INTERNATIONAL STANDARD

ISO/IEC 23090-23

First edition
2023-10

Information technology — Coded representation of immersive media —

Part 23: Conformance and reference software for MPEG immersive video

Technologies de l'information — Représentation codée de média immersifs —

Partie 23: Conformité et logiciels de référence pour la vidéo immersive MPEG

Reference number
ISO/IEC 23090-23:2023(E)

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC had not received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents and https://patents.iec.ch. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

A list of all parts in the ISO/IEC 23090 series can be found on the ISO and IEC websites.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

Introduction

ISO/IEC 23090-12 was developed to support compression of immersive video content, in which a real or virtual 3D scene is captured by multiple real or virtual cameras. The use of this document enables storage and distribution of immersive video content over existing and future networks, for playback with 6 degrees of freedom of view position and orientation.

Information technology — Coded representation of immersive media —

Part 23: Conformance and reference software for MPEG immersive video

1 Scope

This document specifies a set of tests and procedures designed to indicate whether encoders or decoders meet the requirements specified in ISO/IEC 23090-12.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 23090-12:2023, *Information technology — Coded representation of immersive media — Part 12: MPEG immersive video*

3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 23090-12 and the following apply.

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at https://www.electropedia.org/

3.1
alternative decoder
decoder (3.4) other than the *reference software decoder* (3.7)

3.2
bitstream
sequence of bits that conforms to the syntax requirements specified by ISO/IEC 23090-12 or sequence of bits to be tested for conformance to those syntax requirements

3.3
bitstream zip-file
archive file containing a bitstream plus files with information on that bitstream for the purpose of decoder conformance testing

3.4
decoder
embodiment of the decoding process specified by ISO/IEC 23090-12 or process to be tested for conformance to that decoding process specification

Note 1 to entry: The decoding process does not include post-decoding, reconstruction and display process, which are outside of the scope of this document.

3.5
decoder output log
log file that is output by a *decoder* (3.4) in response to a *bitstream* (3.2) for use in bitstream analysis or decoder conformance testing

3.6
encoder
embodiment of a process, not specified in this document, that produces a *bitstream* (3.2)

3.7
reference software decoder
particular *decoder* (3.4) provided as a software package for use as an example available for study, as a potential starting basis for the development of other decoders, as a way of testing bitstreams for conformance, and as a reference for comparison with the behaviour of other decoders

3.8
reference software encoder
particular *encoder* (3.6) provided as a software package for use as an example available for study, as a potential starting basis for the development of other encoders, and as a reference for comparison with the behaviour of other encoders

4 Abbreviated terms

For the purposes of this document, the abbreviated terms in ISO/IEC 23090-12:2023, Clause 4 apply.

5 Conventions

For the purposes of this document, the conventions specified in ISO/IEC 23090-12:2023, Clause 5 apply.

6 Conformance testing

6.1 General

The following clauses specify normative tests for verifying the conformance of bitstreams as well as decoders. Those normative tests make use of test data (bitstream test suites) provided as an electronic annex to this document and the reference software decoder.

6.2 Bitstream conformance

The bitstream conformance of ISO/IEC 23090-12 is specified by ISO/IEC 23090-12:2023, Clause E.4.

6.3 Decoder conformance

The decoder conformance of ISO/IEC 23090-12 is specified by ISO/IEC 23090-12:2023, Clause E.5.

