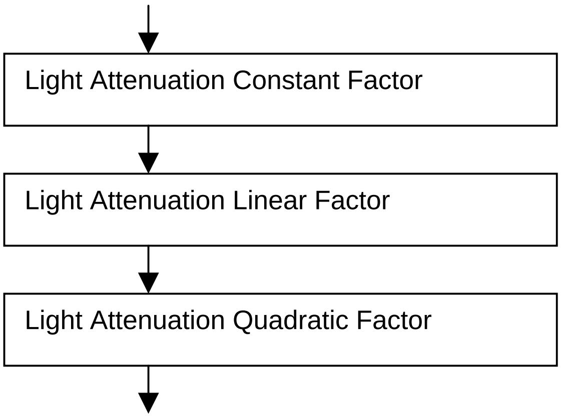
1 / (C + L*D + Q*D*D)

D: distance from vertex position to light position

C: attenuation constant factor

L: attenuation linear factor

Q: attenuation quadratic factor


Light Attenuation Constant Factor
Light Attenuation Linear Factor
Light Attenuation Quadratic Factor

9.8.1.5.1 F32: Light Attenuation Constant Factor

Light Attenuation Constant Factor is used to calculate attenuation for spot and point lights.

9.8.1.5.2 F32: Light Attenuation Linear Factor

Light Attenuation Linear Factor is used to calculate attenuation for spot and point lights.

9.8.1.5.3 F32: Light Attenuation Quadratic Factor

Light Attenuation Quadratic Factor is used to calculate attenuation for spot and point lights.

9.8.1.6 F32: Light Spot Angle

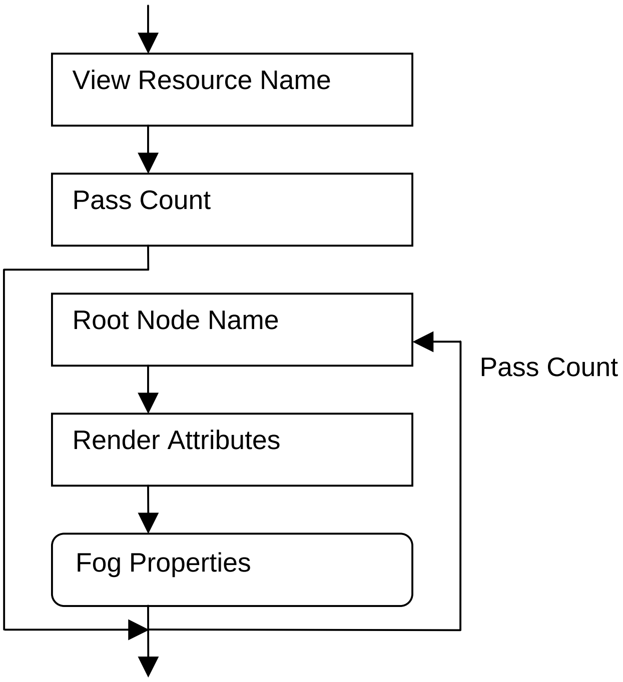
Light Spot Angle is the angle of the cone that emanates from the light position and defines what portions of the scene are affected by this light. The Light Spot Angle is only used if the light is of Spot type.

9.8.1.7 F32: Light Intensity

Light Intensity is multiplied into the affect that this light has on a scene. It is similar in practice to 1 / (Light Attenuation Constant Factor), but works on Directional lights in addition to Point and Spot lights (this does not affect lights with type Ambient). Keep in mind that this value can have any value (including negative and 0), resulting in the ability to produce some strange effects.

9.8.2 View Resource (blocktype: 0xFFFFFF52)

The View Resource contains information regarding the rendered view that is not specific to a particular view instance. Fields include: fog and frame buffer properties. More fields, such as view port, backdrops and overlays are stored at the node level and are specific to each instance.



9.8.2.1 String: View Resource Name

The View Resource Name is the name used to identify this view resource.

9.8.2.2 U32: Pass Count

The Pass Count is the number of passes that are used when rendering this view. Note that the rendering system may change the order to correctly render transparent objects

9.8.2.3 String: Root Node Name

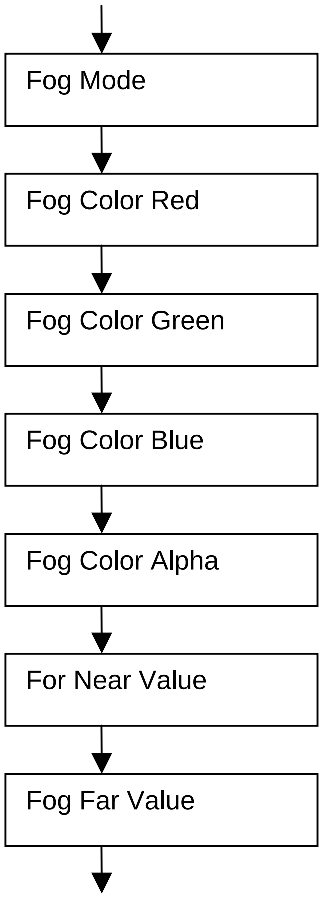
The Root Node Name is the name of a node. The view will render this node and all of the node’s children.

9.8.2.4 U32: Render Attributes

Render Attributes is a bit field that determines properties of the view. The only property defined for this edition is Fog Enabled. The properties are combined with a bitwise or operation. Other values are reserved.

0x00000001: Fog Enabled

9.8.2.5 Fog Properties



9.8.2.5.1 U32: Fog Mode

The fog mode determines the method used for rendering fog. In the following equations, d represents the distance from the view. Fog is enabled or disabled by the flag in the render attributes field above.

0x00000000: Linear

$$f = \frac{end - d}{end - start}$$

0x00000001: Exponential

$$f = e^{-(d \cdot density)}, \; density = \frac{\ln(100)}{fogfar}$$

0x00000002: Exponential 2

$$f = e^{-(d \cdot density)^2}, \; density = \frac{\sqrt{\ln(100)}}{fogfar}$$

Fog far is the fog far value specified below (9.8.2.5.7).

9.8.2.5.2 F32: Fog Color Red

Fog Color Red is the red component of the fog’s color.

9.8.2.5.3 F32: Fog Color Green

Fog Color Green is the green component of the fog’s color.

9.8.2.5.4 F32: Fog Color Blue

Fog Color Blue is the blue component of the fog’s color.

9.8.2.5.5 F32: Fog Color Alpha

Fog Color Alpha is the alpha component of the fog’s color.

9.8.2.5.6 F32: Fog Near Value

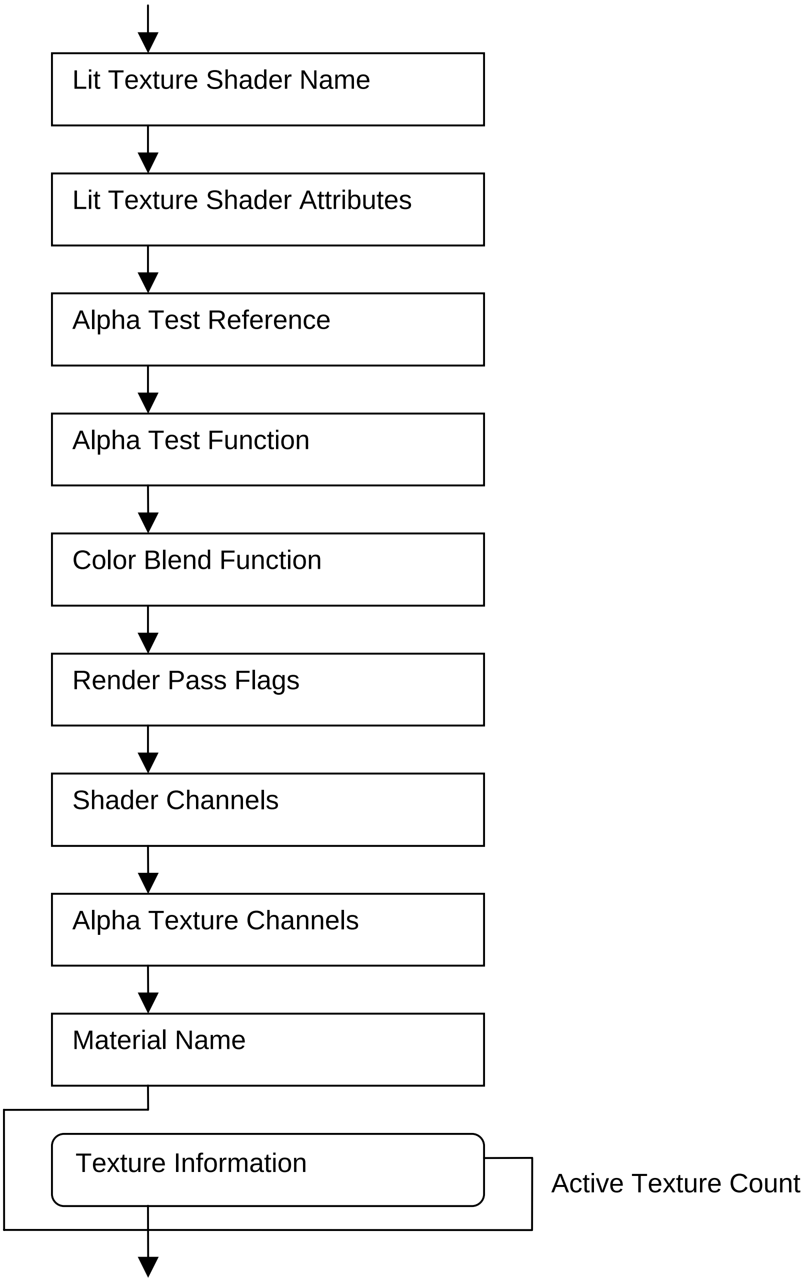
For linear fog mode, the Fog Near Value is the distance from the view where fog begins.

9.8.2.5.7 F32: Fog Far Value

For linear fog mode, the Fog Far Value is the distance from the view where the fog reaches its maximum density. The fog far value is also used to calculate the fog density scale factor used with exp and exp2 fog modes (details are in 9.8.2.5.1 Fog Mode).

9.8.3 Lit Texture Shader (blocktype: 0xFFFFFF53)

The Lit Texture Shader contains information needed to determine the appearance of a surface during rendering. The Lit Texture Shader includes references to Material Resources and Texture Resources and how to combine those resources when rendering.



9.8.3.1 String: Lit Texture Shader Name

Lit Texture Shader Name is the string used to identify this shader.

9.8.3.2 U32: Lit Texture Shader Attributes

Lit Texture Shader Attibutes is a bit field that stores information about the shader. The attributes are combined by a bitwise OR operation. All other values are reserved.

0x00000001: Lighting Enabled

0x00000002: Alpha Test Enabled

0x00000004: Use Vertex Color

9.8.3.3 F32: Alpha Test Reference

Alpha Test Reference is the value used in comparisons when alpha test is enabled.

9.8.3.4 U32: Alpha Test Function

0x00000610: NEVER: The test never passes. No pixels are drawn.

0x00000611: LESS: The rendered alpha value must be less than the reference value.

0x00000612: GREATER: The rendered alpha value must be greater than the ref. value.

0x00000613: EQUAL: The rendered alpha value must be equal to the reference value.

0x00000614: NOT_EQUAL: The rendered alpha value must not be equal to the ref. value.

0x00000615: LEQUAL: The rendered alpha value must be less than or equal to the reference value.

0x00000616: GEQUAL: The rendered alpha value must be greater than or equal to the reference value.

0x00000617: ALWAYS: The test always passes. No pixels are rejected.

9.8.3.5 U32: Color Blend Function

Color Blend Function is the function used to blend rendered pixels and the existing frame buffer.

0x00000604: FB_ADD: Add the RGB components into the framebuffer

0x00000605: FB_MULTIPLY: Multiply the RGB components into the framebuffer

0x00000606: FB_ALPHA_BLEND: Linear blend the RGB components into the framebuffer based on the rendered alpha value.

0x00000607: FB_INV_ALPHA_BLEND: Linear blend the RGB components into framebuffer based on the inverse (1.0 - a) of the rendered alpha.

9.8.3.6 U32: Render Pass Enabled Flags

The Render Pass Enable Flags determines which passes this shader uses. Each bit (1<<n) in the flags determines if the shader is used in pass n. The flags are combined with the bitwise OR operation.

9.8.3.7 U32: Shader Channels

Shader Channels is a bit field that determines which of the model's texture coordinate layers are used for this shader. The least significant 8 bits are used to store this information. A layer is active if the corresponding bit is set. The Active Texture Count is the number of active shader channels. The active bits are combined with a bitwise OR operation. The remaining 24 bits are reserved.

Example: A Shader Channel Value of (binary 00001001) would mean the first and fourth texture coordinate layers are used. Another shader using the same model could have a Shader Channel value of (binary 00000111) meaning the first, second, and third texture coordinate layers are used by that shader.

9.8.3.8 U32: Alpha Texture Channels

Alpha Texture Channels is a bit field that determines which texture layers should use the alpha component if an alpha component exists. The Alpha Texture Channels bits correspond to the Shader Channels bits. The Alpha Texture Channel bit shall not be set if the corresponding Shader Channel bit is not set. The least significant 8 bits are used to store this information. A layer is active if the corresponding bit is set. The active bits are combined with a bitwise OR operation. The remaining 24 bits are reserved.

