INTERNATIONAL STANDARD

ISO/IEC 21794-2

First edition
2021-04

Information technology — Plenoptic image coding system (JPEG Pleno) —

Part 2:
Light field coding

Technologies de l'information — Système de codage d'images plénoptiques (JPEG Pleno) —

Partie 2: Codages des champs de lumière

Reference number
ISO/IEC 21794-2:2021(E)

Contents

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).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents) or the IEC list of patent declarations received (see patents.iec.ch).

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 21794 series can be found on the ISO website.

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

This document is part of a series of standards for a system known as JPEG Pleno. This document defines the JPEG Pleno framework. It facilitates the capture, representation, exchange and visualization of plenoptic imaging modalities. A plenoptic image modality can be a light field, point cloud or hologram, which are sampled representations of the plenoptic function in the form of, respectively, a vector function that represents the radiance of a discretized set of light rays, a collection of points with position and attribute information, or a complex wavefront. The plenoptic function describes the radiance in time and in space obtained by positioning a pinhole camera at every viewpoint in 3D spatial coordinates, every viewing angle and every wavelength, resulting in a 7D function.

JPEG Pleno specifies tools for coding these modalities while providing advanced functionality at system level, such as support for data and metadata manipulation, editing, random access and interaction, protection of privacy and ownership rights.

Information technology — Plenoptic image coding system (JPEG Pleno) —

Part 2: Light field coding

1 Scope

This document specifies a coded codestream format for storage of light field modalities as well as associated metadata descriptors that are light field modality specific. This document also provides information on the encoding tools.

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.

ITU-T Rec. T.800 | ISO/IEC 15444-1, *Information technology — JPEG 2000 image coding system — Part 1: Core coding system*

ITU-T Rec. T.801 | ISO/IEC 15444-2, *Information technology — JPEG 2000 image coding system — Part 2: Extensions*

ISO/IEC 21794-1:2020, *Information technology — Plenoptic image coding system (JPEG Pleno) — Part 1: Framework*

ISO/IEC 60559, *Information technology — Microprocessor Systems — Floating-Point arithmetic*

3 Terms and definitions

For the purposes of this document the terms and definitions given in ISO/IEC 21794-1 and the following apply.

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

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

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

3.1
arithmetic coder
entropy coder that converts variable length strings to variable length codes (encoding) and vice versa (decoding)

3.2
bit-plane
two-dimensional array of bits

3.3
4D bit-plane
four-dimensional array of bits

3.4
coefficient
numerical value that is the result of a transformation or linear regression

3.5
compression
reduction in the number of bits used to represent source image data

3.6
depth
distance of a point in 3D space to the camera plane

3.7
disparity view
image that for each pixel of the subaperture view contains the apparent pixel shift between two subaperture views along either horizontal or vertical axis

3.8
hexadeca-tree
division of a 4D region into 16 (sixteen) 4D subregions

3.9
pixel
collection of sample values in the spatial image domain having all the same sample coordinates

EXAMPLE A pixel may consist of three samples describing its red, green and blue value.

3.10
plenoptic function
amount of radiance in time and in space by positioning a pinhole camera at every viewpoint in 3D spatial coordinates, every viewing angle and every wavelength, resulting in a 7D representation

3.11
reference view
subaperture view that is used as one of the references to generate the intermediate views

3.12
subaperture view
subaperture image
image taken of the 3D scene by a pinhole camera positioned at a particular viewpoint and viewing angle

3.13
texture
pixel attributes

EXAMPLE Colour information, opacity, etc.

3.14
transform
transformation
mathematical mapping from one signal space to another

4 Symbols and abbreviated terms

4.1 Symbols

Codestream_Body()	coded image data in the codestream without Codestream_Header()
Codestream_Header()	codestream header preceding the image data in the codestream
$\tilde{D}^{DEC}(t,s,v,u)$	decoded normalized disparity value at view (t,s) for pixel location (v,u)
$\tilde{D}(t,s,v,u)$	normalized disparity value at view (t,s) for pixel location (v,u)
$DPEC_k$	pointer to contiguous codestream for normalized disparity view k
D_{shift}	scaling parameter to translate quantized normalized disparity maps to positive range
DCODEC	disparity view codec type
f	focal length
FPW_p	fixed-weight merging parameter for view p
$H(t,s)$	view hierarchy value for view (t,s)
$HCC(t,s)$	horizontal camera centre coordinate for view (t,s)
$H_D(t,s)$	binary value defining the availability of a normalized disparity view (t,s)
J_0	Lagrangian encoding cost
J_1	Lagrangian encoding cost of spatial partitioning
J_2	Lagrangian encoding cost of view partitioning
$KR_{p,c}$	sparse filter regressor mask of texture component c for view p
LightField()	JPEG Pleno light field codestream
$LSW_j^{p,c}$	quantized least-squares merging weight of texture component c for view p, $j=1,2,\ldots,NLS_p$
MIDV	absolute value of the minimum value over all quantized normalized disparity views

$MMODE_p$	view merging mode for intermediate view p
MSP_p	sparse filter order for view p
NLS_p	number of least-squares merging coefficients for intermediate view p
NRT_p	regressor template size parameter for sparse filter for view p
NC	number of components in an image
N_I	number of intermediate views
N_{NDV}	number of reference normalized disparity views
N_p^D	number of normalized disparity reference views for intermediate view p
N_p^T	number of texture reference views for intermediate view p
N_{REF}	number of reference views
N_{RES}	number of prediction residual views
N_{sp}	total available number of regressors for sparse filter
Plev	level a particular codestream complies to
Ppih	profile a particular codestream complies to
Q_p	2D image of dimensions $V \times U$, defines the occlusion state-based segmentation at intermediate view p
Q	normalized disparity quantization parameter
R	rate or bitrate, expressed in bit per sample
RCODEC	prediction residual view codec type
RDATA	array of bytes containing for a single prediction residual view the RCODEC codestream after header information has been stripped
RENCODING	array of bytes containing for a single prediction residual view the full DCODEC codestream
RGB	colour data for the red, green and blue colour component of a pixel
RHEADER	array of bytes containing for a single prediction residual view the header information from the RCODEC codestream