6.4 Reference bitstreams

The bitstreams used for the decoder conformance testing specified in this document shall be those listed in Table 1. They are available at:

https://standards.iso.org/iso-iec/23090/-23/ed-1/en/

Table 1 — List of reference bitstreams

ID	Codec group IDC	Toolset IDC	Features tested
CB01	HEVC Main10	MIV Extended	Packed video data, entity coding
CB02	VVC Main10	MIV Extended	Full occupancy, occupancy video data, different atlas frame sizes
CB03	VVC Main10	MIV Main	Explicit view IDs
CB04	VVC Main10	MIV Main	Grouping, explicit view IDs, multiple CVSs
CB05.1	VVC Main10	MIV Extended	None of the atlases contain constant depth patches
CB05.2	VVC Main10	MIV Extended	One atlas has constant depth patches, one does not
CB05.3	VVC Main10	MIV Extended	All atlases contain constant depth patches
CB06	VVC Main10	MIV Main	Ancillary atlas flag
CB07.1	VVC Main10	MIV Main	Pruning graph (cluster graph)
CB07.2	VVC Main10	MIV Main	Pruning graph (connected graph)
CB07.3	VVC Main10	MIV Main	No pruning graph
CB08	HEVC Main10	MIV Extended	Restricted geometry
CB09	HEVC Main10	MIV Main	Geometry scaling (different factors)
CB10	HEVC Main10	MIV Main	8-bit video sub-bitstreams
CB11	VVC Main10	MIV Main	Inpaint patches
CB12	VVC Main10	MIV Main	Coordinate system parameters
CB14	VVC Main10	MIV Main	360-degree equirectangular projection (clip Classroom-Video)
CB15	VVC Main10	MIV Main	Perspective projection matrix layout (clip Painter)
CB16	VVC Main10	MIV Main	180-degree equirectangular projection (clip Museum)
CB17	VVC Main10	MIV Main	Perspective projection vector layout (clip Frog)
CB18	VVC Main10	MIV Geometry absent	Only texture video (clip Cadillac)
CB19	VVC Main10	MIV Main	Non-IRAP frames: NAL_CAF_TRIAL and NAL_TRAIL_N
CB20	VVC Main10	MIV Extended	Multiple tiles

6.5 Procedure to test bitstreams

A bitstream that claims conformance with ISO/IEC 23090-12 shall pass the following test:

The bitstream shall be decoded by processing it with the reference software decoder. When processed by the reference software decoder, the bitstream shall not cause any error or non-conformance messages to be reported by the reference software decoder. This test should not be applied to bitstreams that are known to contain errors introduced by transmission, as such errors are highly likely to result in bitstreams that lack conformance to ISO/IEC 23090-12.

Additional tests can be necessary to more thoroughly check that the bitstream properly meets all the requirements specified in ISO/IEC 23090-12, including the hypothetical reference decoder (HRD) conformance, supplemental enhancement information messages (SEI) and volumetric usability information (VUI). These complementary tests may be performed using other bitstream verifiers that perform more complete tests than those implemented by the reference software decoder.

To check the correctness of a bitstream, it is necessary to parse the entire bitstream and to extract all the syntax elements and other values derived from those syntactic elements and used by the decoding process specified in ISO/IEC 23090-12.

A verifier will not necessarily perform all stages of the decoding process specified in ISO/IEC 23090-12 to verify bitstream correctness. Many tests can be performed on syntax elements in a state prior to their use in some processing stages.

6.6 Procedure to test decoders

6.6.1 Conformance bitstreams

A bitstream has values of ptl_profile_codec_group_idc, ptl_profile_toolset_idc and ptl_level_idc corresponding to a set of specified constraints on a bitstream for which a decoder conforming to a specified profile and level is required to properly perform the decoding process.

6.6.2 Contents of the bitstream zip-files

The conformance bitstreams are available as an electronic attachment to this document. The following information is included in a single zip-file for each bitstream:

— the test bitstream in V3C sample stream format, with file extension `.bit`,

— a decoder output log, as specified in Clause 8, with file extension `.dec`.

— a short textual description of the bitstream, with file extension `.txt`, including:

  — contact information,

  — profile-tier-level information,

  — a description of the encoder, including version information, that was used to create the bitstream,

  — the version of the reference software decoder that was used to create the decoder output log,

  — a description of the alternative decoders (if any), including version information, that were used to verify the decoder output log,

  — a list of features of ISO/IEC 23090-12 that are specifically exercised by this bitstream.