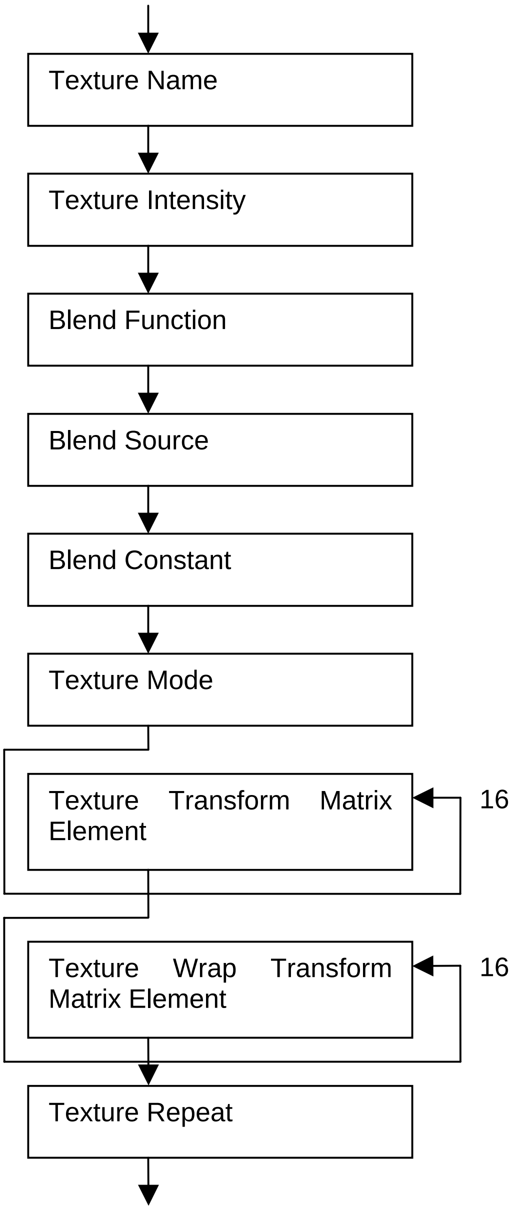
Example: A shader has a Shader Channel value of (binary 00000011) and an Alpha Texture Channels value of (binary 00000010) would mean that the shader should use the alpha component for the second texture layer and should ignore the alpha component for the first texture layer. Ignoring the alpha component is equivalent to assuming the alpha value is 1.0.

9.8.3.9 String: Material Name

The Material Name is the name of the material associated with this shader that determines how the shader appears when lit.

9.8.3.10 Texture Information

Texture Information identifies the texture used by a particular shader channel. Texture Information also describes how the textures are blended and which texture coordinates to use for that shader channel. Texture Information is repeated once for each active shader channel. Active Texture Count is the number of active shader channels as described in 9.8.3.7 Shader Channels.



9.8.3.10.1 String: Texture Name

The Texture Name is the name of the texture resource that is used for this texture layer.

9.8.3.10.2 F32: Texture Intensity

Texture Intensity is a scale factor applied to the color components of the texture.

9.8.3.10.3 U8: Blend Function

The Blend Function determines how the current texture layer is combined with the result from previous layers.

0 – Multiply: blended = current * previous

1 – Add: blended = current + previous

2 – Replace: blended = current

3 – Blend: blended = current * currentAlpha + previous * (1 – currentAlpha).

9.8.3.10.4 U8: Blend Source

Blend Source indicates whether the blending operation combines the current layer with the result from previous layers using a blending constant or the alpha value of each pixel.

0 – Alpha value of each pixel

1 – Blending constant.

9.8.3.10.5 F32: Blend Constant

The Blending constant is used when combining the results of texture layers.

9.8.3.10.6 U8: Texture Mode

The Texture Mode indicates the source of the texture coordinates used to map the texture onto the model. TM_NONE indicates the shader should use the texture coordinates of the model. All other coordinates are generated by the shader as needed.

0x00: TM_NONE	The shader does not generate texture coordinates.
0x01: TM_PLANAR	The shader transforms the model by the inverse of the texture wrap transform and then performs a planar x, y mapping of the texture onto the model.
0x02: TM_CYLINDRICAL	The shader transforms the model by the inverse of the texture wrap transform and then performs a cylindrical mapping of the texture onto the model. The Z-axis of the transformed model is the cylinder axis.
0x03: TM_SPHERICAL	The shader transforms the model by the inverse of the texture wrap transform and then performs a spherical mapping of the texture onto the model. The Z-axis of the transformed model is the sphere’s vertical axis.
0x04: TM_REFLECTION:	The shader performs a spherical reflection mapping. This is used to generate texture coordinates for reflection mapping when using a specially designed spherical reflection texture.

9.8.3.10.7 F32: Texture Transform Matrix Element

The Texture Transform Matrix operates on the texture coordinates in this texture coordinate layer of the model. This transform is used for all texture modes.

The matrix is written in the alphabetic order described below:

$$\begin{bmatrix} A & E & I & M \\ B & F & J & N \\ C & G & K & O \\ D & H & L & P \end{bmatrix}.$$

9.8.3.10.8 F32: Texture Wrap Transform Matrix Element

The Texture Wrap Transform is used for the following texture modes: TM_PLANAR; TM_CYLINDRICAL; TM_SPHERICAL. In these texture modes, texture coordinates are procedurely generated based on the position values of vertices in the model.

The texture coordinates from a reference shape are projected onto the model. The Texture Wrap Transform operates on the procedurely generated texture coordinates before they are applied to the model.

The matrix is written in the alphabetic order described below:

$$\begin{bmatrix} A & E & I & M \\ B & F & J & N \\ C & G & K & O \\ D & H & L & P \end{bmatrix}.$$

NOTE
In an equivalent implementation, the inverse of the Texture Wrap Transform could operate on the position value in the model to look up the texture coordinate values in the reference shape.

The Texture Wrap Transform is also discussed in 9.8.3.10.6 Texture Mode.

9.8.3.10.9 U8: Texture Repeat

Texture Repeat indicates whether or not the texure in the specified texture layer should be tiled beyond the coordinate range. Texture Repeat is a bitfield and the values below are combined using a bitwise OR operator. All other values are reserved.

0x01 – Repeat in the direction of the first texture coordinate dimension

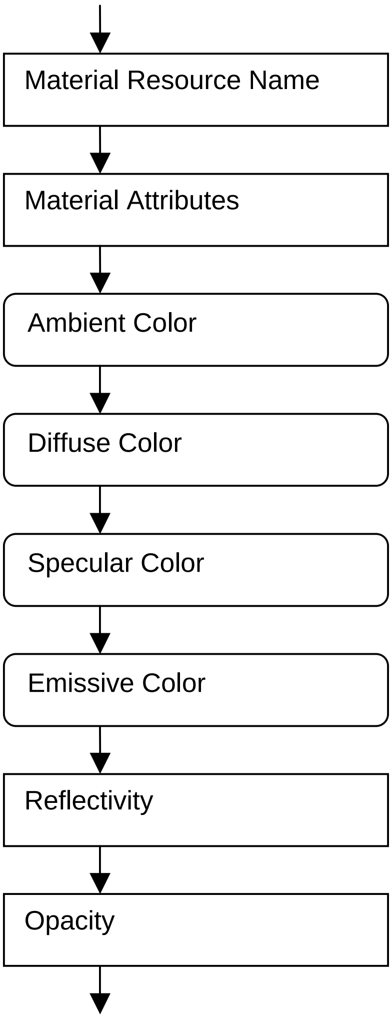
0x02 – Repeat in the direction of the second texture coordinate dimension

Repeating the texture shall be accomplished in the manner of tiling the texture image.

NOTE
This edition of the specification does not support 3 and 4-dimensional texture resources. Future editions may support additional repeat modes and may support 3 and 4-dimensional textures.

9.8.4 Material Resource (blocktype: 0xFFFFFF54)

The Material Resource contains information defining how a material interacts with light in a scene. A shader references a Material Resource to determine how surfaces will appear when rendered.



9.8.4.1 String: Material Resource Name

Material Resource Name is the string used to identify this material.

9.8.4.2 U32: Material Attributes

Material Attributes is a collection of flags that define which of the material attributes specified below are enabled. The flags are combined using the binary OR operator. Other values are reserved.

0x00000001 – Ambient

0x00000002 – Diffuse

0x00000004 – Specular

0x00000008 – Emissive

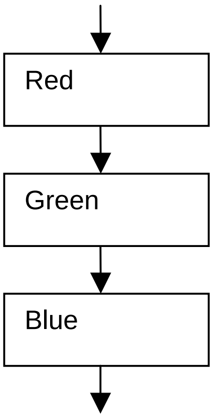
0x00000010 – Reflectivity

0x00000020 – Opacity

The material attributes are described below.

9.8.4.3 Ambient Color

The Ambient Color defines the material's appearance in ambient light. The normal range of color component values is 0.0 (darkest) to 1.0 (brightest). Values outside this range are allowed.



9.8.4.3.1 F32: Red

Red is the red component of the color.

9.8.4.3.2 F32: Green

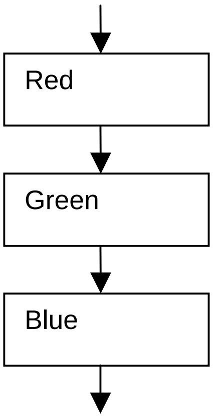
Green is the green component of the color.

9.8.4.3.3 F32: Blue

Blue is the blue component of the color.

9.8.4.4 Diffuse Color

The Diffuse Color defines the material's appearance in diffuse light. The normal range of color component values is 0.0 (darkest) to 1.0 (brightest). Values outside this range are allowed.



9.8.4.4.1 F32: Red

Red is the red component of the color.

9.8.4.4.2 F32: Green

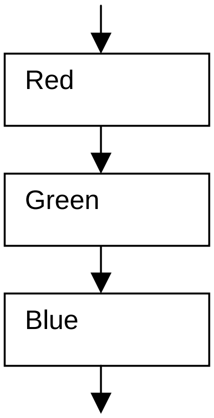
Green is the green component of the color.

9.8.4.4.3 F32: Blue

Blue is the blue component of the color.

9.8.4.5 Specular Color

The Specular Color defines the material's appearance in specular light. The normal range of color component values is 0.0 (darkest) to 1.0 (brightest). Values outside this range are allowed.



9.8.4.5.1 F32: Red

Red is the red component of the color.

9.8.4.5.2 F32: Green

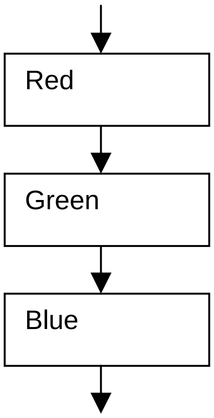
Green is the green component of the color.

9.8.4.5.3 F32: Blue

Blue is the blue component of the color.

9.8.4.6 Emissive Color

The Emissive Color defines the light that the material appears to give off. The normal range of color component values is 0.0 (darkest) to 1.0 (brightest). Values outside this range are allowed.



9.8.4.6.1 F32: Red

Red is the red component of the color.

9.8.4.6.2 F32: Green

Green is the green component of the color.

9.8.4.6.3 F32: Blue

Blue is the blue component of the color.

9.8.4.7 F32: Reflectivity

Reflectivity measures how shiny a material appears to be. Specular reflections are calculated based on the light position, surface normal, and camera position. The result is then raised to an exponent to control specular light falloff. The exponent is determined from the reflectivity value. The normal range of reflectivity is 0.0 (exponent of zero disables specular lighting) to 1.0 (exponent of 128). Values outside this range are allowed but are clamped before use.

9.8.4.8 F32: Opacity

Opacity is a measure of on object's transparency. The value is used when alpha blending. Higher Opacity means the object is less transparent and objects behind it will be less visible. The normal range of opacity is 0.0 (invisible) to 1.0 (completely opaque). Values outside this range are allowed.

9.8.5 Texture Resource (blocktypes: 0xFFFFFF55; 0xFFFFFF5C)

The Texture Resource contains information for creating a texture image to be applied to geometry. The usage of the texture resource is controlled by a shader. The texture resource is divided into two parts: the declaration and the continuation. The texture declaration contains information for creating the texture resource object and allocating memory. The texture continuation contains image data for the texture.

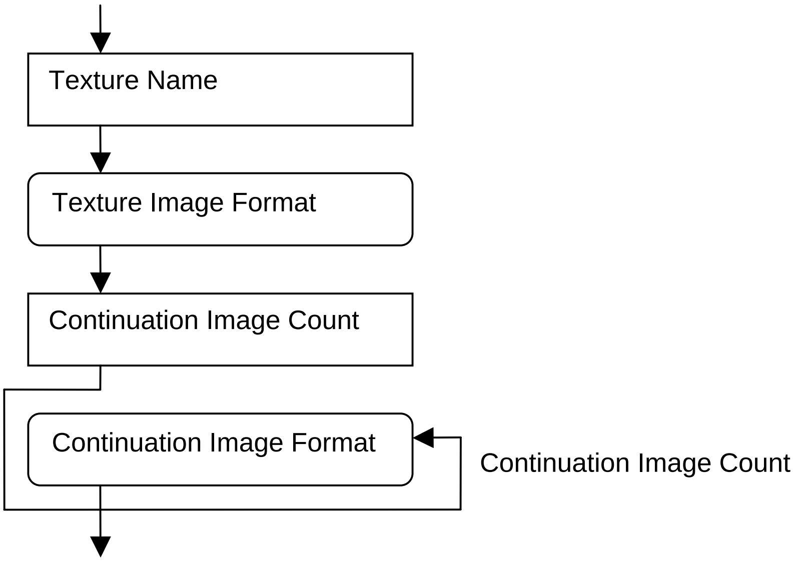
The texture image may be created by composing more than one continuation image. For example, an RGBA texture image may be created by composing an RGB continuation image with an Alpha continuation image.

A continuation image may be contained in one or more continuation blocks in the same U3D file as the declaration block.

As an alternative, the continuation image may be contained in an external referenced image file.

9.8.5.1 Texture Declaration (blocktype: 0xFFFFFF55)

The Texture Declaration describes the texture image and the continuation images.



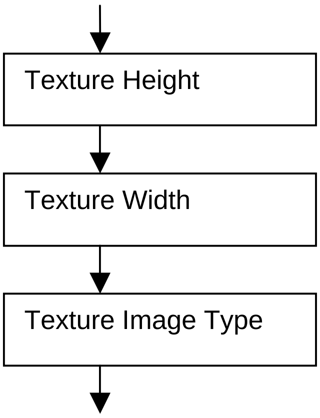
9.8.5.1.1 String: Texture Name

The Texture Name is the name used to identify the texture.