$RPEC_j$	pointer to contiguous codestream for prediction residual view j
s	coordinate of the addressed subaperture image along the *s*-axis
S	size of the light field image along the *s*-axis (COLUMNS)
s_{ii}^{Tr}	subscript of the column index of the reference view, $ii = 1, 2, \ldots, N_p^T$ in the light field array in row-wise scanning order
s_{jj}^{Dr}	subscript of the column index of the reference normalized disparity view, $jj = 1, 2, \ldots, N_p^D$ in the light field array in row-wise scanning order
SF_p	binary variable, determines if sparse filter is used (true) or not (false)
$SPW_j^{p,0}$	quantized sparse filter coefficients of texture component c for view p, $j = 1, 2, \ldots, MSP_p$
$\widehat{SPW}_j^{p,c}$	de-quantized sparse filter coefficients of texture component c for view p, $j = 1, 2, \ldots, MSP_p$
t	coordinate of the addressed subaperture image along the *t*-axis
T	size of the light field image along the *t*-axis (ROWS)
t_{ii}^{Tr}	subscript of the row index of the reference view, $ii = 1, 2, \ldots, N_p^T$ in the light field array in row-wise scanning order
t_{jj}^{Dr}	subscript of the row index of the reference normalized disparity view, $jj = 1, 2, \ldots, N_p^D$ in the light field array in row-wise scanning order
$\left(t_k^D, s_k^D\right)$	view coordinate subscripts for normalized disparity view k
$\left(t_l^X, s_l^X\right)$	view coordinate subscripts for reference view l
$\left(t_p^I, s_p^I\right)$	view coordinate subscripts for intermediate view p
$t_k \times s_k \times v_k \times u_k$	4D block dimensions at the 4D block partitioning stage
$t_b \times s_b \times v_b \times u_b$	4D block dimensions at the bit-plane hexadeca-tree decomposition stage
TCODEC	reference view codec type
TDATA	array of bytes, containing for a single reference view, the TCODEC codestream, after header information has been stripped

TENCODING	array of bytes, containing for a single reference view the full TCODEC codestream
THEADER	array of bytes, containing for a single reference view the header information from the TCODEC codestream
$TPEC_l$	pointer to contiguous codestream for reference view l
u	sample coordinate along the u-axis within the addressed subaperture image
U	size of the subaperture image along the u-axis (WIDTH)
v	sample coordinate along the v-axis within the addressed subaperture image
V	size of the subaperture image along the v-axis (HEIGHT)
$VCC(t,s)$	vertical camera centre coordinate for view (t,s)
VPP_p	view prediction parameters for intermediate view p
$X(t,s,v,u,c)$	texture value at view (t,s) for pixel location (v,u) for texture component c
$X^{DEC}(t,s,v,u,c)$	decoded texture value at view (t,s) for pixel location (v,u) for texture component c
$X_W^{(t_1,s_1)}(t_2,s_2)$	result of warping the texture view (t_1,s_1) to view location (t_2,s_2)
Δx	horizontal distance between a pair of camera centres
Δy	vertical distance between a pair of camera centres
YCbCr	colour data for the luminance, the blue chrominance and the red chrominance component of a pixel
$z(t,s,v,u)$	depth value at view (t,s) for pixel location (v,u)
$\hat{\theta}_i^p$	distance based merging weight for reference view $i=1,\ldots,N_p^T$ at intermediate view p
α_i^p	distance based factor, used for defining the merging weight, at intermediate view p for reference view $i=1,\ldots,N_p^T$
Γ_p	binary matrix, defining the locations of the non-zero merging weights in merging weight matrix $\Theta_{p,c}$ at intermediate view p. It is identical between all colour components c

$\theta_j^{p,c}$	de-quantized least-squares merging weight of texture component c for view p, $j = 1, 2, \ldots, NLS_p$
$\theta_{p,c}^{sp}$	sparse filter coefficients at intermediate view p for colour component c
$\Theta_{p,c}$	merging weight matrix for intermediate view p for colour component c
$Y_{p,c}$	locations of the non-zero elements of $\Psi_{(v,u)}$
$\Psi_{(v,u)}$	regressor template at pixel location (v,u)
Ω_p^{Dr}	set of reference normalized disparity views for intermediate view p
Ω_p^{occlD}	set of occluded pixels, which remain to be inpainted, during normalized disparity view synthesis at intermediate view p
Ω_p^{occlT}	set of occluded pixels, which remain to be inpainted, during texture view synthesis at intermediate view p
Ω_p^{Tr}	set of reference views for intermediate view p

4.2 Abbreviated terms

2D	two dimensional
3D	three dimensional
4D	four dimensional
DCT	discrete cosine transform
floating point	floating point notation as specified in ISO/IEC 60559
HTTP	hypertext transfer protocol
IDCT	inverse DCT
IPR	intellectual property rights
IV	intermediate view; subaperture view that is generated from surrounding reference view(s)
JPEG	Joint Photographic Experts Group
JPL	JPEG Pleno file format
LSB	least significant bit

MSB	most significant bit
R-D	rate-distortion
RV	reference view
URL	uniform resource locator
XML	eXtensible Markup Language

5 Conventions

5.1 Naming conventions for numerical values

Integer numbers are expressed as bit patterns, hexadecimal values or decimal numbers. Bit patterns and hexadecimal values have both a numerical value and an associated particular length in bits.

Hexadecimal notation, indicated by prefixing the hexadecimal number by "0x", may be used instead of binary notation to denote a bit pattern having a length that is an integer multiple of 4. For example, 0x41 represents an eight-bit pattern having only its second most significant bit and its least significant bit equal to 1. Numerical values that are specified under a "**Code**" heading in tables that are referred to as "code tables" are bit pattern values (specified as a string of digits equal to 0 or 1 in which the left-most bit is considered the most-significant bit). Other numerical values not prefixed by "0x" are decimal values. When used in expressions, a hexadecimal value is interpreted as having a value equal to the value of the corresponding bit pattern evaluated as a binary representation of an unsigned integer (i.e. as the value of the number formed by prefixing the bit pattern with a sign bit equal to 0 and interpreting the result as a two's complement representation of an integer value). For example, the hexadecimal value 0xF is equivalent to the 4-bit pattern '1111' and is interpreted in expressions as being equal to the decimal number 15.

5.2 Operators

NOTE Many of the operators used in document are similar to those used in the C programming language.

5.2.1 Arithmetic operators

+	addition
−	subtraction (as a binary operator) or negation (as a unary prefix operator)
×	multiplication
/	division without truncation or rounding
<<	left shift; x<<s is defined as $x \times 2^s$
>>	right shift; x>>s is defined as $\lfloor x/2^s \rfloor$
++	increment with 1
--	decrement with 1

umod	x umod a is the unique value y between 0 and a–1 for which y+Na = x with a suitable integer N
&	bitwise AND operator; compares each bit of the first operand to the corresponding bit of the second operand If both bits are 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.
^	bitwise XOR operator; compares each bit of the first operand to the corresponding bit of the second operand If both bits are equal, the corresponding result bit is set to 0. Otherwise, the corresponding result bit is set to 1.

5.2.2 Logical operators

||	logical OR
&&	logical AND
!	logical NOT

5.2.3 Relational operators

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to

5.2.4 Precedence order of operators

Operators are listed in descending order of precedence. If several operators appear in the same line, they have equal precedence. When several operators of equal precedence appear at the same level in an expression, evaluation proceeds according to the associativity of the operator either from right to left or from left to right.