6.6.3 Requirements on decoder output and timing

The output of the decoding process is specified in ISO/IEC 23090-12.

For output timing conformance, a conforming decoder shall also output the decoded samples at the rates and times specified in ISO/IEC 23090-12.

Post-decoding, reconstruction and display processes are outside of the scope of this document.

7 Reference software

7.1 Purpose of the reference software

The purpose of the reference software includes:

— serving as an example available for study, e.g. to illustrate how the standard can be implemented,

— demonstrating the capabilities and behaviour that can be expected from implementations of the standard to clarify how to interpret what is specified in ISO/IEC 23090-12,

— providing a potential basis for the development of implementations of ISO/IEC 23090-12,

— providing a reference for comparison with the behaviour of other decoders,

— testing of bitstreams for conformance to ISO/IEC 23090-12.

The use of the reference software is not a requirement for the implementation of ISO/IEC 23090-12. An implementer is permitted to use only the text of ISO/IEC 23090-12 for creating an implementation of a conforming encoder or decoder.

7.2 Software location

The reference software is available at the following location:

https://standards.iso.org/iso-iec/23090/-23/ed-1/en/

This document corresponds to version 15.1.1 of the reference software.

7.3 Software license

The software license is included in the electronic attachment that provides the reference software.

7.4 Software installation

The software manual is included in the electronic attachment that provides the reference software.

7.5 Software architecture

7.5.1 Reference software encoder

The reference software encoder consists of the `TmivEncoder` and `TmivMultiplexer` executables. To encode a bitstream, the following steps are followed:

a) Invoke the `TmivEncoder` to output:

   1) a partial bitstream that does not include the video sub bitstreams,

   2) an uncompressed video data file per video sub bitstream.

b) For each uncompressed video data file, invoke a video encoder to encode the video data file, outputting a video sub bitstream.

c) Invoke the `TmivMultiplexer` to output a bitstream.

7.5.2 Reference software decoder

The reference software decoder consists of a single executable named `TmivDecoder`.

The reference software decoder is configurable to output a decoder output log according to the format that is specified in Clause 8.

8 Decoder output logging process

8.1 General decoder output logging process

The decoder output log is a file in UTF-8 format specified in ISO/IEC 10646, without byte order marking and with line endings formed by a single U-000A [LF] code point. The file consists of a single-space-separated table with one row per line. An example of a decoder output log is:

```
   0 0 1024 1024 G 21623b35 A 0 6c9de1e8 b4d473f8 e797e251 432058a9 73b0f2c9 00000000
26c9cb0a
   0 1 1024 1024 G 14fcdcf2 A 0 36ffa830 0ba8e5c2 fd42de31 432058a9 73b0f2c9 00000000
26c9cb0a
   1 0 1024 1024 G df1e301d A 0 e3f15be5 b4d473f8 e797e251 432058a9 73b0f2c9 00000000
26c9cb0a
   1 1 1024 1024 G f1a1a0c4 A 0 a89a7200 0ba8e5c2 fd42de31 432058a9 73b0f2c9 00000000
26c9cb0a
   2 0 1024 1024 G 8f3a24d3 A 0 3399c717 9a9b5536 cd0a09a7 cab9c5db 73b0f2c9 00000000
26c9cb0a
   2 1 1024 1024 G 29641a88 A 0 d0e419f0 0ba8e5c2 7103d7b1 cab9c5db 73b0f2c9 00000000
26c9cb0a
```

There is one row per volumetric frame and atlas. The rows are ordered by frame order count (presentation order) and atlas ID.