9.8.5.1.2 Texture Image Format

Texture Image Format describes the size (height and width) of the texture image and the format of the texture data.

A rendering system may convert the texture image to a different size (height and width) for rendering. For example, the rendering system may filter and re-size the image because it is too big or the dimensions are not a power of two. Such rendering details are outside the scope of this specification.



9.8.5.1.2.1 U32: Texture Height

Texture Height is the height of the texture in pixels. Texture Height shall be greater than zero.

9.8.5.1.2.2 U32: Texture Width

Texture Width is the width of the texture in pixels. Texture Width shall be greater than zero.

9.8.5.1.2.3 U8: Texture Image Type

Texture Image Type identifies the color channels present in the texture image. The valid values are:

0x01 – alpha component

0x0E – color RGB (red, green, and blue)

0x0F – color RGBA (red, green, blue, and alpha)

0x10 – luminance (greyscale)

0x11 – luminance and alpha (greyscale and alpha)

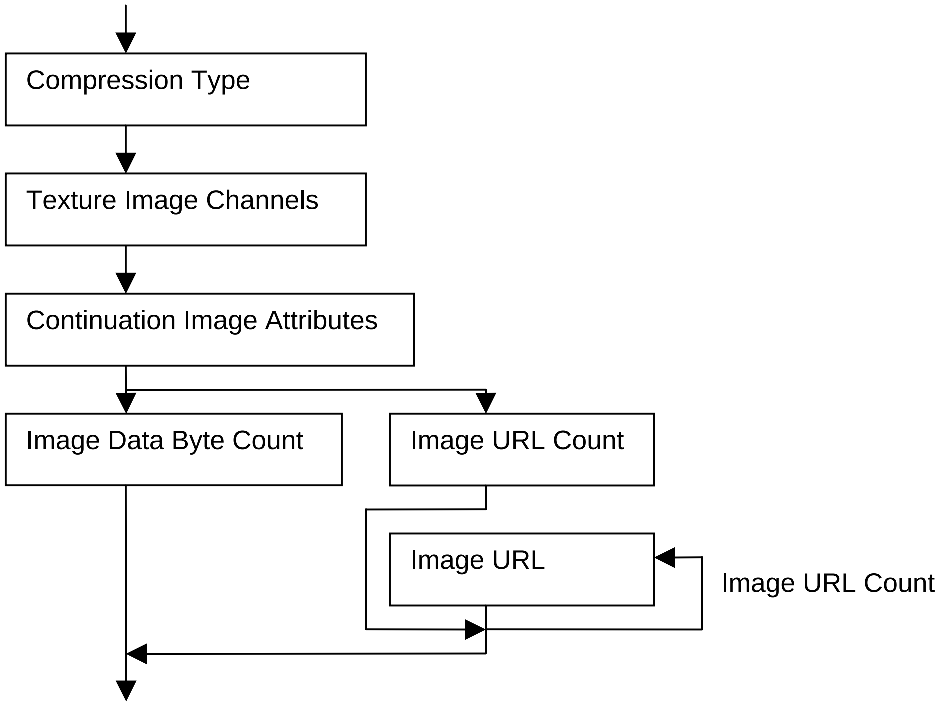
All other values are reserved.

9.8.5.1.3 U32: Continuation Image Count

Continuation Image Count is the number of continuation images used to compose the texture image. This count is not the number of texture continuation blocks because each continuation image is contained in one or more blocks. The index into the sequence of continuation image formats that follows is used by the continuation blocks to indicate which continuation image their image data is for.

9.8.5.1.4 Continuation Image Format

Continuation Image Format provides some information about the continuation image in the texture declaration.



9.8.5.1.4.1 U8: Compression Type

Compression Type defines the scheme used to compress the Image Data in the texture continuation blocks. The types are:

0x01 – JPEG-24 (color, baseline profile)

0x02 – PNG

0x03 – JPEG-8 (greyscale, baseline profile)

0x04 – TIFF

9.8.5.1.4.2 U8: Texture Image Channels

Texture Image Channels indicates which color channels of the texture image are composed using this continuation image. The texture image channel bits can be combined using the OR operator. A particular texture image channel can be composed from only one continuation image. The values for the texture image channel bits are:

0x01: alpha channel

0x02: blue channel

0x04: green channel

0x08: red channel

0x10: luminance (red, blue and green channels)

9.8.5.1.4.3 U16: Continuation Image Attributes

Continuation Image Attributes contains additional information about the continuation image. All other values are reserved.

0x0000: default attributes

0x0001: external continuation image file reference

By default, the continuation image data is contained in texture continuation blocks in the same U3D file as the texture declaration block. If the external continuation image file reference bit is set, then the continuation image data is contained in an external file.

9.8.5.1.4.4 U32: Image Data Byte Count

Image Data Byte Count is the sum of the number of bytes of Image Data in all the continuation blocks for this continuation image. This value can be useful for setting up an image decoder and for determining when all of the image data is available for decoding. Image Data Byte Count is not present if the external continuation image file reference bit is set.

9.8.5.1.4.5 U32: Image URL Count

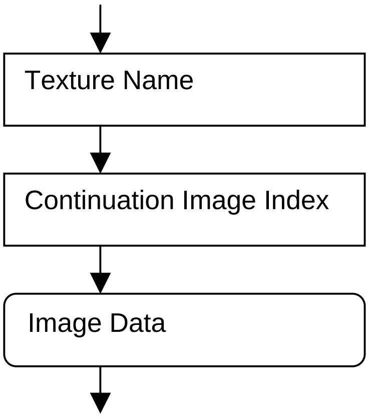
Image URL Count is the number of URL strings that follow. Image URL Count is only present if the external continuation image file reference bit is set.

9.8.5.1.4.6 String: Image URL

Image URL is a String identifying the external image file location. Multiple locations can be specified for the external file. A loader shall load the image file from one of the locations. HTTP and FTP protocols will be recognized with absolute and relative addressing. Image URL is only present if the Image URL Count is greater than zero and the external continuation image file reference bit is set.

9.8.5.2 Texture Continuation (blocktype: 0xFFFFFF5C)

The Texture Continuation contains image data for a continuation image previously described in the texture declaration.



9.8.5.2.1 String: Texture Name

Texture Name is the name of the texture resource with which this continuation block is associated.

9.8.5.2.2 U32: Continuation Image Index

This block's image data is used to decode the continuation image indicated by Continuation Image Index. This value is an index into the sequence of continuation image formats in the texture declaration.

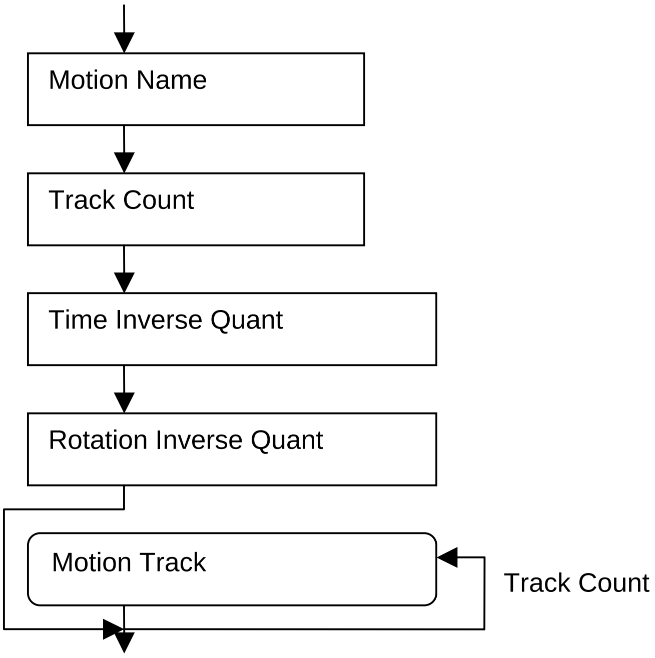
9.8.5.2.3 Image Data

The Image Data is the data for the continuation image used for a texture. The format of the image data is indicated by Compression Type in the texture declaration. The size of the compressed image data can be determined by subtracting the size of Texture Name and Continuation Image Index from the size of the data section. The image data can be contained in multiple texture continuation blocks with the same Texture Name and Continuation Image Index. Spreading the image data across several blocks is particularly useful when used with a progressive compressed image format.

The setting of the no compression mode bit in 9.4.1.2 Profile Identifier does not affect the encoding of Image Data.

9.8.6 Motion Resource (blocktype: 0xFFFFFF56)

The motion resource contains animation data. The data is stored in a number of tracks. Each track is composed of key frames with rotation, displacement and time information. A motion track can be used to animate a bone in a bone hierarchy. A motion track can also be used to animate a node in the scene graph.



9.8.6.1 String: Motion Name

Motion Name is the name of this motion resource.

9.8.6.2 U32: Track Count

Track Count is the number of motion tracks in this motion resource.

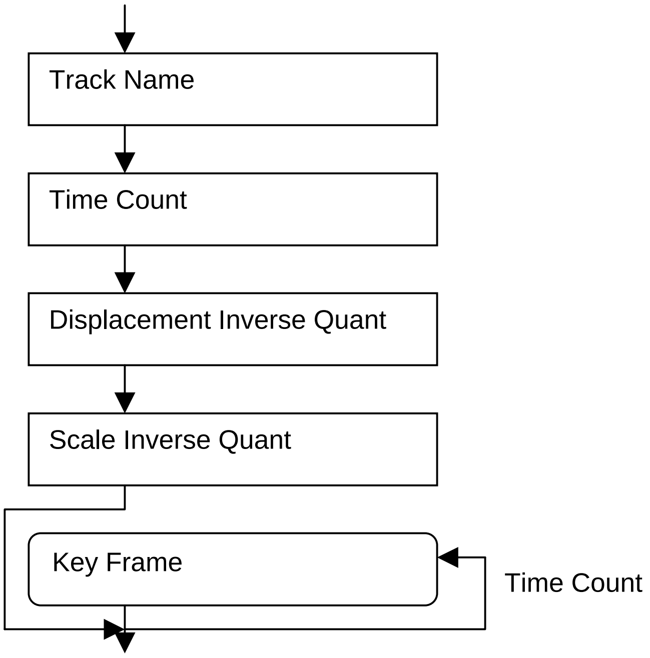
9.8.6.3 F32: Time Inverse Quant

Time Inverse Quant is the inverse quantization factor for time values.

9.8.6.4 F32: Rotation Inverse Quant

Rotation Inverse Quant is the inverse quantization factor for rotation values.

9.8.6.5 Motion Track



9.8.6.5.1 String: Track Name

Track Name is the name of this motion track.

9.8.6.5.2 U32: Time Count

Time Count is the number of time samples for this motion track.

9.8.6.5.3 F32: Displacement Inverse Quant

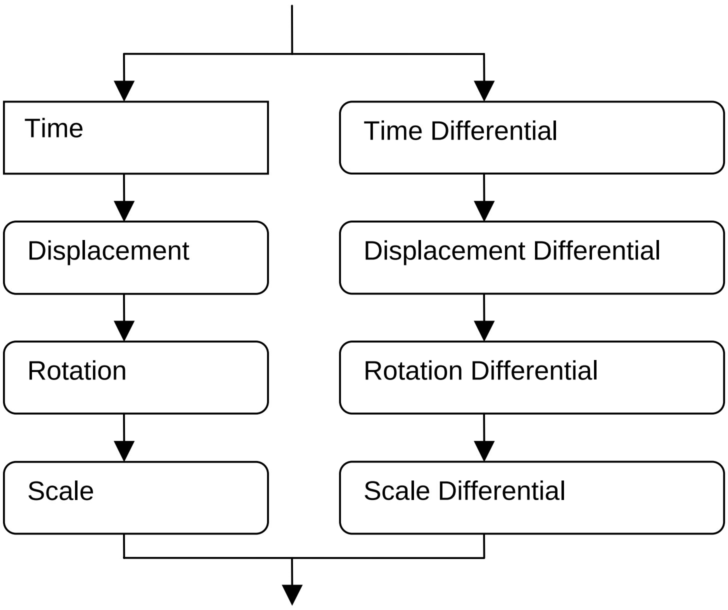
Displacement Inverse Quant is the inverse quantization factor for displacement values for this motion track.

9.8.6.5.4 F32: Scale Inverse Quant

Scale Inverse Quant is the inverse quantization factor for scale values for this motion track.

9.8.6.5.5 Key Frame

The motion track has one Key Frame for each time sample. The first and last Key Frames use unquantized values and all other key frames used quantized differential values.

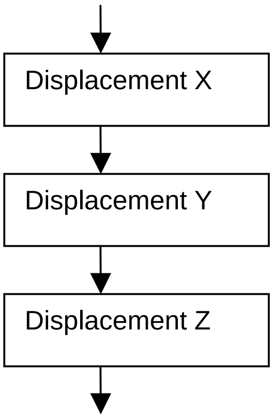


9.8.6.5.5.1 F32: Time

Time is the time value for this Key Frame.

9.8.6.5.5.2 Displacement

Displacement is the translation of the start of the bone from the end of its parent bone. For a root bone or for a node, Displacement is the translation from the origin of the local coordinate space.



9.8.6.5.5.2.1 F32: Displacement X

Displacement X is the X coordinate of the Displacement.

9.8.6.5.5.2.2 F32: Displacement Y

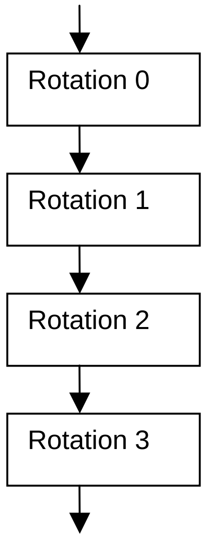
Displacement Y is the Y coordinate of the Displacement.