Operators	Type of operation	Associativity
()	expression	left to right
[]	indexing of arrays	left to right
++, --	increment, decrement	left to right
!, –	logical not, unary negation	

×, /	multiplication, division	left to right
umod	modulo (remainder)	left to right
+, −	addition and subtraction	left to right
&	bitwise AND	left to right
^	bitwise XOR	left to right
&&	logical AND	left to right
||	logical OR	left to right
<<, >>	left shift and right shift	left to right
< , >, <=, >=	relational	left to right

5.2.5 Mathematical functions

$\|x\|$	absolute value, is –x for x < 0, otherwise x
sign(x)	sign of x, zero if x is zero, +1 if x is positive, -1 if x is negative
clamp(x,min,max)	clamps x to the range [min,max]: returns min if x < min, max if x > max or otherwise x
$\lceil x \rceil$	ceiling of x; returns the smallest integer that is greater than or equal to x
$\lfloor x \rfloor$	floor of x; returns the largest integer that is less than or equal to x
$\lfloor x \rceil$	rounding of x to the nearest integer, equivalent to $sign(x)\lfloor \|x\|+0.5 \rfloor$

6 General

6.1 Functional overview on the decoding process

This document specifies the JPEG Pleno Light Field superbox and the JPEG Pleno light field decoding algorithm. The generic JPEG Pleno Light Field superbox syntax is specified in Annex A.

The specified light field decoding algorithm distinguishes two coding modes:

— **4D Transform mode:** this mode is specified in Annex B and is based on a 4D inverse discrete cosine transform (IDCT) and 4D block partitioning and 4D bit-plane hexadeca-tree decoding;

— **4D Prediction mode:** this mode is based the prediction of intermediate views based on reference views and normalized disparity maps. The signalling syntax and decoding of the reference views is addressed in Annex C, the normalized disparity views in Annex D, and the prediction parameters and residual views in Annex E. The intermediate views are reconstructed in a decoding process that involves view warping, view merging and prediction error correction.

The overall architecture (Figure 1) provides the flexibility to configure the encoding and decoding system depending on the requirements of the addressed use case.

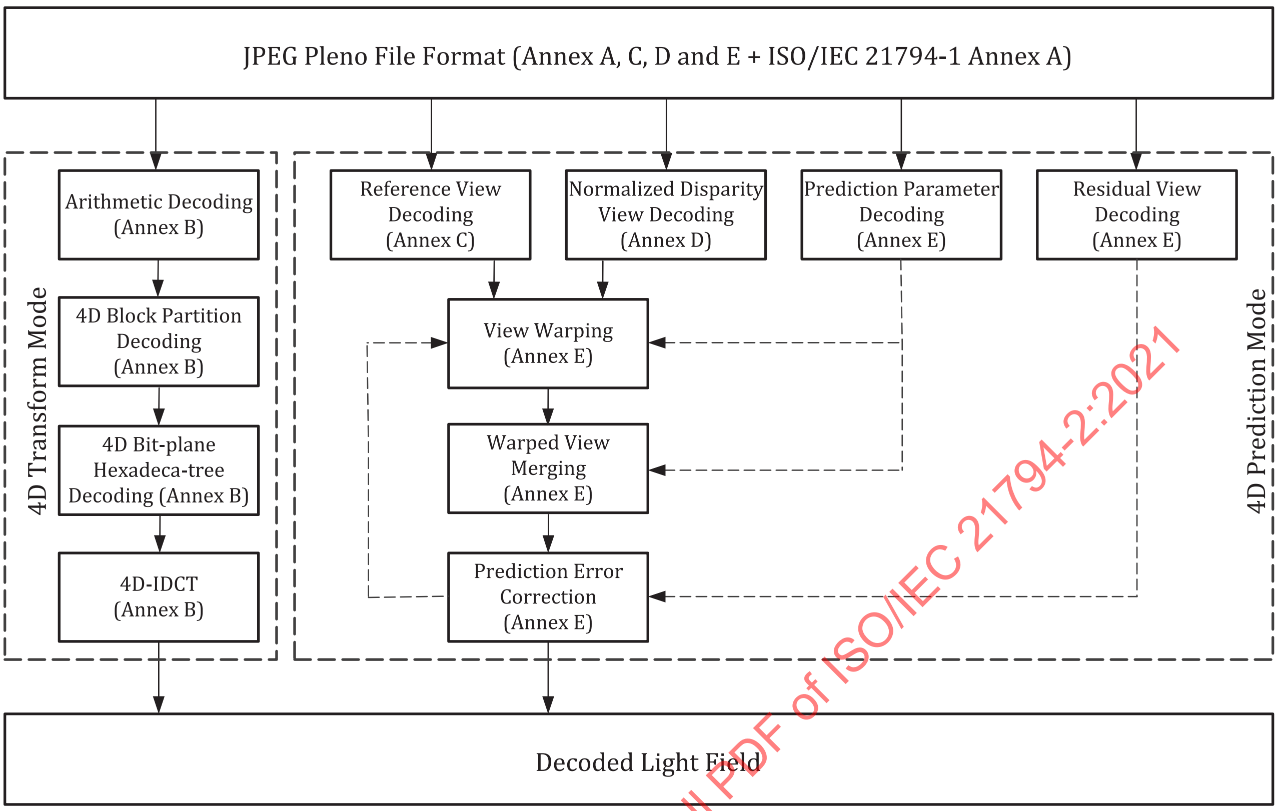

JPEG Pleno File Format (Annex A, C, D and E + ISO/IEC 21794-1 Annex A)
4D Transform Mode
Arithmetic Decoding (Annex B)
4D Block Partition Decoding (Annex B)
4D Bit-plane Hexadeca-tree Decoding (Annex B)
4D-IDCT (Annex B)
Reference View Decoding (Annex C)
Normalized Disparity View Decoding (Annex D)
Prediction Parameter Decoding (Annex E)
Residual View Decoding (Annex E)
View Warping (Annex E)
Warped View Merging (Annex E)
Prediction Error Correction (Annex E)
4D Prediction Mode
Decoded Light Field

Figure 1 — Generic JPEG Pleno light field decoder architecture

6.2 Encoder requirements

An encoding process converts source light field data to coded light field data.

In order to conform with this document, an encoder shall conform with the codestream format syntax and file format syntax specified in the annexes for the encoding process(es) embodied by the encoder.

6.3 Decoder requirements

A decoding process converts coded light field data to reconstructed light field data. Annexes A through E describe and specify the decoding process.

A decoder is an embodiment of the decoding process. In order to conform to this document, a decoder shall convert all, or specific parts of, any coded light field data that conform to the file format syntax and codestream syntax specified in Annex A to E to a reconstructed light field.

7 Organization of the document

Annex A specifies the description of the JPEG Pleno Light Field superbox.

This document specifies two approaches to represent a compressed representation of light field data: the 4D Transform mode is specified in Annex B and the 4D Prediction mode is specified Annex C, Annex D and Annex E. Annex C details the signalling of the reference view data, Annex D the signalling of the normalized disparity views and finally, Annex E the signalling of the prediction parameters to generate the intermediate views and residual view data to compensate for prediction errors.

Annex A
(normative)

JPEG Pleno Light Field superbox

A.1 General

This annex specifies the use of the JPEG Pleno Light Field superbox which is designed to contain compressed light field data and associated metadata. The listed boxes shall comply with their definitions as specified in ISO/IEC 21794-1.