The columns in order from left to right are:

— frame order count;

— atlas ID;

— atlas frame width;

— atlas frame height;

— if decoded occupancy video data is present:

  — literal “O”;

  — hash over *DecOccFrames*, as specified in 8.3.1;

— if decoded geometry video data is present:

  — literal “G”;

  — hash over *DecGeoFrames*, as specified in 8.3.2

— if decoded attribute video data is present:

  — literal “A”;

  — for each attribute in order of increasing index:

    — attribute index;

    — hash over *DecAttrFrames*, as specified in 8.3.3;

— if decoded packed video data is present:

  — literal “P”

  — hash over *DecPckFrames*, as specified in 8.3.4;

— block to patch map hash, specified in 8.4;

— patch params list hash, specified in 8.5;

— view params list hash, specified in 8.6;

— atlas sequence parameter set MIV extension hash, specified in 8.7;

— atlas frame parameter set MIV extension hash, specified in 8.8;

— common atlas sequence parameter set MIV extension hash, specified in 8.9.

All hashes are formatted as an eight-digit zero-padded lower-case hexadecimal number.

8.2 General hashing process

8.2.1 General

This process and subprocesses calculate a CRC-32 hash, specified in ISO/IEC/IEEE 8802-3, as follows:

a) Pre-compute the CRC32 table (8.2.2).

b) Initialize the hash state (8.2.3).

c) Consume multiple unsigned integers (8.2.4), signed integers (8.2.5), or floating-point values (8.2.6) to hash.

d) Calculate the hash value (8.2.7).

8.2.2 Hash table pre-computation process

This process calculates the table *Crc32Table* of length 256 as follows:

```
for (i = 0; i < 0x100; i++) {
   n = i
   for (j = 0; j < 8; j++) {
      n = (n & 1) == 0 ? n >> 1 : (n >> 1) ^ 0xEDB88320
   }
   Crc32Table[ i ] = n
}
```

8.2.3 Hash state initialization process

This process is defined by the function *InitHashState*( ) as follows:

```
InitHashState( ) {
   return 0xFFFFFFFF
}
```

8.2.4 Hash state update process for unsigned integer values

The inputs to this process are:

— the variable *state* in range 0 .. 2^{32} - 1, inclusive,

— the variable *value* in range 0 .. 2^{32} - 1, inclusive.

The output of this process is a new hash state in range 0 .. 2^{32} - 1.

This process is defined by the function *HashConsumeU32*(*state*, *value*) as follows:

```
HashConsumeU32( state, value ) {
   s = state
   s = (s >> 8) ^ crc32Table[ (s ^ ( value >> 24 ) ) & 0xFF ]
   s = (s >> 8) ^ crc32Table[ (s ^ ( value >> 16 ) ) & 0xFF ]
   s = (s >> 8) ^ crc32Table[ (s ^ ( value >> 8 ) ) & 0xFF ]
   s = (s >> 8) ^ crc32Table[ (s ^ value ) & 0xFF ]
   return s
}
```

8.2.5 Hash state update process for signed integer values

The inputs to this process are:

— the variable *state* in range 0 .. 2^{32} - 1, inclusive,

— the signed integer variable *value*.

The output of this process is a new hash state in range 0 .. 2^{32} - 1.

This process defines the function HashConsumeI32(state, value) as follows:

```
HashConsumeI32( state, value ) {
   return 0 <= value
      ? HasConsumeU32( state, value )
      : HasConsumeU32( state, value + 2^32 )
}
```

8.2.6 Hash state update process for floating-point values

The inputs to this process are:

— the variable *state* in range 0 .. 2^{32} - 1, inclusive,

— the floating-point variable *valueFL32*.

The output of this process is a new hash state in range 0 .. 2^{32} - 1.

This process defines the function HashConsumeFL32(state, valueFL32) as follows:

a) The floating-point value *valueFL32* is converted to a bit string according to the fl(32) descriptor.

b) The bit string is parsed using the u(32) descriptor, resulting in a value *valueU32*.

c) The output of this process is the result of *HashConsumeU32*(*state*, *valueU32*).

8.2.7 Hash value computation process

The input to this process is the variable *state* in range 0 .. 2^{32} - 1, inclusive.