9.8.6.5.5.2.3 F32: Displacement Z

Displacement Z is the Z coordinate of the Displacement.

9.8.6.5.5.3 Rotation

Rotation is the change in orientation of the bone relative to the parent bone. Rotation is expressed as a quaternion with the real part first.



9.8.6.5.5.3.1 F32: Rotation 0

Rotation 0 is the real part of the Rotation quaternion.

9.8.6.5.5.3.2 F32: Rotation 1

Rotation 1 is the coefficient for i in the Rotation quaternion.

9.8.6.5.5.3.3 F32: Rotation 2

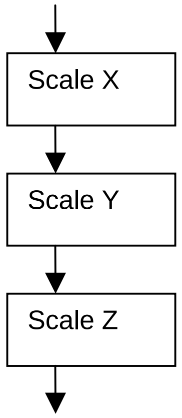
Rotation 2 is the coefficient for j in the Rotation quaternion.

9.8.6.5.5.3.4 F32: Rotation 3

Rotation 3 is the coefficient for k in the Rotation quaternion.

9.8.6.5.5.4 Scale

Scale is the scaling component of the transformation of the bone relative to its parent bone.



9.8.6.5.5.4.1 F32: Scale X

Scale X is the scaling factor in the X dimension.

9.8.6.5.5.4.2 F32: Scale Y

Scale Y is the scaling factor in the Y dimension.

9.8.6.5.5.4.3 F32: Scale Z

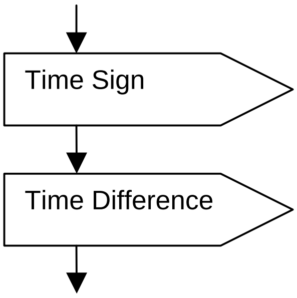
Scale X is the scaling factor in the Z dimension.

9.8.6.5.5.5 Time Differential

Time Differential is the quantized difference between the actual time value and the predicted time value. The reconstructed time value from the previous Key Frame is used as a prediction for this Key Frame.

The reconstructed time is calculated as

reconstructed time = InverseQuant(predicted time,Time Sign,Time Difference,Time Inverse Quant).



9.8.6.5.5.5.1 U8 [cTimeSign]: Time Sign

Time Sign contains the sign bits for the time prediction difference.

0x00 – the prediction difference is positive or zero

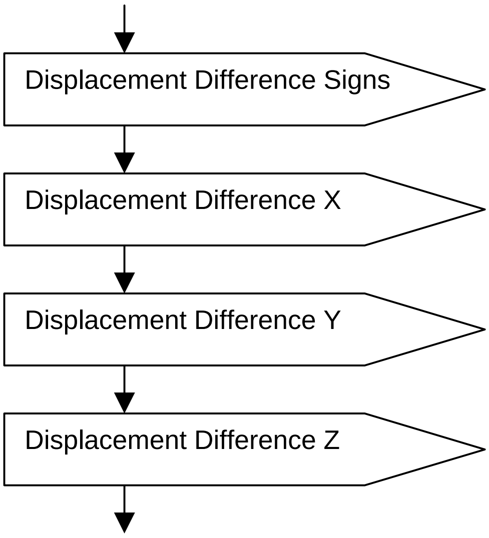
0x01 – the prediction difference is negative

9.8.6.5.5.5.2 U32 [cTimeDiff]: Time Difference

Time Difference is the quantized absolute prediction difference for the time value.

9.8.6.5.5.6 Displacement Differential

Displacement Differential is the quantized difference between the actual displacement value and the predicted displacement value. The reconstructed displacement value from the previous Key Frame is used as a prediction for this Key Frame.


Displacement Difference Signs
Displacement Difference X
Displacement Difference Y
Displacement Difference Z

9.8.6.5.5.6.1 U8 [cDispSign]: Displacement Difference Signs

Displacement Difference Signs is a collection of sign bits for the prediction differences.

0x01 – Sign bit for Displacement Difference X

0x02 – Sign bit for Displacement Difference Y

0x04 – Sign bit for Displacement Difference Z

9.8.6.5.5.6.2 U32 [cDispDiff]: Displacement Difference X

reconstructed X = InverseQuant(predicted X, (DisplacementDifferenceSigns & 0x01),
Displacement Difference X, Displacement Inverse Quant).

9.8.6.5.5.6.3 U32 [cDispDiff]: Displacement Difference Y

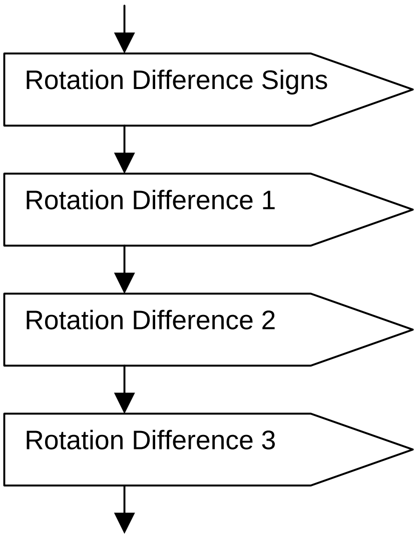
reconstructed Y = InverseQuant(predicted Y, ((DisplacementDifferenceSigns & 0x02)>>1),
Displacement Difference Y, Displacement Inverse Quant).

9.8.6.5.5.6.4 U32 [cDispDiff]: Displacement Difference Z

reconstructed Z = InverseQuant(predicted Z, ((DisplacementDifferenceSigns & 0x04)>>2),
Displacement Difference Z, Displacement Inverse Quant).

9.8.6.5.5.7 Rotation Differential

The reconstructed Rotation quaternion from the previous Key Frame is used as the prediction for this Key Frame. The reconstructed Rotation quaternion for this Key Frame is obtained by multiplying the prediction quaternion by the reconstructed quaternion difference.



9.8.6.5.5.7.1 U8 [cRotSign]: Rotation Difference Signs

Rotation Difference Signs is a collection of sign bits for the prediction differences.

0x01 – Sign bit for Rotation Difference 0

0x02 – Sign bit for Rotation Difference 1

0x04 – Sign bit for Rotation Difference 2

0x08 – Sign bit for Rotation Difference 3

9.8.6.5.5.7.2 U32 [cRotDiff]: Rotation Difference 1

The coefficient for i in the reconstructed quaternion difference is calculated as

RQD1 = InverseQuant(0,((RotationDifferenceSigns&0x02)>>1),Rotation Difference 1,Rotation Inverse Quant)

9.8.6.5.5.7.3 U32 [cRotDiff]: Rotation Difference 2

The coefficient for j in the reconstructed quaternion difference is calculated as

RQD2 = InverseQuant(0,((RotationDifferenceSigns&0x04)>>2),Rotation Difference 2,Rotation Inverse Quant)

9.8.6.5.5.7.4 U32 [cRotDiff]: Rotation Difference 3

The coefficient for k in the reconstructed quaternion difference is calculated as

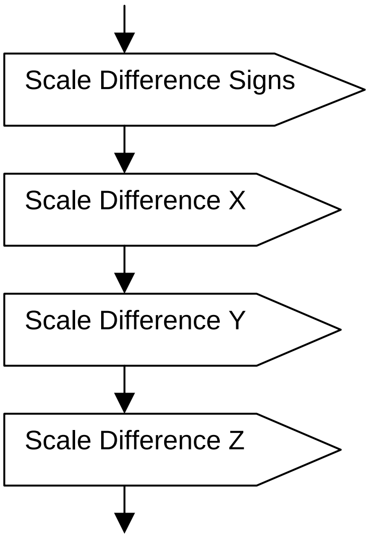
RQD3 = InverseQuant(0,((RotationDifferenceSigns&0x08)>>3),Rotation Difference 3,Rotation Inverse Quant)

The real part of the reconstructed quaternion difference is calculated as

$$RQD0 = (1 - 2 \cdot (RotationDifferenceSigns \& 0x01)) \cdot \sqrt{1.0 - (RQD1^2 + RQD2^2 + RQD3^2)}$$

9.8.6.5.5.8 Scale Differential

Scale Differential is the quantized difference between the actual scale value and the predicted scale value. The reconstructed scale value from the previous Key Frame is used as a prediction for this Key Frame.



9.8.6.5.5.8.1 U8 [cScalSign]: Scale Difference Signs

Scale Difference Signs is a collection of sign bits for the prediction differences.

0x01 – Sign bit for Scale Difference X

0x02 – Sign bit for Scale Difference Y

0x04 – Sign bit for Scale Difference Z

9.8.6.5.5.8.2 U32 [cScalDiff]: Scale Difference X

reconstructed X = InverseQuant(predicted X, (ScaleDifferenceSigns & 0x01),
Scale Difference X, Scale Inverse Quant).

9.8.6.5.5.8.3 U32 [cScalDiff]: Scale Difference Y

reconstructed Y = InverseQuant(predicted Y, ((ScaleDifferenceSigns & 0x02)>>1),
Scale Difference Y, Scale Inverse Quant).

9.8.6.5.5.8.4 U32 [cScalDiff]: Scale Difference Z

reconstructed Z = InverseQuant(predicted Z, ((ScaleDifferenceSigns & 0x04)>>2),

Scale Difference Z, Scale Inverse Quant).

10 Bit Encoding Algorithm

10.1 Definitions

The following definitions are applicable to the U3D bit encoding algorithm.

Term	Definition
Clear	Set the bits to 0.
Context	Captures the estimated probability distribution of the current symbol of a particular type.
Cumulative frequency of a symbol	Sum of the occurrences of all of the symbols less than current symbol stored in the histogram for a given context.
Dynamic context	Estimated probability is based on previous occurrences of symbols.
Dynamic Histogram	Used to store the frequency counts of symbols written by the encoding algorithm in a dynamic context. The values written have a corresponding symbol equal to the value + 1. The symbol 0 is the escape symbol and has an initial frequency of 1. All other symbols have an initial frequency of 0. A dynamic histogram is updated as values are written by the bit encoding algorithm.
Frequency of the symbol	Number of occurrences of a symbol stored in the histogram for a given context.
High	High probability limit.
Low	Low probability limit.
Set	Assign the bits to the specified value (if no value specified, value is 1)
Static context	Estimated probability is not based on previous occurrences of symbols, i.e. all valid symbols have equal probability.
Static Histogram	Used by the encoding algorithm to encode symbols. The static histogram is defined by a number R that represents a range of valid values from 0 to R - 1. Each value has a corresponding symbol equal to the value + 1. Each symbol has a frequency of 1. The symbol 0 is the escape symbol used by the encoding algorithm and has a frequency of 0 for all static histograms. Static Histograms are constant.
Total cumulative frequency	Total occurrences of all symbols including the escape symbol stored in the histogram for a given context.
Underflow bits	A count of the series of bits that are the same between the high and low probabilities starting with the QBits of high and low and comparing until a mismatch is found (due to the fixed storage size for Low and High).
Write-bit	Write a bit in order with least significant bit of the byte written first to the output stream.

10.2 Acronyms and Abbreviations

The following acronyms and abbreviations are applicable to the U3D bit encoding algorithm.

Acronym	Description
HBit	'half' bit representing 0.5 in the fixed-point 16.16 format (in a 32-bit word this is the 17th most significant bit)
LBit	Least significant bit
MBit	Most significant bit
QBit	'quarter' bit representing 0.25 in the fixed-point 16.16 format (in a 32-bit word this is the 18th most significant bit)

10.3 Overview

This section provides an overview of the U3D bit encoding algorithm. The bit encoding algorithm shall be used to write all U3D files. The bit encoding algorithm defines a platform independent representation of the binary data that is a U3D file.

The bit encoding algorithm is a single pass statistical data compression method using an arithmetic algorithm that transforms the input into a single floating point number between 0 and 1. The encoding performed is lossless, i.e. data is not discarded during the encoding process and the original data is obtained after the decoding process. All data blocks as defined in Clause 9 of the specification are processed through the encoder. Only encoding is specified as normative.

The compression algorithm supports compression of unsigned integers. When a value is to be compressed and encoded, a compression context must be specified. The compression context determines which histogram is used to encode the value. The algorithm may use multiple histograms to encode a series of values. Clause 9 identifies the values that are compressed and the contexts for the compression. Compression is performed for parts of the file that are expected to contain large amounts of compressible data, e.g. geometry and animation.

The algorithm uses static or dynamic histograms for the encoding. The static histograms represent a uniform distribution over a range of numbers. The dynamic histograms build the distribution from the values written using a context. The dynamic contexts are used for values that are expected to have a narrow distribution.

10.3.1 Prerequisites and Inputs

The algorithm accepts as input U8, U16, U32, U64, I32, and F32 values. All uncompressed values are cast to unsigned integers and written as a sequence of compressed U8 using a static context with range 0-255. The algorithm transforms compressed U8, U16, and U32 values into symbols based on the context given.

10.3.2 Description

When encoding a value, a context with an associated histogram is specified. Static histograms are used when the distribution of the encoding numbers is expected to be roughly even or very random and difficult to predict. Dynamic contexts should be used to encode data that tends to cluster around a few values and is easy to predict.

The bit encoding algorithm transforms all of the input value into a sequence of one or more symbols. All of the uncompressed values written are cast to unsigned integer values with the same number of bytes as the original type, and then broken into a series of unsigned 8-bit values. The 8-bit values are then written as symbols using a static context with a range from 0 to 255.

The symbols written by the algorithm are represented as a floating point number. Each value in the histogram is given a portion of the numbers between 0 and 1. The size of the range allocated to each value corresponds to the probability the value will appear. Each value in a static context has the same probability and the same range size. For dynamic contexts, the probability is based on the number of times that the symbol has been written to that context. When writing a symbol, the encoding algorithm encodes the range that corresponds to that value. Encoding a sequence of values involves encoding the range of the value to be written with in the range of the previous value. Because the encoding algorithm is very order dependent, all of the steps must be performed in the order described in the algorithm below.