This document may redefine the binary structure of some boxes defined as part of the ISO/IEC 15444-1 or ISO/IEC 15444-2 file formats. For those boxes, the definition found in this document shall be used for all JPL files.

A.2 Organization of the JPEG Pleno Light Field superbox

Figure A.1 shows the hierarchical organization of the JPEG Pleno Light Field superbox contained by a JPL file. This illustration does not specify nor imply a specific order to these boxes. In many cases, the file will contain several boxes of a particular box type. The meaning of each of those boxes is dependent on the placement and order of that particular box within the file.

This superbox is composed out of the following core elements:

— a JPEG Pleno Light Field Header box containing parameterization information about the light field such as size and colour parameters;

— a JPEG Pleno Light Field Reference View box containing the compressed reference views of the light field;

— a JPEG Pleno Light Field Disparity View box signalling disparity information for all or a subset of subaperture views;

— a JPEG Pleno Light Field Intermediate View box containing prediction parameters and eventual compressed residual signals for subaperture views not encoded as reference views.

Table A.1 lists all boxes defined as part of this document. Boxes defined as part of the ISO/IEC 15444-1 or ISO/IEC 15444-2 file formats are not listed. A box that is listed in Table A.1 as "Required" shall exist within all conforming JPL files. For the placement of and restrictions on each box, see the relevant section defining that box.

Note that the IPR, XML, UUID and UUID boxes introduced in Annex A can be signalled, as well at the level of the JPEG Pleno Light Field box, to carry light field specific metadata.

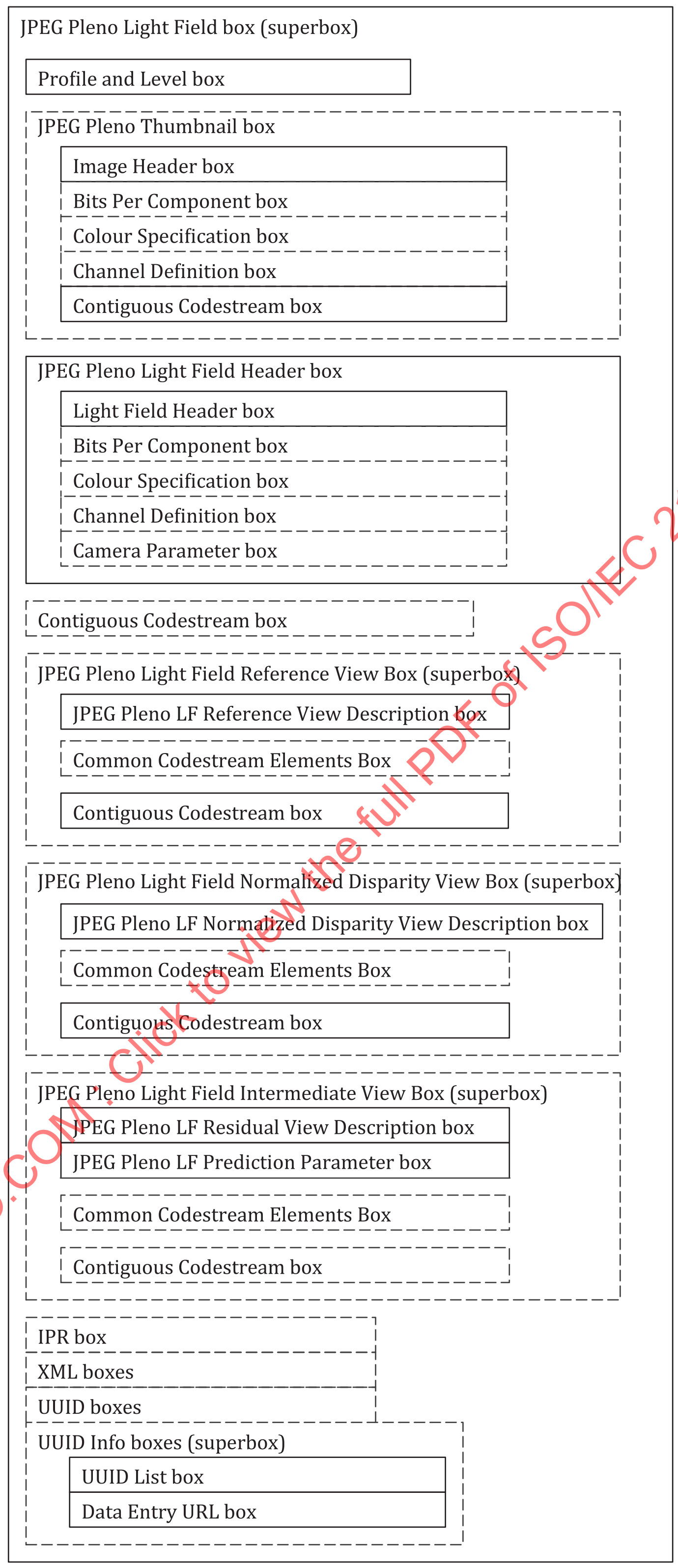

JPEG Pleno Light Field box (superbox)
Profile and Level box
JPEG Pleno Thumbnail box
Image Header box
Bits Per Component box
Colour Specification box
Channel Definition box
Contiguous Codestream box
JPEG Pleno Light Field Header box
Light Field Header box
Bits Per Component box
Colour Specification box
Channel Definition box
Camera Parameter box
Contiguous Codestream box
JPEG Pleno Light Field Reference View Box (superbox)
JPEG Pleno LF Reference View Description box
Common Codestream Elements Box
Contiguous Codestream box
JPEG Pleno Light Field Normalized Disparity View Box (superbox)
JPEG Pleno LF Normalized Disparity View Description box
Common Codestream Elements Box
Contiguous Codestream box
JPEG Pleno Light Field Intermediate View Box (superbox)
JPEG Pleno LF Residual View Description box
JPEG Pleno LF Prediction Parameter box
Common Codestream Elements Box
Contiguous Codestream box
IPR box
XML boxes
UUID boxes
UUID Info boxes (superbox)
UUID List box
Data Entry URL box

Figure A.1 — Hierarchical organization of a JPEG Pleno Light Field superbox

A.3 Defined boxes

A.3.1 Overview

The following boxes shall properly be interpreted by all conforming readers. Each of these boxes conforms to the standard box structure as defined in ISO/IEC 21794-1:2020, Annex A. The following clauses define the value of the DBox field. It is assumed that the LBox, TBox and XLBox fields exist for each box in the file as defined in ISO/IEC 21794-1:2020, Annex A.