The output of this process is the hash value in range 0 .. 2^{32} - 1.

This process is defined by the function *hashValue*(*state*) as follows:

```
ComputeHashValue( state ) {
   return ~state
}
```

8.3 Video data hashing process

8.3.1 Occupancy video data hashing process

The inputs to this process are:

— a decoded MIV access unit with decoded video frame index *frameIdx*,

— an atlas ID *atlasId*.

The output of this process is the value *hashValue* in range 0 .. 2^{32} - 1.

The process is defined as follows:

```
s = InitHashState(

for( i = 0; i < DecOccHeight[ frameIdx ]; i++ )
   for( j = 0; j < DecOccWidth[ frameIdx ]; j++ )
      s = HashConsumeU32( s,
         DecOccFrames[ frameIdx ][ 0 ][ i ][ j ] )

hashValue = ComputeHashValue( s )
```

8.3.2 Geometry video data hashing process

The inputs to this process are:

— a decoded MIV access unit with decoded video frame index *frameIdx*,

— an atlas ID *atlasId*.

The output of this process is the value *hashValue* in range 0 .. 2^{32} - 1.

The process is defined as follows:

```
s = InitHashState( )

for( i = 0; i < DecGeoHeight[ 0 ][ frameIdx ]; j++ )
   for( j = 0; j < DecGeoWidth[ 0 ][ frameIdx ]; j++ )
      s = HashConsumeU32( s,
         DecGeoFrames[ 0 ][ frameIdx ][ 0 ][ i ][ j ] )

hashValue = ComputeHashValue( s )
```

8.3.3 Attribute video data hashing process

The inputs to this process are:

— a decoded MIV access unit with decoded video frame index *frameIdx*,

— an atlas ID *atlasId*,

— an attribute index *attrIdx*.

The output of this process is the value *hashValue* in range 0 .. 2^{32} - 1.

The process is defined as follows:

```
s = InitHashState( )

for( i = 0; i < DecAttrHeight[ attrIdx ][ 0 ][ 0 ][ frameIdx ]; i++ )
   for( j = 0; j < DecAttrWidth[ attrIdx ][ 0 ][ 0 ][ frameIdx ]; j++ )
      for( c = 0; c < DecAttrNumComp[ attrIdx ][ 0 ][ 0 ][ frameIdx ]; c++ )
         s = HashConsumeU32( s,
            DecAttrFrames[ attrIdx ][ 0 ][ 0 ][ frameIdx ][ c ][ i ][ j ] )

hashValue = ComputeHashValue( s )
```

8.3.4 Packed video data hashing process

The inputs to this process are:

— a decoded MIV access unit with decoded video frame index *frameIdx*,

— an atlas ID *atlasId*.

The output of this process is the value *hashValue* in range 0 .. 2^{32} - 1.

The process is defined as follows:

```
s = InitHashState( )

for( i = 0; i < DecPckHeight[ frameIdx ]; i++ )
   for( j = 0; j < DecPckWidth[ frameIdx ]; j++ )
      for( c = 0; c < DecPckNumComp[ frameIdx ]; c++ )
         s = HashConsumeU32( s,
            DecPckFrames[ frameIdx ][ c ][ i ][ j ] )

hashValue = ComputeHashValue( s )
```

8.4 Block to patch map hashing process

The inputs to this process are a decoded block to patch map and an atlas ID *atlasId*.

The output of this process is the value *hashValue* in range 0 .. 2^{32} - 1.

The process is defined as follows:

```
s = InitHashState( )

for( i = 0; i < AtlasBlockToPatchMapHeight; i++ ) {
   for( j =0; j < AtlasBlockToPatchMapWidth; j++ ) {
      p = AtlasBlockToPatchMap[ i ][ j ]
      s = HashConsumeU32( s, p == -1 ? 0xFFFFFFFF : p )
```

```
    }
}
hashValue = ComputeHashValue( s )
```

8.5 Patch params list hashing process

The inputs to this process are the variable *AtlasTotalNumPatches* and the arrays of which the name starts with *AtlasPatch*.