10.4 Encoding Algorithm

This section describes the details and the normative requirements for the bit encoding algorithm. Specifically, only the encoding algorithm is specified in this clause since either one of the encoding algorithm or the decoding algorithm needs to be specified normatively. Refer to Annex A for an example implementation of the bit encoding algorithm.

Note that the algorithm follows standard rules for operator precedence in arithmetic expressions.

Static contexts are specified by the range from: 0 to (R-1), maximum value for R is: 0x3FFE (16382). If R is larger than this value an uncompressed U16 or U32 is written, as appropriate.

10.4.1 General Requirements

This section specifies general requirements for the bit encoding algorithm.

1. All blocks shall be processed through the bit encoding algorithm
2. Encoding procedure shall depend on the type of the block (as defined in Clause 9 File Format)

10.4.2 Operations

The operations of the algorithm shall be performed as in the following table:

Type of value to be written to output stream	*How value is written*
Compressed U32	Refer to algorithm in section 10.4.5 below
Compressed U16	Refer to algorithm in section 10.4.6 below
Compressed U8	Refer to algorithm in section 10.4.7 below
U8	Refer to algorithm in section 10.4.8 below
U16	Low order U8 followed by high order U8
U32	Low order U16 followed by high order U16
U64	Low order U32 followed by high order U32
I32	Memory pattern of 2’s complement signed integer interpreted as U32
F32	Memory pattern of IEEE 32-bit format interpreted as U32
String	U16 count of U8s in string followed by the U8s in the string

10.4.3 Initialization

The bit encoding algorithm shall perform the following initializations:

high probability limit = 0x0000FFFF (represents 1.0 as the fixed point equivalent of the binary repeating number 0.111…)

low probability limit = 0

underflow count = 0

initial histogram for a dynamic context: escape symbol frequency = 1 and all other frequencies = 0 (the histogram is modified as symbols are written to the context) histogram for a static context: escape symbol frequency = 0, all symbols <= R the frequency = 1 and frequency for all other symbols = 0

10.4.4 Algorithm for Writing a Compressed Symbol

The procedure for how symbols shall be written by the bit encoding algorithm are detailed in the following subsections.

Inputs: value to be written and context

10.4.4.1 Obtain Frequency Values

set symbol = value + 1

Obtain total cumulative frequency of all symbols for this context

Obtain total cumulative frequency of this symbol for this context

Obtain frequency of this symbol for this context

If frequency of this symbol for this context is 0 then prepare to write the escape symbol as follows:

Set symbol = 0

Obtain total cumulative frequency of this symbol for this context.

NOTE
Total cumulative frequency of the escape symbol for all contexts is 0.

Obtain frequency of this symbol for this context

10.4.4.2 Update the Probability Limits

probability range = High - Low + 0x00000001

Update the high probability limit: High = Low + probability range * (cumulative frequency of symbol + frequency of the symbol) / (total cumulative frequency of all symbols) – 0x00000001

Update the low probability limit: Low = Low + (probability range * cumulative frequency of symbol / total cumulative frequency of all symbols)

10.4.4.3 Update the Compression Context

Update the compression context with this symbol. Note that this symbol may be the escape symbol

10.4.4.4 Write to Output Stream Based on Current Probability Range

```
WHILE Hbit of High and Low are same
    set output bit = Hbit
    write-bit output bit
    clear HBit for High, left shift High, and set LBit of High
    clear Hbit for Low, left shift Low, and clear LBit of Low
    WHILE underflow count > 0
        write-bit NOT(output bit)
        decrement underflow count
    END WHILE
END WHILE
```

10.4.4.5 Determine Underflow Count

```
WHILE QBit Low is same as QBit High
    clear Hbit for High, left shift High, set LBit High and set Hbit High
clear Hbit for Low, left shift Low, clear LBit Low and clear Hbit Low
increment underflow count
END WHILE
```

10.4.4.6 Return

Return either success or a warning that an escape value was written

10.4.5 Algorithm for Writing a Compressed U32 Value

```
if context is static with (R > maximum R)
then
write uncompressed U32
else
result = write value as compressed symbol
if result is a warning that an escape was written
then
write value as an uncompressed U32
update the histogram for this context with value + 1
```

10.4.6 Algorithm for Writing a Compressed U16 Value

```
if context is static with (R > maximum R)
then
write uncompressed U16
else
result = write value as compressed symbol
if result is a warning that an escape was written
then
write value as an uncompressed U16
update the histogram for this context with value + 1
```

10.4.7 Algorithm for Writing a Compressed U8 Value

```
result = write value as compressed symbol
if result is a warning that an escape was written
then
write value as an uncompressed U8
update the histogram for this context with value + 1
```

NOTE
For static contexts, the maximum value of R = 256.

10.4.8 Algorithm for Writing an Uncompressed U8 Value

```
set symbol = value with bit order swapped, i.e. reverse order of bits (most significant bit exchanged with least significant bit and so on…)
write symbol as compressed symbol with static context R=256
```

10.4.9 Algorithm for Updating the Compression Context

A histogram stores a limited number of symbol occurrences. When the total number of symbol occurrences = 0x1FFF all of symbol frequencies in the histogram shall be divided by 2 rounding down. The frequency count of the escape symbol is then incremented by 1.

NOTE
Limiting the number of occurrences stored allows the histogram to adapt to changing distributions within a given context. More recent values will have a greater influence on the histogram for a given context. The value 0x1FFF was determined empirically. Larger values for this number allow for more efficient compression for stable probability distributions, whereas, smaller values enable faster adaptation to changing probability distributions. Additionally, values larger than 0x1FFF may cause numeric overflow issues on some 32-bit hardware.

10.4.10 Algorithm for Flushing the Compression State

If one or more compressed values have been written then an uncompressed U32 value of 0 shall be written in order to ensure that all the bits required for decoding are written to the output stream.

Annex A
(informative)

Bit Encoding Algorithm – An Implementation

A.1 Introduction

This Annex provides an example implementation of the compression algorithm used to encode the fields defined in Clause 9. An implementation of the corresponding decompression algorithm is also provided. The algorithms are described in the C# Language (ECMA-334). It is highly recommended that all implementations of the compression algorithm presented with the same input sequence of values and compression contexts produce the same output sequence of bytes. Refer to Clause 10 for the normative requirements of the bit encoding algorithm.

The classes defined in A.3 define the compression and decompression algorithms. The interfaces supported by these classes are provided in A.2.

The `BitStreamWrite` class is used to encode the bits for a sequence of values. Compressed write methods accept a compression context parameter in addition to the value to be written. The compression context is used to estimate the probability distribution of values. These probability estimates are provided by the `ContextManager` class and used by `BitStreamWrite` to attempt to reduce the number of bits required to store the value.

The `BitStreamRead` class is used to recreate the sequence of values from the encoded bits. The `BitStreamRead` class uses the `ContextManager` class to re-create the same probability estimates as used by the `BitStreamWrite` class.

The `ContextManager` class supports probability estimates that are based on the range of possible value or based on an adapting history of previous values encountered.

The `DataBlock` class is used as a container to hold the encoded bits. The `DataBlock` class is also used to hold the other fields defined in 9.2 for the block structure.

The `Constants` class provides names for certain constant values used in the other classes.

A.1.1 Usage

The usual writing operation would make several calls to the write methods on the interface of the `BitStreamWrite` class. After all writing is done, the `DataBlock` would be retrieved from the `BitStreamWrite` class. The `BitStreamWrite` class provides the encoding for the Data field (9.2.4) in the block format. Additional fields such as block type and meta data would be modified and then the `DataBlock` would be stored in a file according to the format in 9.2.

The usual reading operation would read a block from the file into a `DataBlock` class. To interpret the Data field of the block, the `DataBlock` class would be provided to a `BitStreamRead` class after which several calls to the read methods on `BitStreamRead` would provide the encoded values.

For correct results, the sequence of context parameters used by the reading operation must be the same as the sequence used by the writing operation.

A.2 Interfaces

A.2.1 Bit Stream Write

```
using System;

namespace U3D
```

```
{
    /// <summary> IBitStreamWrite.cs
    /// This file defines the IBitStreamWrite interface and the associated
    /// identifier. IBitStreamWrite is used to write compressed and
    /// uncompressed data to a datablock.
    /// </summary>
    /// <remarks>
    /// <para> The IBitStreamWrite is supported by the BitStreamWrite class.
    /// </para>
    /// <para> Bytes are written in little-endian order.
    /// </para>
    /// </remarks>
    public interface IBitStreamWrite
    {
        /// <summary>Write a U8 to the datablock.
        /// </summary>
        /// uValue <param name = "uValue">
        /// the value to write to the datablock
        /// </param>
        /// reuturn <returns>
        /// void</returns>
        void WriteU8(Byte uValue);

        /// <summary>Write a U16 to the datablock.
        /// </summary>
        /// uValue <param name = "uValue">
        /// the value to write to the datablock
        /// </param>
        /// returns <returns>
        /// void</returns>
        void WriteU16(UInt16 uValue);

        /// <summary>Write a U32 to the datablock.
        /// </summary>
        /// uValue <param name = "uValue">
        /// the value to write to the datablock
        /// </param>
        /// returns <returns>
        /// void</returns>
```

```
void WriteU32(UInt32 uValue);

/// <summary>Write a U64 to the datablock.
/// </summary>
/// uValue <param name = "uValue">
/// the value to write to the datablock
/// </param>
/// return <returns>
/// void</returns>
void WriteU64(UInt64 uValue);

/// <summary>Write an I32 to the datablock.
/// </summary>
/// iValue <param name = "iValue">
/// the value to write to the datablock
/// </param>
/// returns <returns>
/// void</returns>
void WriteI32(Int32 iValue);

/// <summary>Write a F32 datablock.
/// </summary>
/// fValue <param name = "fValue">
/// the value to write to the datablock
/// </param>
/// returns <returns>
/// void</returns>
void WriteF32(Single fValue);

/// <summary>Write a compressed U32 to the datablock.
/// </summary>
/// context <param name = "context">
/// the context to use for the arithmetic encoder.
/// </param>
/// uValue <param name = "uValue">
/// the value to compress and write to the datablock
/// </param>
/// returns <returns>
/// void</returns>
```

```
void WriteCompressedU32(UInt32 context, UInt32 uValue);

/// <summary>Write a compressed U16 to the datablock.
/// </summary>
/// context <param name = "context">
/// the context to use for the arithmetic encoder.
/// </param>
/// uValue <param name = "uValue">
/// the value to compress and write to the datablock
/// </param>
/// return <returns>void</returns>
void WriteCompressedU16(UInt32 context, UInt16 uValue);

/// <summary>Write a compressed U8 to the datablock.
/// </summary>
/// context <param name = "context">
/// the context to use for the arithmetic encoder.
/// </param>
/// uValue <param name = "uValue">
/// the value to compress and write to the datablock
/// </param>
/// return <returns>
/// void</returns>
void WriteCompressedU8(UInt32 context, Byte uValue);

/// <summary>Stores the data written by the bit stream writer
/// in a datablock.
/// </summary>
/// rDataBlock <param name = "rDataBlock">
/// returns the data written by the BitStreamWriter in a datablock
/// </param>
/// return <returns>
/// void</returns>
void GetDataBlock(out IDataBlock rDataBlock);

/// <summary>Set the current position to the next byte boundary
/// </summary>
/// return <returns>
/// void</returns>
```

```
            void AlignToByte();

            /// <summary>Set the current position to the next 4 byte boundary
            /// </summary>
            /// return <returns>
            /// void</returns>
            void AlignTo4Byte();

      }
}
```

A.2.2 Bit Stream Read

```
using System;
namespace U3D
{
      /// <summary>IBitStreamRead.cs
      /// This file defines the IBitStreamRead interface and the associated
      /// identifier. IBitStreamRead is used to read compressed and uncompressed
      /// data to a data block.
      /// </summary>
      /// <remarks>
      /// <para>The IBitStreamRead is supported by the BitStreamRead class.
      /// </para>
      /// <para>Bytes are read in little-endian order.
      /// </para>
    /// </remarks>
      public interface IBitStreamRead
      {
            /// <summary>Read a U8 from the datablock associated with this
            ///  bitstream.
            /// </summary>
            /// rValue <param name = "rValue"><description>
            /// the value read is returned in rValue.</description>
            /// </param>
            /// return <returns>
            /// void</returns>
            void ReadU8(out Byte rValue);

            /// <summary>Read a U16 from the datablock.
            /// </summary>
/// rValue <param name = "rValue"><description>
/// the value read is returned in rValue</description>
/// </param>
/// return <returns>
/// void</returns>
void ReadU16(out UInt16 rValue);

/// <summary>Read a U32 from the datablock.
/// </summary>
/// rValue <param name = "rValue"><description>
/// the value read is returned in rValue</description>
/// </param>
/// return <returns>
/// void</returns>
void ReadU32(out UInt32 rValue);

/// <summary>Read a U64 from the datablock.
/// </summary>
/// rValue <param name = "rValue"><description>
/// the value read is returned in rValue</description>
/// </param>
/// return <returns>
/// void</returns>
void ReadU64(out UInt64 rValue);

/// <summary>Read a I32 from the datablock.
/// </summary>
/// rValue <param name = "rValue"><description>
/// the value read is returned in rValue</description>
/// </param>
/// return <returns>
/// void</returns>
void ReadI32(out Int32 rValue);

/// <summary>Read a F32 from the datablock.
/// </summary>
/// rValue <param name = "rValue"><description>
/// the value read is returned in rValue</description>
/// </param>
/// return <returns>
/// void</returns>
void ReadF32(out Single rValue);

/// <summary>Read a compressed U32 from the datablock.
/// </summary>
/// context <param name="context">
/// the context used to interpret the compressed value
/// </param>
/// rValue <param name = "rValue"><description>
/// the value read is returned in rValue</description>
/// </param>
/// return <returns>
/// void</returns>
void ReadCompressedU32(UInt32 context, out UInt32 rValue);

/// <summary>Read a compressed U16 from the datablock.
/// </summary>
/// context <param name="context">
/// the context used to interpret the compressed value
/// </param>
/// rValue <param name = "rValue"><description>
/// the value read is returned in rValue</description>
/// </param>
/// return <returns>
/// void</returns>
void ReadCompressedU16(UInt32 context, out UInt16 rValue);

/// <summary>Read a compressed U8 from the datablock.
/// </summary>
/// context <param name="context">
/// the context used to interpret the compressed value
/// </param>
/// rValue <param name = "rValue"><description>
/// the value read is returned in rValue</description>
/// </param>
/// return <returns>
/// void</returns>
void ReadCompressedU8(UInt32 context, out Byte rValue);
```

```
                /// <summary>Set the data that is read by the BitStreamReader.
                /// </summary>
                /// <param name = "dataBlock">the data that is to be read
                /// </param>
                /// <returns>void</returns>
                void SetDataBlock(IDataBlock dataBlock);

        }
}
```