Table A.1 — Defined boxes

Box name	Type	Superbox	Required?	Comments
JPEG Pleno Light Field box	‘jplf’ (0x6A70 6C66)	Yes	Yes	This box contains a series of boxes that contain the encoded light field, its parameterization and associated metadata. (Defined in ISO/IEC 21794-1:2020, Annex A)
JPEG Pleno Profile and Level box	‘jppl’ (0x6A70 706C)	No	Yes	This box indicates to which profile and associated level the file format and codestream complies. (Defined in Annex A.3.2)
JPEG Pleno Light Field Header box	'jplh' (0x6A70 6C68)	Yes	Yes	This box contains generic information about the file, such as the number of components, bits per component and colour space. (Defined in Annex A.3.3)
Light Field Header box	‘lhdr’ (0x6C68 6472)	No	Yes	This box contains fixed length generic information about the light field, such as light field dimensions, subaperture image size, number of components, codec and bits per component. (Defined in Annex A.3.3.2)
Camera Parameter box	‘lfcp’ (0x6C66 6370)	No	No	This box signals intrinsic and extrinsic camera parameters for calibration of the light field data. (Defined in Annex A.3.3.3)
Contiguous Codestream box	'jp2c' (0x6A70 3263)	No	No	This box contains a JPEG Pleno codestream (Defined in Annex A.3.4)
JPEG Pleno Light Field Reference View superbox	‘lfrv’ (0x6C66 7276)	Yes	No	This box contains a series of boxes that contain the encoded reference views and their associated parameters. (Defined in Annex C.2)
JPEG Pleno Light Field Reference View Description box	‘lfrd’ (0x6C66 7264)	No	No	This box signals which views are encoded as reference views and their encoding configuration. (Defined in Annex C.3.1)
Common Codestream Elements box	‘lfcc’ (0x6C66 6363)	No	No	This box contains the redundant part of the signalled codestreams. (Defined in Annex C.3.2)
JPEG Pleno Light Field Normalized Disparity View superbox	‘lfdv’ (0x6C66 6476)	Yes	No	This box contains a series of boxes that contain the encoded normalized disparity views and their associated parameters. (Defined in Annex D.2)
JPEG Pleno Light Field Normalized Disparity View Description box	‘lfdd’ (0x6C66 6464)	No	No	This box signals for which views normalized disparity information is signalled and their encoding configuration. (Defined in Annex D.3.1)

Table A.1 *(continued)*

Box name	Type	Superbox	Required?	Comments
JPEG Pleno Light Field Intermediate View superbox	‘lfiv’ (0x6C66 6976)	Yes	No	This box contains a series of boxes that contain both the prediction parameters for the intermediate views and the encoded residual views. (Defined in Annex E.2)
JPEG Pleno Light Field Prediction Parameter box	‘lfpp’ (0x6C66 7070)	No	No	This box signals prediction parameter information for the intermediate views. (Defined in Annex E.3.1)
JPEG Pleno Light Field Residual View Description box	‘lfre’ (0x6C66 7265)	No	No	This box signals the encoding configuration for the residual views containing the prediction errors. (Defined in Annex E.3.2)

A.3.2 JPEG Pleno Profile and Level box

The conformance to profiles is indicated in the file type box by the addition of the compatible profiles as brands within the compatibility list. Derived and application specifications based on this specification may define additional brands.

The type of the JPEG Pleno Profile and Level box shall be ‘jppl’ (0x6A70 706C) and contents of the box shall have the organization as in Figure A.2 and format as in Table A.2.

Key

Ppih	profile of the codestream
Plev	level of the codestream

Figure A.2 — Organization of the contents of a JPEG Pleno Profile and Level box

Table A.2 — Format of the contents of the JPEG Pleno Profile and Level box

Field name	Size (bits)	Value
Ppih	16	Reserved for future ISO/IEC use
Plev	16	Reserved for future ISO/IEC use

A.3.3 JPEG Pleno Light Field Header box

A.3.3.1 General

The JPEG Pleno Header box contains generic information about the file, such as the number of components, bits per component and colour space. This box is a superbox. Within a JPL file, there shall be one and only one JPEG Pleno Header box. The JPEG Pleno Header box may be located anywhere within the file after the File Type box but before the Contiguous Codestream box. It also must be at the same level as the JPEG Pleno Signature and File Type boxes (it shall not be inside any other superbox within the file).

The type of the JPEG Pleno Header box shall be 'jplh' (0x6A70 6C68).

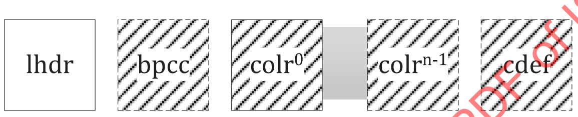
This box contains several boxes. Other boxes may be defined in other documents and may be ignored by conforming readers. Those boxes contained within the JPEG Pleno Header box that are defined within this document are shown in Figure A.3:

— The Light Field Header box specifies information about the reference grid geometry, bit depth and the number of components. This box shall be the first box in the JPEG Pleno Header box and is specified in A.3.3.2.

— The Bits Per Component box specifies the bit depth of the components in the file in cases where the bit depth is not constant across all components. Its structure shall be as specified in ISO/IEC 15444-1.

— The Colour Specification boxes specify the colour space of the decompressed image. Their structures shall be as specified in ISO/IEC 15444-2. There shall be at least one Colour Specification box within the JPEG Pleno Header box. The use of multiple Colour Specification boxes provides the ability for a decoder to be given multiple optimization or compatibility options for colour processing. These boxes may be found anywhere in the JPEG Pleno Header box provided that they come after the Light Field Header box. All Colour Specification boxes shall be contiguous within the JPEG Pleno Header box.

— The Channel Definition box defines the channels in the image. Its structure shall be as specified in ISO/IEC 15444-1. This box may be found anywhere in the JPEG Pleno Header box, provided that it comes after the Light Field Header box.



Key

lhdr	Light Field Header box
bppc	Bits Per Component box
colr^i	Colour Specification boxes
cdef	Channel Definition box

Figure A.3 — Organization of the contents of a JPEG Pleno Header box

A.3.3.2 Light Field Header box

A.3.3.2.1 General

This box contains fixed length generic information about the light field, such as light field dimensions, subaperture image size, number of components, codec and bits per component. The contents of the JPEG Pleno Header box shall start with a Light Field Header box. Instances of this box in other places in the file shall be ignored. The length of the Light Field Header box shall be 30 bytes, including the box length and type fields. Much of the information within the Light Field Header box is redundant with information stored in the codestream itself.

All references to "the codestream" in the descriptions of fields in this Light Field Header box apply to the codestream found in the first Contiguous Codestream box in the file. Files that contain contradictory information between the Light Field Header box and the first codestream are not conforming files. However, readers may choose to attempt to read these files by using the values found within the codestream.

The type of the Light Field Header box shall be 'lhdr' (0x6C68 6472) and the contents of the box shall have the format as in Figure A.4 and Table A.3:

— **ROWS (T):** The value of this parameter indicates the number of rows of the subaperture view array. This field is stored as a 4-byte big-endian unsigned integer.

— **COLUMNS (*S*):** The value of this parameter indicates the number of columns of the subaperture view array. This field is stored as a 4-byte big-endian unsigned integer.

— **HEIGHT (*V*):** The value of this parameter indicates the height of the sample grid. This field is stored as a 4-byte big-endian unsigned integer.

— **WIDTH (*U*):** The value of this parameter indicates the width of the sample grid. This field is stored as a 4-byte big-endian unsigned integer.