The output of this process is the value *hashValue* in range 0 .. 2^{32} - 1.

This process is defined as follows:

```
s = InitHashState( )

for( p = 0; p < AtlasTotalNumPatches; p++ ) {
    s = HashConsumeU32( s, AtlasPatch3dOffsetU[ p ] )
    s = HashConsumeU32( s, AtlasPatch3dOffsetV[ p ] )
    s = HashConsumeU32( s, AtlasPatch3dOffsetD[ p ] )
    s = HashConsumeU32( s, AtlasPatch3dRangeD[ p ] )
    s = HashConsumeU32( s, AtlasPatchProjectionId[ p ] )
    s = HashConsumeU32( s, AtlasPatchOrientationIndex[ p ] )
    s = HashConsumeU32( s, AtlasPatchLoDScaleX[ p ] )
    s = HashConsumeU32( s, AtlasPatchLoDScaleY[ p ] )
    s = HashConsumeU32( s, AtlasPatchEntityId[ p ] )
    s = HashConsumeU32( s, AtlasPatchDepthOccThreshold[ p ] )
    s = HashConsumeI32( s, AtlasPatchTextureOffset[ p ][ 0 ] )
    s = HashConsumeI32( s, AtlasPatchTextureOffset[ p ][ 1 ] )
    s = HashConsumeI32( s, AtlasPatchTextureOffset[ p ][ 2 ] )
    s = HashConsumeU32( s, AtlasPatchInpaintFlag[ p ] )
}
hashValue = ComputeHashValue( s )
```

8.6 View params list hashing process

The inputs to this process are the variable *NumViews* and the arrays of which the name starts with View.

The output of this process is the value *hashValue* in range 0 .. 2^{31} - 1.

This process is defined as follows:

```
s = InitHashState( )
for( v = 0; v < NumViews; v++ ) {
    s = HashConsumeU32( s, ViewType[ v ] )
    s = HashConsumeU32( s, ViewProjectionPlaneWidth[ v ] )
    s = HashConsumeU32( s, ViewProjectionPlaneHeight[ v ] )
    if( ViewType[ v ] == 0 ) {
        s = HashConsumeFL32( s, ViewErpPhiMin[ v ] )
        s = HashConsumeFL32( s, ViewErpPhiMax[ v ] )
        s = HashConsumeFL32( s, ViewErpThetaMin[ v ] )
        s = HashConsumeFL32( s, ViewErpThetaMax[ v ] )
    } else if( ViewType[ v ] == 1 ) {
        s = HashConsumeFL32( s, ViewPerspectiveFocalHor[ v ] )
        s = HashConsumeFL32( s, ViewPerspectiveFocalVer[ v ] )
        s = HashConsumeFL32( s, ViewPerspectivePrincipalPointHor[ v ] )
        s = HashConsumeFL32( s, ViewPerspectivePrincipalPointVer[ v ] )
    } else if( ViewType[ v ] == 2 ) {
        s = HashConsumeFL32( s, ViewOrthoWidth[ v ] )
        s = HashConsumeFL32( s, ViewOrthoHeight[ v ] )
    }
    s = hash.consumeFL32( s, ViewPosX[ v ] )
    s = hash.consumeFL32( s, ViewPosY[ v ] )
    s = hash.consumeFL32( s, ViewPosZ[ v ] )
    s = hash.consumeFL32( s, ViewQuatW[ v ] )
    s = hash.consumeFL32( s, ViewQuatX[ v ] )
    s = hash.consumeFL32( s, ViewQuatY[ v ] )
    s = hash.consumeFL32( s, ViewQuatZ[ v ] )
    if( ViewQuantizationLaw[ v ] == 0 ) {
        s = hash.consumeFL32( s, ViewNormDispMin[ v ] )
```