A.2.3 Context Manager

```
using System;
namespace U3D
{

        /// <summary>IContextManager.cs
        ///
        /// This file defines the IContextManager interface.
        /// IContextManager is used to access the static and dynamic contexts used
        /// for the reading and writing of compressed data.
        /// </summary>
        /// <remarks>
        /// <para> Dynamic Context: dynamic contexts are specified as 0x0001
        /// through 0x3FFF.  Dynamic contexts keep a histogram that stores
        /// the number of occurrences of symbols that are added through the
        /// AddSymbol method.
        /// </para>
        /// <para> Static Context: static contexts are specified as 0x4000
        /// through 0x7FFF.  Static contexts represent histograms where each
        /// value between 0 and (context - 0x4000) are equally likely.  Static
        /// contexts histograms are not changed by the AddSymbol method.
        /// </para>
        /// <para> Context 0 or Context8: context 0 is a shortcut to context
        /// 0x40FF which corresponds to values from 0 through 255.
        /// </para>
        /// <para> When a histogram for a dynamic context is initialized,
        /// the symbol frequency of the escape symbol is initialized to 1.
        /// </para>
        /// <para> Symbols larger than 0xFFFF are treated as static.
```

```
    ///   </para>
    /// <para> The IContextManager interface is supported by the
    /// ContextManager class.
    /// </para>
    /// </remarks>
    public interface IContextManager
    {
          /// <summary>Add an occurance of the symbol to the specified context.
          /// </summary>
          /// context <param name="context">
          ///   add the occurrence to this context's histogram</param>
          /// symbol <param name="symbol">
          ///   add an occurrence of this symbol to the histogram</param>
          void AddSymbol(UInt32 context, UInt32 symbol);

          /// <summary>Get the number of occurrences of the given symbol
          /// in the context.
          /// </summary>
          /// context <param name="context">
          ///   get the frequency from this context's histogram
          ///   </param>
          /// symbol <param name="symbol">
          /// get the frequency of this symbol the symbol
          ///   </param>
          /// <returns>the number of occurences of the specified symbol in the
          /// specified context
          /// </returns>
          UInt32 GetSymbolFrequency(UInt32 context, UInt32 symbol);

          /// <summary>Get the total number of occurrences for all of the
symbols
          /// that are less than the given symbol in the context.
          /// </summary>
          /// context <param name="context">
          ///   use this context's histogram
          ///   </param>
          /// symbol <param name="symbol">
          ///   use this symbol
          ///   </param>
          /// return <returns>
```

```
        ///   sum of all symbol freqs for symbols less than the
        /// given symbol in the given context
        /// </returns>
        UInt32 GetCumulativeSymbolFrequency(UInt32 context, UInt32 symbol);


        /// <summary>Get the total occurrences of all the symbols in this
        /// context.
        /// </summary>
        /// context<param name="context">use this context's histogram</param>
        /// <returns>total occurances of all symbols for the given context
        /// </returns>
        UInt32 GetTotalSymbolFrequency(UInt32 context);

        /// <summary>Find the symbol in a histogram that has
        /// the cumulative frequency specified.
        /// </summary>
        /// context<param name="context">
        ///   use this context's histogram
        /// </param>
        /// symbolFrequency<param name="symbolFrequency">
        ///   use this frequency
        ///   </param>
        /// return<returns>
        ///   the symbol that corresponds to the given cumulative frequency
        /// and context</returns>
        UInt32 GetSymbolFromFrequency(UInt32 context, UInt32
symbolFrequency);
    }
}
```

A.2.4 Data Block

```
using System;
namespace U3D
{
    /// <summary>
    /// The IDataBlock interface defines the properties associated with
    /// a block of data.  IDataBlock is used by the bitstream objects.
    /// </summary>
    public interface IDataBlock
    {
           /// <summary>
           /// DataSize is the size of the data in bytes.
           /// </summary>
           UInt32 DataSize
           {
                 get;
                 set;
           }

           /// <summary>
           /// Data is an array that stores the information.  The information
           /// in the DataBlock is in byte increments; so, not all of the array
           /// will contain valid data.  See the DataSize property for the
amount
           /// of valid data.
           /// </summary>
           UInt32[] Data
           {
                 get;
                 set;
           }

           /// <summary>
           /// MetaDataSize is the size of the MetaData in bytes.
           /// </summary>
           UInt32 MetaDataSize
           {
                 get;
                 set;
           }

           /// <summary>
           /// MetaData is an array that stores the information.  The
information
           /// in the DataBlock is in byte increments; so, not all of the array
           /// will contain valid data.  See the MetaDataSize property for the
           ///  amount of valid data.
           /// </summary>
           UInt32[] MetaData
```

```
            {
                  get;
                  set;
            }

            /// <summary>
            /// BlockType identifies the type of data stored in the data block so
            /// that it can be interpreted correctly.
            /// </summary>
            UInt32 BlockType
            {
                  get;
                  set;
            }

            /// <summary>
            /// Priority indicates where the block should be placed in relation
to
            ///  other blocks.  Blocks should be ordered in increasing priority.
            /// </summary>
            UInt32 Priority
            {
                  get;

                  set;
            }
      }
}
```

A.3 Classes

A.3.1 Bit Stream Write

```
using System;


namespace U3D
{

      /// <summary>BitStreamWrite.cs
      ///   BitStreamWrite is the implementation of IBitStreamWrite.
/// </summary>
/// <remarks>
/// <para>All uncompressed writes are converted to unsigned integers and
///  broken down into a sequence of U8 values that are written with the
/// private method WriteSymbol in the static context Context8.
/// </para>
/// <para> All compressed writes are for unsigned integers and are passed
/// through to the private method WriteSymbol with the associated context.
/// </para>
/// </remarks>
public class BitStreamWrite : IBitStreamWrite
{
      public BitStreamWrite()
      {
            this.contextManager = new ContextManager();
            this.high = 0x0000FFFF;
            this.data = new UInt32[DataSizeIncrement];
            this.compressed = false;
      }

      ~BitStreamWrite()
      {
      }

      #region IBitStreamWrite implementation

      public void WriteU8(Byte uValue)
      {
            UInt32 symbol = (UInt32) uValue;
            SwapBits8(ref symbol);
            bool escape = false;
            WriteSymbol(Constants.Context8, symbol, out escape);
      }

      public void WriteU16(UInt16 uValue)
      {
            WriteU8((Byte)(0x00FF & uValue));
            WriteU8((Byte)(0x00FF & (uValue >> 8)));
      }
```

```
public void WriteU32(UInt32 uValue)
{
      WriteU16((UInt16)(0x0000FFFF & uValue));
      WriteU16((UInt16)(0x0000FFFF & (uValue >> 16)));
}

public void WriteU64(UInt64 uValue)
{
      WriteU32((UInt32)(0x00000000FFFFFFFF & uValue));
      WriteU32((UInt32)(0x00000000FFFFFFFF & (uValue >> 32)));
}

public void WriteI32(Int32 iValue)
{
      WriteU32((UInt32)iValue);
}

public void WriteF32(Single fValue)
{
      UInt32 uValue =
            BitConverter.ToUInt32(BitConverter.GetBytes(fValue), 0);
      WriteU32((UInt32) uValue);
}

public void WriteCompressedU32(UInt32 context, UInt32 uValue)
{
      compressed = true;
      bool escape = false;
      if((context != 0) && (context < Constants.MaxRange))
      {
            WriteSymbol(context, uValue, out escape);
            if(escape == true)
            {
                  WriteU32(uValue);
                  this.contextManager.AddSymbol(context, symbol + 1U);
            }
      }
      else
```

```
    {
        WriteU32(uValue);

    }
}

public void WriteCompressedU16(UInt32 context, UInt16 uValue)
{
    compressed = true;
    bool escape = false;
    if((context != 0) && (context < Constants.MaxRange))
    {
        WriteSymbol(context, uValue, out escape);
        if(escape == true)
        {
            WriteU16(uValue);
            this.contextManager.AddSymbol(context, symbol + 1U);
        }
    }
    else
    {
        WriteU16(uValue);
    }
}


public void WriteCompressedU8(UInt32 context, Byte uValue)
{
    compressed = true;
    bool escape = false;
    if((context != 0) && (context < Constants.MaxRange))
    {
        WriteSymbol(context, uValue, out escape);
        if(escape == true)
        {
            WriteU8(uValue);
            this.contextManager.AddSymbol(context, symbol + 1U);
        }
    }
    else
```

```
      {
            WriteU8(uValue);
      }
}

public void GetDataBlock(out IDataBlock rDataBlock)
{
      if(compressed)               //Flush the arithmetic coder
      {
            this.WriteU32(0);
      }
      AlignToByte();
      UInt32 numBytes = ((UInt32)this.dataPosition << 2)
                             + ((UInt32)this.dataBitOffset >> 3);
      rDataBlock = new DataBlock();
      PutLocal();
      rDataBlock.DataSize = numBytes;
      UInt32[] tempData = rDataBlock.Data;
      Array.Copy(this.data, tempData, tempData.Length);
      rDataBlock.Data = tempData;
}

public void AlignToByte()
{
      // Check input(s)
      Int32 uBitCount = 0;
      GetBitCount(ref uBitCount);

      uBitCount = (8 - (uBitCount & 7)) & 7;
      this.dataBitOffset += uBitCount;

      if(this.dataBitOffset >= 32)
      {
            this.dataBitOffset -= 32;
            IncrementPosition();
      }
}

public void AlignTo4Byte()
```

```
            {
                  if(this.dataBitOffset > 0)
                  {
                        this.dataBitOffset = 0;
                        IncrementPosition();
                  }
            }

            #endregion IBitStreamWriter methods
            #region private helper methods

            /*
             * WriteSymbol
             * Write the given symbol to the datablock in the specified context.
             * rEscape returns as false if the symbol was written successfully.
             * rEscape will return true when writing in dynamically compressed
             * contexts when the symbol to write has not appeared yet in the
             * context's histogram.  In this case, the escape symbol, 0, is
             * written.
             */
            private  void  WriteSymbol(UInt32  context,  UInt32  symbol,  out  bool
rEscape)
            {
                  symbol++;
                  rEscape = false;
                  UInt32 totalCumFreq = 0;
                  UInt32 symbolCumFreq = 0;
                  UInt32 symbolFreq = 0;

                  totalCumFreq                                                  =
this.contextManager.GetTotalSymbolFrequency(context);
                  symbolCumFreq = this.contextManager
                              .GetCumulativeSymbolFrequency(context, symbol);
                  symbolFreq =
                        this.contextManager.GetSymbolFrequency(context, symbol);

                  if(0 == symbolFreq)
                  {     //the symbol has not occurred yet.
                        //Write out the escape symbol, 0.
                        symbol = 0;
```

```
                       symbolCumFreq = this.contextManager
                                        .GetCumulativeSymbolFrequency(context,
symbol);
                       symbolFreq =
                             this.contextManager.GetSymbolFrequency(context,
symbol);
                 }
                 if (0 == symbol)
                 {     //the symbol is the escape symbol.
                       rEscape = true;
                 }
                 UInt32 range = this.high + 1 - this.low;


                 this.high = this.low -1 + range
                                  * (symbolCumFreq + symbolFreq) / totalCumFreq;
                 this.low = this.low + range * symbolCumFreq / totalCumFreq;

                 this.contextManager.AddSymbol(context, symbol);

                 //write bits
                 UInt32 bit = this.low >> 15;

                 UInt32 highmask = this.high & Constants.HalfMask;
                 UInt32 lowmask = this.low & Constants.HalfMask;

                 while ((this.high & Constants.HalfMask)
                             == (this.low & Constants.HalfMask))
                 {
                       this.high &= ~Constants.HalfMask;
                       this.high += this.high + 1;
                       WriteBit(bit);

                       while(this.underflow > 0)
                       {
                             this.underflow--;
                             WriteBit((~bit) & 1);
                       }

                       this.low &= ~Constants.HalfMask;
                       this.low += this.low;
```

```
                        bit = this.low >> 15;
                  }

                  //check for underflow
                  // Underflow occurs when the values stored in this.high and
                  // this.low differ only in the most significant bit.
                  //  The  precision  of  the  variables  is  not  large  enough  to
predict
                  // the next symbol.
                  while ((0 == (this.high & Constants.QuarterMask))
                        && (Constants.QuarterMask
                              == (this.low & Constants.QuarterMask)))
                  {
                        this.high &= ~Constants.HalfMask;
                        this.high <<= 1;
                        this.low <<= 1;
                        this.high |= Constants.HalfMask;
                        this.high |= 1;
                        this.low &= ~Constants.HalfMask;
                        this.underflow++;
                  }
            }

            /*
             * SwapBits8
             * changes the ordering of an 8 bit value so that the first
             * 4 bits become the last 4 bits and the last 4 bits become
             * the first 4.  E.g. abcdefgh -> efghabcd
             */
            private void SwapBits8(ref UInt32 rValue)
            {
                  rValue = (Constants.Swap8[(rValue) & 0xf] << 4)
                              | (Constants.Swap8[(rValue) >> 4]);
            }

            /*
             * WriteBit
             * Write the given bit to the datablock.
             */
            private void WriteBit(UInt32 bit)
            {
```