— **NC:** This parameter specifies the number of components in the codestream and is stored as a 2-byte big-endian unsigned integer. The value of this field shall be equal to the value of the NC field in the LFC marker in the codestream (as defined in B.3.2.6.3). If no Channel Definition Box is available, the order of the components for colour images is R-G-B-Aux or Y-U-V-Aux.

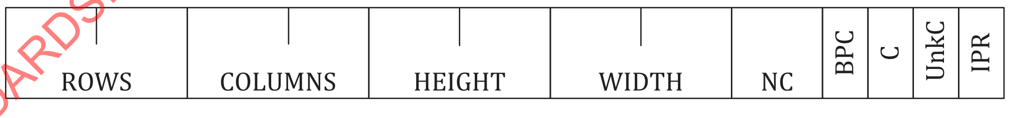
— **BPC:** This parameter specifies the bit depth of the components in the codestream, minus 1, and is stored as a 1-byte field (Table A.4).

The low 7-bits of the value indicate the bit depth of the components. The high-bit indicates whether the components are signed or unsigned. If the high-bit is 1, then the components contain signed values. If the high-bit is 0, then the components contain unsigned values. If the components vary in bit depth or sign, or both, then the value of this field shall be 255 and the Light Field Header box shall also contain a Bits Per Component box defining the bit depth of each component (as defined in A.3.3.2.2).

— **C:** This parameter specifies the compression algorithm used to compress the image data. It is encoded as a 1-byte unsigned integer. It the value is 0, the 4D Transform mode coding is activated. If the value is 1, the 4D Prediction mode is activated. All other values are reserved for ISO/IEC use.

— **UnkC:** This field specifies if the actual colour space of the image data in the codestream is known. This field is encoded as a 1-byte unsigned integer. Legal values for this field are 0, if the colour space of the image is known and correctly specified in the Colour Space Specification boxes within the file, or 1 if the colour space of the light field is not known. A value of 1 will be used in cases such as the transcoding of legacy images where the actual colour space of the image data is not known. In these cases, while the colour space interpretation methods specified in the file may not accurately reproduce the image with respect to an original, the image should be treated as if the methods do accurately reproduce the image. Values other than 0 and 1 are reserved for ISO/IEC use.

— **IPR:** This parameter indicates whether this JPL file contains intellectual property rights information. If the value of this field is 0, this file does not contain rights information, and thus the file does not contain an IPR box. If the value is 1, then the file does contain rights information and thus does contain an IPR box as defined in ISO/IEC 15444-1. Other values are reserved for ISO/IEC use.



Key

ROWS (*T*)	number of rows of the subaperture view array
COLUMNS (*S*)	number of columns of the subaperture view array
HEIGHT (*V*)	subaperture view height
WIDTH (*U*)	subaperture view width
NC	number of components
BPC	bits per component
C	compression type
UnkC	colour space unknown
IPR	intellectual property

Figure A.4 — Organization of the contents of a Light Field Header box

Table A.3 — Format of the contents of the Light Field Header box

Field name	Size (bits)	Value
ROWS	32	1 to (2^{32}– 1)
COLUMNS	32	1 to (2^{32}– 1)
HEIGHT	32	1 to (2^{32}– 1)
WIDTH	32	1 to (2^{32}– 1)
NC	16	1 to 16 384
BPC	8	See Table A.4
UnkC	8	0 to 1
IPR	8	0 to 1

Table A.4 — BPC values

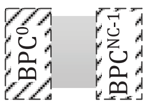
Values (bits) MSB LSB	Component sample precision
x000 0000 to x010 0101	Component bit depth = value + 1. From 1 bit deep to 38 bits deep respectively (counting the sign bit, if appropriate).
0xxx xxxx	Components are unsigned values.
1xxx xxxx	Components are signed values.
1111 1111	Components vary in bit depth.
	All other values reserved for ISO/IEC use.

A.3.3.2.2 Bits Per Component box

The Bits Per Component box specifies the bit depth of each component. If the bit depth of all components in the codestream is the same (in both sign and precision), then this box shall not be found. Otherwise, this box specifies the bit depth of each individual component. The order of bit depth values in this box is the actual order in which those components are enumerated within the codestream. The exact location of this box within the JPEG Pleno Header box may vary provided that it follows the Light Field Header box.

There shall be one and only one Bits Per Component box inside a JPEG Pleno Header box.

The type of the Bits Per Component box shall be 'bpcc' (0x6270 6363). The contents of this box shall be as in Table A.5 and Figure A.5.



Key

BPC^i	bits per component

Figure A.5 — Organization of the contents of a Bits Per Component box

Table A.5 — Format of the contents of the Bits Per Component box

Field name	Size (bits)	Value
BPC^i	8	See Table A.6

This parameter specifies the bit depth of component *i*, minus 1, encoded as a 1-byte value (Table A.6). The ordering of the components within the Bits Per Component box shall be the same as the ordering of the components within the codestream. The number of BPC^i fields shall be the same as the value of the NC field from the Light Field Header box. The value of this field shall be equivalent to the respective Ssiz^i field in the LFC marker in the codestream. The low 7-bits of the value indicate the bit depth of this component. The high-bit indicates whether the component is signed or unsigned. If the high-bit is 1, then the component contains signed values. If the high-bit is 0, then the component contains unsigned values.

Table A.6 — BPC^i values

Values (bits) MSB LSB	Component sample precision
x000 0000 to x010 0101	Component bit depth = value + 1. From 1 bit deep to 38 bits deep respectively (counting the sign bit, if appropriate).
0xxx xxxx	Components are unsigned values.
1xxx xxxx	Components are signed values.
	All other values reserved for ISO/IEC use.

A.3.3.3 Camera Parameter box

A.3.3.3.1 General

The Camera Parameter box provides information on the positioning of the local reference grid in the global reference grid, its size, and calibration information about the light field. This box is optional.

Camera models can be represented by matrices with particular properties that represent the mapping of the 3D world coordinate system to the image coordinate system. This 3D to 2D transform depends on a number of parameters, known as the intrinsic and extrinsic parameters.

A.3.3.3.2 JPEG Pleno camera parameters

As specified in ISO/IEC 21794-1, JPEG Pleno provides a mechanism to co-register plenoptic data contained by the JPL file on a 3D reference grid system. This reference grid system exists out of a global and a local reference grid. The global reference grid allows the positioning of the individual modalities in the represented 3D scene. In addition, each JPEG Pleno Light Field, Point Cloud and Hologram box shall be assigned a local reference grid to address their sampled plenoptic data. This local reference grid is specified by signalling its translation and rotation with respect to the global reference grid. The rotation angles shall be determined utilizing the right-hand rule for curve orientation.

The parameters related to the local reference grid are signalled per plenoptic object in the associated JPEG Pleno Light Field, Point Cloud or Hologram box.