NOTE
Shift operations on U32s are only valid for shifts of 0 to 31 bits.

```
                  UInt32 mask = 1;
                  bit &= mask;

                  this.dataLocal &= ~(mask << this.dataBitOffset);
                  this.dataLocal |= (bit << this.dataBitOffset);

                  this.dataBitOffset += 1;
                  if(this.dataBitOffset >= 32)
                  {
                        this.dataBitOffset -= 32;
                        IncrementPosition();
                  }
            }

            /*
             * IncrementPosition
             *  Updates  the  values  of  the  datablock  stored  in  dataLocal  and
dataLocalNext
             * to the next values in the datablock.
             */
            private void IncrementPosition()
            {
                  this.dataPosition++;
                  CheckPosition();
                  this.data[this.dataPosition-1] = this.dataLocal;
                  this.dataLocal = this.dataLocalNext;
                  this.dataLocalNext = this.data[this.dataPosition+1];
            }

            /*
             * GetLocal
             * store the initial 64 bits of the datablock in dataLocal and
             * dataLocalNext
             */
            private void GetLocal()
            {
                  CheckPosition();
            this.dataLocal = this.data[this.dataPosition];
            this.dataLocalNext = this.data[this.dataPosition+1];
      }

      /*
       * PutLocal
       * stores the local values of the data to the data array
       *
       */
      private void PutLocal()
      {
            this.data[this.dataPosition] = dataLocal;
            this.data[this.dataPosition+1] = dataLocalNext;
      }



      /*
       * CheckPosition
       * checks that the array allocated for writing is large
       * enough.  Reallocates if necessary.
       */
      private void CheckPosition()
      {
            if(this.dataPosition + 2 > this.data.Length)
            {
                  AllocateDataBuffer(this.dataPosition      +      2      +
DataSizeIncrement);
            }
      }

      /*
       * AllocateDataBuffer
       * Creates and new array for storing the data written.  Copies
       * values of the old array to the new arry.
       */
      private void AllocateDataBuffer(Int32 size)
      {
            // Store an old   buffer if it exists
            if(null    != this.data)
```

```
                  {
                        UInt32[] oldData = this.data;
                        this.data = new UInt32[size];

                        for(int i = 0; i < oldData.Length; i++)
                        {
                              this.data[i] = oldData[i];
                        }
                  }
                  else
                  {
                        this.data = new UInt32[size];
                  }
            }

            /*
             * GetBitCount
             * returns the number of bits written in rCount
             */
            void GetBitCount(ref Int32 rCount)
            {
                  rCount = (this.dataPosition << 5) + this.dataBitOffset;
            }

            #endregion private helper methods

            #region member variables

            private IContextManager contextManager; //the context manager handles
the updates
            //to the histograms for the compression contexts.

            private UInt32 high;                    //high and low are the upper and
lower
            private UInt32 low;                     //limits on the probability
            private UInt32 underflow;               //stores the number of bits of
underflow
                                              //caused by the limited range of high
and
                                              //low
            private bool compressed;     //this is true if a compressed value was
                                   //written.  when the datablock is retrieved,
                                   //a 32 bit 0 is written to reset the values of
                                   //high, low, and underflow.

            private UInt32[] data;       //the  data  section  of  the  datablock  to
write.
            private Int32 dataPosition;  //the position currently to write in the
                                         //datablock    specified    in    32    bit
increments.
            private UInt32 dataLocal;          //the   local   value   of   the   data
corresponding
                                         //to dataposition
            private UInt32 dataLocalNext;      //the   32   bits   in   data   after
dataLocal
            private Int32 dataBitOffset; //the  offset  into  dataLocal  that  the
next
                                         //write will occur

            #endregion member variables

            #region constants

            private const Int32 DataSizeIncrement = 0x000023F8;

            #endregion constants
      }
}
```

A.3.2 Bit Stream Read

```
using System;

namespace U3D
{
      /// <summary> BitStreamRead.cs
      /// BitStreamRead is the implementation of IBitStreamRead.</summary>
      ///
      /// <remarks>
      /// <para> All uncompressed reads are read in as a sequence of U8s
      /// with the private method ReadSymbol in context Context8 and then built
```

```
    /// up      to the appropriate size and cast to the appropriate type for
the
    /// read call. are converted to unsigned integers and broken down
    /// into a sequence of U8 values that are writen with the private method
    /// WriteSymbol in the static context Context8.
    /// </para>
    ///
    /// <para> All compressed reads are for unsigned integers and are passed
    /// through to the private method ReadSymbol with the associated context.
    /// </para>
    /// </remarks>

    public class BitStreamRead : IBitStreamRead
    {
        public BitStreamRead()
        {
            this.contextManager = new ContextManager();
            this.high = 0x0000FFFF;

        }

        ~BitStreamRead()
        {
        }
#region     IBitStreamRead interface implementation
        public void ReadU8(out Byte rValue)
        {
            UInt32 uValue = 0;

            ReadSymbol(Constants.Context8, out uValue);
            uValue--;
            SwapBits8(ref uValue);

            rValue = (Byte) uValue;
        }

        public void ReadU16(out UInt16 rValue)
        {
            Byte low = 0;
            Byte high = 0;
            ReadU8(out low);
            ReadU8(out high);

            rValue = (UInt16) (((UInt16) low) | (((UInt16) high) << 8));
        }

        public void ReadU32(out UInt32 rValue)
        {
            UInt16 low = 0;
            UInt16 high = 0;


            ReadU16(out low);
            ReadU16(out high);

            rValue = ((UInt32) low) | ((UInt32) (high << 16));
        }

        public void ReadU64(out UInt64 rValue)
        {
            UInt32 low = 0;
            UInt32 high = 0;

            ReadU32(out low);
            ReadU32(out high);

            rValue = ((UInt64) low) | (((UInt64) high) << 32);
        }

        public void ReadI32(out Int32 rValue)
        {
            UInt32 uValue = 0;

            ReadU32(out uValue);

            rValue = (Int32)(uValue);
        }

        public void ReadF32(out Single rValue)
        {
            UInt32 uValue = 0;
            ReadU32(out uValue);
            rValue   =   BitConverter.ToSingle(BitConverter.GetBytes(uValue),
0);
        }

        public void ReadCompressedU32(UInt32 context, out UInt32 rValue)
        {
            UInt32 symbol = 0;

            if    (context    !=    Constants.Context8    &&    context    <
Constants.MaxRange)
            {   //the context is a compressed context
                ReadSymbol(context, out symbol);
                if (symbol != 0)
                {   //the symbol is compressed
                    rValue = symbol - 1;
                }
                else
                {     //escape character, the symbol was not compressed
                    ReadU32(out rValue);
                    this.contextManager.AddSymbol(context, rValue + 1U);
                }
            }
            else
            {     //The context specified is uncompressed.
                ReadU32(out rValue);
            }
        }

        public void ReadCompressedU16(UInt32 context, out UInt16 rValue)
        {
            UInt32 symbol = 0;

            if (context != 0 && context < Constants.MaxRange)
            {   //the context is a compressed context
                ReadSymbol(context, out symbol);
                if (symbol != 0)
                {   //the symbol is compressed
```

```
                    rValue = (UInt16) (symbol - 1);
                }
                else
                {   //the symbol is uncompressed
                    ReadU16(out rValue);
                    this.contextManager.AddSymbol(context, rValue + 1U);
                }
            }
            else
            {   //the context specified is not compressed
                ReadU16(out rValue);
            }
        }

        public void ReadCompressedU8(UInt32 context, out Byte rValue)
        {
            UInt32 symbol = 0;

            if (context != 0 && context < Constants.MaxRange)
            {  //the context is a compressed context
                ReadSymbol(context, out symbol);
                if (symbol != 0)
                {  //the symbol is compressed
                    rValue = (Byte)(symbol - 1);
                }
                else
                {   //the symbol is not compressed
                    ReadU8(out rValue);
                    this.contextManager.AddSymbol(context,   rValue   +
(UInt32)1);
                }
            }
            else
            {   //the context specified is not compressed
                ReadU8(out rValue);
            }

        }

        public void SetDataBlock(IDataBlock dataBlock)
```

```
           {     //set the data to be read to data and get the first part of the
data
                 //into local variables
                 UInt32[] tempData = dataBlock.Data;
                 this.data = new UInt32[tempData.Length];
                 Array.Copy(tempData, this.data, tempData.Length);
                 this.dataPosition = 0;
                 this.dataBitOffset = 0;
                 GetLocal();
           }

#endregion IBitStreamRead implementation

#region private helper methods
           /*  internally  the  BitStreamRead  object  stores  64  bits  from  the
DataBlock's
            * data section in dataLocal and dataLocalNext.
            */

           /* SwapBits8
            * changes the ordering of an 8 bit value so that the first
            * 4 bits become the last 4 bits and the last 4 bits become
            * the first 4.  E.g. abcdefgh -> efghabcd
            */
           private void SwapBits8(ref UInt32 rValue)
           {
                 rValue = (Constants.Swap8[(rValue) & 0xf] << 4)
                            | (Constants.Swap8[(rValue) >> 4]);
           }

           /* ReadSymbol
            * Read a symbol from the datablock using the specified context.
            * The symbol 0 represents the escape value and signifies that the
            * next symbol read will be uncompressed.
            */
           private void ReadSymbol(UInt32 context, out UInt32 rSymbol)
           {
                 UInt32 uValue = 0;

                 // Fill in the code word
                 UInt32 position = 0;
                 GetBitCount(out position);
                 ReadBit(out this.code);
                 this.dataBitOffset += (Int32)this.underflow;

                 while (this.dataBitOffset >= 32)
                 {
                       this.dataBitOffset -= 32;
                       IncrementPosition();
                 }

                 UInt32 temp = 0;
                 Read15Bits(out temp);
                 this.code <<= 15;
                 this.code |= temp;
                 SeekToBit(position);

                 // Get total count to calculate probabilites
                 UInt32 totalCumFreq =
                       this.contextManager.GetTotalSymbolFrequency(context);

                 // Get the cumulative frequency of the current symbol
                 UInt32 range = this.high + 1 - this.low;

                 // The relationship:
                 // codeCumFreq <= (totalCumFreq * (this.code - this.low)) /
range

                 // is used to calculate the cumulative frequency of the current
                 // symbol.  The +1 and -1 in the line below are used to
counteract
                 // finite word length problems resulting from the division by
range.
                 UInt32 codeCumFreq =
                       ((totalCumFreq) * (1 + this.code - this.low) - 1) /
(range);
                 // Get the current symbol
                 uValue = this.contextManager
                                   .GetSymbolFromFrequency(context, codeCumFreq);
                 // Update state and context
                 UInt32 valueCumFreq =
                       this.contextManager
                             .GetCumulativeSymbolFrequency(context, uValue);
                 UInt32 valueFreq =
                       this.contextManager.GetSymbolFrequency(context, uValue);

                 UInt32 low = this.low;
                 UInt32 high = this.high;

                 high = low - 1 + range * (valueCumFreq + valueFreq) /
totalCumFreq;
                 low  = low + range * (valueCumFreq) / totalCumFreq;
                 this.contextManager.AddSymbol(context, uValue);



                 Int32 bitCount;
                 UInt32 maskedLow;
                 UInt32 maskedHigh;
                 // Count bits to read

                 // Fast count the first 4 bits
                 //compare most significant 4 bits of low and high
                 bitCount =
                       (Int32)ReadCount[((low >> 12) ^ (high >> 12)) &
0x0000000F];

                 low &= FastNotMask[bitCount];
                 high &= FastNotMask[bitCount];

                 high <<=  bitCount;
                 low <<= bitCount;

                 high |= (UInt32) ((1 << bitCount) -1);
                 // Regular count the rest

                 maskedLow = Constants.HalfMask & low;
                 maskedHigh = Constants.HalfMask & high;

                 while (((maskedLow | maskedHigh) == 0)
                       || ((maskedLow == Constants.HalfMask)
                             && maskedHigh == Constants.HalfMask))
```

```
{
      low = (Constants.NotHalfMask & low) << 1;
      high = ((Constants.NotHalfMask & high) << 1) | 1;
      maskedLow = Constants.HalfMask & low;
      maskedHigh = Constants.HalfMask & high;
      bitCount++;
}

UInt32 savedBitsLow = maskedLow;
UInt32 savedBitsHigh = maskedHigh;

if(bitCount > 0)
{
      bitCount += (Int32)this.underflow;
      this.underflow = 0;
}

// Count underflow bits
maskedLow = Constants.QuarterMask & low;
maskedHigh = Constants.QuarterMask & high;

UInt32 underflow = 0;

while ((maskedLow == 0x4000) && (maskedHigh == 0))
{
      low &= Constants.NotThreeQuarterMask;
      high &= Constants.NotThreeQuarterMask;

      low += low;
      high += high;

      high |= 1;
      maskedLow = Constants.QuarterMask & low;
      maskedHigh = Constants.QuarterMask & high;
      underflow++;
}

// Store the state
this.underflow += underflow;
```

```
            low |= savedBitsLow;
            high |= savedBitsHigh;
            this.low = low;
            this.high = high;

            // Update bit read position
            this.dataBitOffset += bitCount;

            while(this.dataBitOffset >= 32)
            {
                this.dataBitOffset -= 32;
                IncrementPosition();
            }
            // Set return value
            rSymbol = uValue;
        }

        /*
         * GetBitCount
         * returns the number of bits read in rCount
         */
        private void GetBitCount(out UInt32 rCount)
        {
            rCount = (UInt32)((this.dataPosition << 5) +
this.dataBitOffset);
        }

        /* ReadBit
         * Read the next bit in the datablock.  The value is returned in
         * rValue.
         */
        private void ReadBit(out UInt32 rValue)
        {
            UInt32 uValue = 0;

            uValue = this.dataLocal >> this.dataBitOffset;

            uValue &= 1;
            this.dataBitOffset ++;
            if(this.dataBitOffset >= 32)
        {
            this.dataBitOffset -= 32;
            IncrementPosition();
        }

        rValue = uValue;
    }

    /* Read15Bits
     * Read the next 15 bits from the datablock.  the value is returned
     * in rValue.
     */
    private void Read15Bits(out UInt32 rValue)
    {
        UInt32 uValue = this.dataLocal >> this.dataBitOffset;

        if(this.dataBitOffset > 17)
        {
            uValue     |=     (this.dataLocalNext     <<     (32     -
this.dataBitOffset));
        }

        uValue += uValue;

        uValue = (Constants.Swap8[(uValue >> 12) & 0xf])
            | ((Constants.Swap8[(uValue >> 8) & 0xf ]) << 4 )
            | ((Constants.Swap8[(uValue >> 4) & 0xf]) << 8 )
            | ((Constants.Swap8[uValue & 0xf]) << 12 );



        rValue = uValue;
        this.dataBitOffset += 15;
        if(this.dataBitOffset >= 32)
        {
            this.dataBitOffset -= 32;
            IncrementPosition();
        }
    }

    /*
```

```
             * IncrementPosition
             *  Updates  the  values  of  the  datablock  stored  in  dataLocal  and
dataLocalNext
             * to the next values in the datablock.
             */
            private void IncrementPosition()
            {
                  this.dataPosition++;

                  this.dataLocal = this.data[dataPosition];
                  if(this.data.Length > this.dataPosition+1)
                  {
                        this.dataLocalNext = this.data[this.dataPosition+1];
                  }
                  else
                  {
                        this.dataLocalNext = 0;
                  }
            }

            /* SeekToBit
             * Sets the dataLocal, dataLocalNext and bitOffSet values so that
             * the next read will occur at position in the datablock.
             */
            private void SeekToBit(UInt32 position)
            {
                  this.dataPosition = position >> 5;
                  this.dataBitOffset = (Int32)(position & 0x0000001F);
                  GetLocal();
            }

            /*
             * GetLocal
             * store the initial 64 bits of the datablock in dataLocal and
             * dataLocalNext
             */
            private void GetLocal()
            {
                  this.dataLocal = this.data[this.dataPosition];
                  if(this.data.Length > this.dataPosition + 1)
```

```
                 {
                       this.dataLocalNext = this.data[this.dataPosition+1];
                 }
           }
           #endregion private helper methods

           #region member variables

           private IContextManager contextManager; //the context manager handles
                                              //the updates to the histograms
                                              //for the compression contexts.

           private UInt32 high;                    //high and low are the upper
and
           private UInt32 low;                     //lower limits on the
                                              //probability
           private UInt32 underflow;               //stores the number of bits
of
                                              //underflow caused by the
                                              //limited range of high and low
           private UInt32 code;                    //the value as represented in
                                              //the datablock

           private UInt32[] data;             //the data section of the
                                              //datablock to read from.
           private UInt32   dataPosition;          //the position currently read
in
                                              //the datablock specified in 32
                                              //bit increments.

           private UInt32 dataLocal;               //the local value of the data
                                              //corresponding to dataposition.
           private UInt32 dataLocalNext;           //the 32 bits in data after
                                              //dataLocal
           private Int32 dataBitOffset;       //the offset into dataLocal that
                                              // the next read will occur

           private static readonly UInt32[] FastNotMask
                 = {0x0000FFFF, 0x00007FFF, 0x00003FFF, 0x00001FFF, 0x00000FFF};
           private static readonly UInt32[] ReadCount
                 = {4, 3, 2, 2, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0};
```

```
        #endregion member variables
    }

}
```