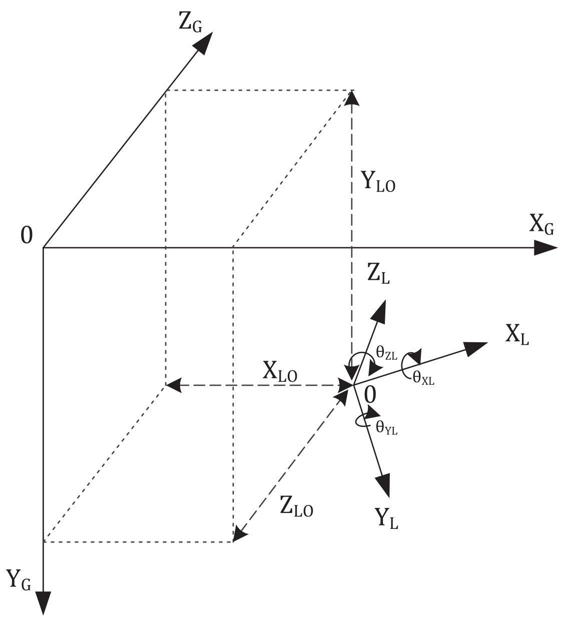


Figure A.6 — The global and local reference grid

In Figure A.6, boundaries and coordinate axes of the global and one local reference grid are shown. In each case, the samples or coefficients coincident with the left and upper boundaries are included in a given bounding box, while samples or coefficients along the right and/or lower boundaries are not included in that bounding box.

A.3.3.3.3 Camera modelling and calibration

A.3.3.3.3.1 General

The camera modelling and calibration is described based on the local reference grid (X_L, Y_L, Z_L) in Figure A.6. Both intrinsic and extrinsic parameters are being signalled to model and calibrate the camera setting and behaviour. The intrinsic parameters are the camera parameters that are internal and fixed to a particular camera/digitization setup, allowing the mapping between camera coordinates and pixel coordinates in the image plane.[1] The extrinsic parameters are the camera parameters that are external to the camera and may change with respect to the 3D local reference grid, defining the location and orientation of the camera with respect to the 3D local reference grid coordinate system.

A.3.3.3.3.2 Intrinsic camera parameters

Considering a pinhole camera model, the centre of projection is the ‘optical centre’ (**C** in Figure A.7). The camera's ‘principal axis’ (Z_{CAM} in Figure A.7) is the line perpendicular to the image plane that passes through the pinhole. Its intersection with the image plane is known as the ‘principal point’ (**p** in Figure A.7) and it is the geometrical centre of the image. The parameters u_0 and v_0 are the principal points offsets, which are the coordinates of the principal point relative to the coordinate axes u and v (Figure A.7).

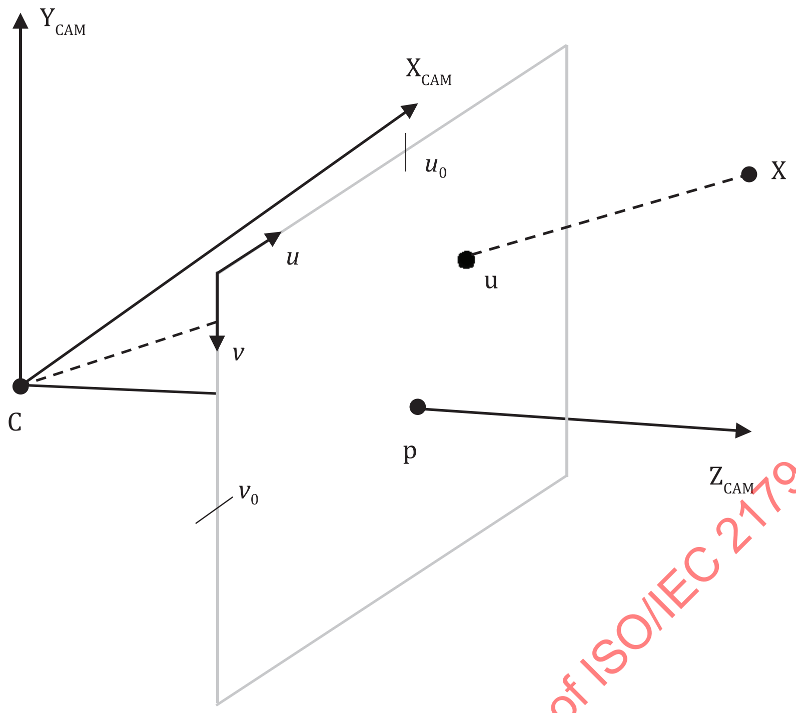


Figure A.7 — Pinhole camera geometry

The focal lengths f_u and f_v correspond to the distance between the optical centres of the cameras and their respective image planes. They are represented in terms of pixel dimensions in the *u* and *v* directions. For example, if the camera focal length is given in mm one needs to convert it to pixel dimensions using Formulae (A.1) and (A.2). For square pixels the sensor height is equal to the sensor width, and f_u is equal to f_v.

$$f_u \text{ (pixel)} = \text{(focal length (mm) / sensor width (mm))} \times \text{image width (pixel)} \tag{A.1}$$

$$f_v \text{ (pixel)} = \text{(focal length (mm) / sensor height (mm))} \times \text{image height (pixel)} \tag{A.2}$$

The axis skew parameter *sk* causes shear distortion in the projected image, and for most of the cameras its value is equal to zero. The parameters f_u , f_v , sk , u_0 and v_0 completely characterize the mapping of an image point from camera to pixels coordinates. They are known as the intrinsic or internal parameters of a camera system and can be represented by the transformation matrix **K** in Formula (A.3):

$$\mathbf{K} = \begin{bmatrix} f_u & sk & u_0 \\ 0 & f_v & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{A.3}$$

The matrix **K** is known as the calibration matrix. In general, the mapping from 3D local reference grid to the image is linear. A camera system is said to be calibrated when its intrinsic parameters are known, otherwise it is an uncalibrated camera system.

A.3.3.3.3.3 Extrinsic camera parameters

The parameters that relate the camera orientation and position to a 3D local reference grid coordinate system are called the extrinsic or external parameters. The geometric quantities (rotational and translational components) describing the relative position and orientation of the cameras are called the extrinsic parameters of the camera system. The rotation and translation can be represented in an extrinsic matrix taking the form of a rigid transformation matrix: a 3×3 rotation matrix **R**, and a 3×1 translation column-vector $\mathbf{t}_M = (XCC, YCC, ZCC)^T$ that can be represented in a 3×4 matrix, as in Formula (A.4).

$$\mathrm{R}|\mathrm{t}_M = \left[\begin{array}{ccc|c} r_{11} & r_{12} & r_{13} & XCC \\ r_{21} & r_{22} & r_{23} & YCC \\ r_{31} & r_{32} & r_{33} & ZCC \end{array}\right] \tag{A.4}$$

Matrix **P** (Formula (A.5)), known as the projection matrix, or camera matrix, represents the pose of the 3D local reference grid coordinates relative to the image coordinates. It contains 6 independent parameters (Degrees of Freedom - DoF): 3 for rotation and 3 for translation. These parameters are expressed in the local reference grid (Figure A.6).