A.3.3 Context Manager

```
using System;


namespace U3D
{
    public class ContextManager : IContextManager
    {
        public ContextManager()
        {
            this.symbolCount = new UInt16[Constants.StaticFull][];
            this.cumulativeCount = new UInt16[Constants.StaticFull][];
        }
        #region IContextManager Members

        public void AddSymbol(UInt32 context, UInt32 symbol)
        {
            if    (context    <    Constants.StaticFull    &&    context    !=
Constants.Context8
                && symbol < MaximumSymbolInHistogram)
            {   //check if dynamic.  nothing to do if static or if the
                //symbol is larger than the maximum symbol allowed in the
                //histogram

                UInt16[] cumulativeCount = this.cumulativeCount[context];
                UInt16[] symbolCount = this.symbolCount[context];

                if (cumulativeCount == null || cumulativeCount.Length <=
symbol)
                {   //allocate new arrays if they do not exist yet or if
they
                    //are too small.
                    cumulativeCount    =    new    UInt16[symbol    +
ArraySizeIncr];
                    symbolCount = new UInt16[symbol + ArraySizeIncr];
```

```
                            if(cumulativeCount != null && symbolCount != null)
                            {//check that the arrays were allocated successfully
                                  if (this.cumulativeCount[context] == null)
                                  {//if  this  is  a  new  context  set  up  the
histogram
                                        this.cumulativeCount[context]          =
cumulativeCount;
                                        this.cumulativeCount[context][0] = 1;
                                        this.symbolCount[context] = symbolCount;
                                        this.symbolCount[context][0] = 1;
                                  }
                                  else
                                  {//if  this  is  an  old  context,  copy  over  the
values in
                                   //the histogram to the new arrays
                                        this.cumulativeCount[context]
                                              .CopyTo(cumulativeCount, 0);

      this.symbolCount[context].CopyTo(symbolCount, 0);
                                  }
                            }

                            this.cumulativeCount[context] = cumulativeCount;
                            this.symbolCount[context] = symbolCount;
                      }

                      if(cumulativeCount[0] >= Elephant)
                      {//if total number of occurances is larger than Elephant,
                       //scale down the values to avoid overflow
                            int len = cumulativeCount.Length;
                            UInt16 tempAccum = 0;
                            for(int i = len - 1; i >= 0; i--)
                            {
                                  symbolCount[i] >>= 1;
                                  tempAccum += symbolCount[i];
                                  cumulativeCount[i] = tempAccum;
                            }
                            //preserve  the  initial  escape  value  of  1  for  the
symbol
                            //count and  cumulative count
```

```
                    symbolCount[0]++;
                    cumulativeCount[0]++;
                }

                symbolCount[symbol]++;
                for(int i = 0; i <= symbol; i++)
                {
                    cumulativeCount[i]++;
                }
            }
        }

        public UInt32 GetSymbolFrequency(UInt32 context, UInt32 symbol)
        {
            //the static case is 1.
            UInt32 rValue = 1;
            if (context < Constants.StaticFull && context !=
Constants.Context8)
            {
                //the default for the dynamic case is 0
                rValue = 0;
                if ((this.symbolCount[context] != null)
                    && (symbol < this.symbolCount[context].Length))
                {
                    rValue = (UInt32) this.symbolCount[context][symbol];
                }
                else if (symbol == 0)
                {   //if the histogram hasn't been created yet, the
symbol 0 is

                    //the escape value and should return 1
                    rValue = 1;
                }
            }

            return rValue;
        }

        public UInt32 GetCumulativeSymbolFrequency(UInt32 context, UInt32
symbol)
        {
            //the static case is just the value of the symbol.
```

```
                 UInt32 rValue = symbol - 1;
                 if (context < Constants.StaticFull && context !=
Constants.Context8)
                 {
                       rValue = 0;
                       if (this.cumulativeCount[context] != null)
                       {
                             if(symbol < this.cumulativeCount[context].Length)
                             {
                                   rValue                                      =
(UInt32)(this.cumulativeCount[context][0]
                                          -
this.cumulativeCount[context][symbol]);
                             }
                             else
                                   rValue                                      =
(UInt32)(this.cumulativeCount[context][0]);
                       }
                 }
                 return rValue;
           }

           public UInt32 GetTotalSymbolFrequency(UInt32 context)
           {
                 if (context < Constants.StaticFull && context !=
Constants.Context8)
                 {
                       UInt32 rValue = 1;
                       if(this.cumulativeCount[context] != null)
                             rValue = this.cumulativeCount[context][0];
                       return rValue;
                 }
                 else
                 {
                       if (context == Constants.Context8)
                             return 256;
                       else
                             return context - Constants.StaticFull;
                 }
           }
```

```
        public UInt32 GetSymbolFromFrequency(UInt32 context, UInt32
symbolFrequency)
        {
            UInt32 rValue = 0;
            if (context < Constants.StaticFull && context !=
Constants.Context8)
            {
                rValue = 0;
                if (this.cumulativeCount[context] != null
                    && symbolFrequency != 0
                    && this.cumulativeCount[context][0] >=
symbolFrequency)
                {

                    UInt32 i = 0;

                    for(i = 0; i < this.cumulativeCount[context].Length;
i++)
                    {
                        if (this.GetCumulativeSymbolFrequency(context,
i)
                                <= symbolFrequency)
                            rValue = i;
                        else
                            break;
                    }
                }
            }
            else
            {
                rValue = symbolFrequency + 1;
            }
            return rValue;
        }

        #endregion

        #region Member variables

        private UInt16[][] symbolCount;       //an array of arrays that
store the
                                //number of occurrences of each
```

```
                                              // symbol for each dynamic context.
            private UInt16[][] cumulativeCount;     //an   array   of   arrays   that
store the
                                              //cumulative frequency of each
                                              //symbol   in   each   context.    the
value
                                              //is the number of occurences of a
                                              //symbol and every symbol with a
                                              //larger value.

            #endregion Member variables

            #region constants

            // The Elephant is a value that determines the number of
            // symbol occurences that are stored in each dynamic histogram.
            // Limiting the number of occurences avoids overflow of the U16 array
            // elements and allows the histogram to adapt to changing symbol
            // distributions in files.
            private const UInt32 Elephant = 0x00001fff;
            //the maximum value that is stored in a histogram
            private const UInt32 MaximumSymbolInHistogram = 0x0000FFFF;
            //the ammount to increase the size of an array when reallocating
            //an array.
            private const UInt32 ArraySizeIncr = 32;
            #endregion constants

      }
}
```

A.3.4 Data Block

```
using System;
namespace U3D
{
      public class DataBlock : IDataBlock
      {
            public DataBlock()
            {
                  this.dataSize = 0;
                  this.data = null;
                  this.metaDataSize = 0;
```

```
    this.metaData = null;
    this.blockType = 0;
    this.priority = 0;
}

public UInt32 DataSize
{
    get
    {
        return this.dataSize;
    }
    set
    {
        this.dataSize = value;
        //allocate data buffer for block.
        //the data is generally aligned to byte values
        //but array is 4 bytes values . . .
        if ((this.dataSize & 0x3) == 0)
            this.data = new UInt32[value >> 2];
        else
            this.data = new UInt32[(value >> 2) + 1];
    }
}

public UInt32[] Data
{
    get
    {
        return this.data;
    }
    set
    {
        this.data = value;
    }
}

public UInt32 MetaDataSize
{
    get
```

```
            {
                return this.metaDataSize;
            }
            set
            {
                this.metaDataSize = value;
                //allocate data buffer for block.
                //the data is generally aligned to byte values
                //but array is 4 bytes values . . .
                if ((this.metaDataSize & 0x3) == 0)
                    this.metaData = new UInt32[value >> 2];
                else
                    this.metaData = new UInt32[(value >> 2) + 1];
            }
        }

        public UInt32[] MetaData
        {
            get
            {
                return this.metaData;
            }
            set
            {
                if(value.Length == this.metaData.Length)
                {
                    Array.Copy(value, this.metaData, value.Length);
                }
            }
        }

        public UInt32 BlockType
        {
            get
            {
                return this.blockType;
            }
            set
            {
                this.blockType = value;
            }
        }

        public UInt32 Priority
        {
            get
            {
                return this.priority;
            }
            set
            {
                this.priority = value;
            }
        }

        private UInt32[] data;
        private UInt32 dataSize;
        private UInt32[] metaData;
        private UInt32 metaDataSize;
        private UInt32 priority;
        private UInt32 blockType;
    }
}
```

A.3.5 Constants

```
using System;


namespace U3D
{

    /// <summary>Constants is a class that holds constants that are needed by
more than
    /// one of the objects in the U3D namespace.</summary>
    public class Constants
    {

        #region arithmetic compression constants
```

```
            //context ranges
            /// <summary>
            /// the context for uncompressed U8
            /// </summary>
            public const UInt32 Context8 = 0;

            /// <summary>
            /// contexts >= StaticFull are static contexts.
            /// </summary>
            public const UInt32 StaticFull = 0x00000400;

            ///<summary>
            ///The largest allowable static context.  values written to contexts
> MaxRange are
            ///written as uncompressed.
            ///</summary>
            public const UInt32 MaxRange = StaticFull + 0x00003FFF;

            /// <summary>
            /// a defualt buffer size for U3D
            /// </summary>

            public const UInt32 SizeBuff = 1024;
            /// <summary>
            /// the initial size allocated for buffers
            /// </summary>
            public const UInt32 DataSizeInitial = 0x00000010;

            //Bit masks for reading and writing symbols.
            /// <summary>
            /// masks all but the most significan bit
            /// </summary>
            public const UInt32 HalfMask = 0x00008000;
            /// <summary>
            /// masks the most significant bit
            /// </summary>
            public const UInt32 NotHalfMask = 0x00007FFF;
            /// <summary>
            /// masks all but the 2nd most significan bit
            /// </summary>
            public const UInt32 QuarterMask = 0x00004000;

            /// <summary>
            /// masks the 2 most significant bits
            /// </summary>
            public const UInt32 NotThreeQuarterMask = 0x00003FFF;

            /// <summary>
            /// used to swap 8 bits in place
            /// </summary>
            public static readonly UInt32[] Swap8
                                       = {0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3,
11, 7, 15};
            #endregion

      }

}
```