$$\mathrm{P}=\mathrm{K}\left[\mathrm{R}\,|\,\mathrm{t}_M\right] \tag{A.5}$$

The 3×4 matrix P relates the sensor plane 2D image coordinates $\mathbf{u} = (u, v)$ ($\tilde{\boldsymbol{u}} = (u, v, 1)$ in homogeneous coordinates) to the 3D local reference grid coordinates $\mathbf{X} = (X_L, Y_L, Z_L)$ ($\tilde{\mathbf{X}} = (X_L, Y_L, Z_L, 1)$ in homogeneous coordinates) via Formula (A.5). The mapping between a point in the 3D world into a 2D image is given by $\tilde{\boldsymbol{u}} = \mathbf{P}\tilde{\boldsymbol{X}}$, where $\tilde{\boldsymbol{u}}$ is the image point in homogeneous coordinates, **P** is the camera matrix and $\tilde{\mathbf{X}}$ is the 3D local reference grid point in homogeneous coordinates. The extrinsic camera parameter matrix represents the current status of the camera in the 3D scene.

For example, a rotation in 3D local reference grid space involves an axis around which to rotate, and an angle of rotation, according to the right-handed coordinates.

Formulae (A.6), (A.7) and (A.8) show the rotation matrix values for rotations around these 3 axes:

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_{X_{CAM}} & -\sin\theta_{X_{CAM}} \\ 0 & \sin\theta_{X_{CAM}} & \cos\theta_{X_{CAM}} \end{bmatrix}, \text{ rotation around the } X_L \text{ axis, rotates } Y_L,\ Z_L, \text{ leaving the } X_L \text{ coordinates fixed} \tag{A.6}$$

$$R_y = \begin{bmatrix} \cos\theta_{Y_{CAM}} & 0 & \sin\theta_{Y_{CAM}} \\ 0 & 1 & 0 \\ -\sin\theta_{Y_{CAM}} & 0 & \cos\theta_{Y_{CAM}} \end{bmatrix}, \text{ rotation around the } Y_L \text{ axis, rotates } X_L,\ Z_L, \text{ leaving the } Y_L \text{ coordinates fixed} \tag{A.7}$$

$$R_z = \begin{bmatrix} \cos\theta_{Z_{CAM}} & \sin\theta_{Z_{CAM}} & 0 \\ -\sin\theta_{Z_{CAM}} & \cos\theta_{Z_{CAM}} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$, rotation around the Z_L axis, rotates X_L, Y_L, leaving the Z_L coordinates fixed. (A.8)

Any rotation can be expressed as a combination of the three rotations about the three axes, as per Formula (A.9):

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_{X_{CAM}} & -\sin\theta_{X_{CAM}} \\ 0 & \sin\theta_{X_{CAM}} & \cos\theta_{X_{CAM}} \end{bmatrix} \begin{bmatrix} \cos\theta_{Y_{CAM}} & 0 & \sin\theta_{Y_{CAM}} \\ 0 & 1 & 0 \\ -\sin\theta_{Y_{CAM}} & 0 & \cos\theta_{Y_{CAM}} \end{bmatrix} \begin{bmatrix} \cos\theta_{Z_{CAM}} & \sin\theta_{Z_{CAM}} & 0 \\ -\sin\theta_{Z_{CAM}} & \cos\theta_{Z_{CAM}} & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{A.9}$$

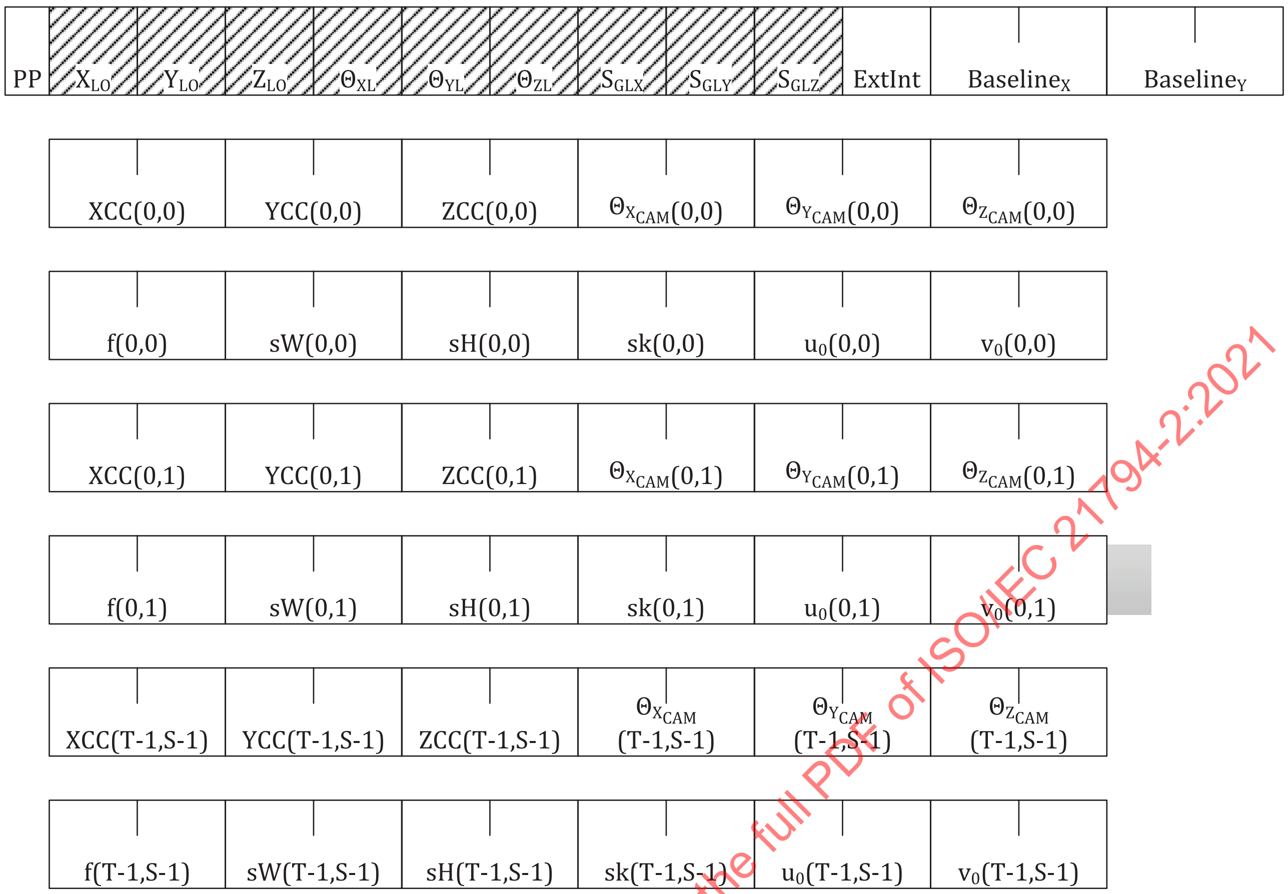
Formula (A.10) shows the mapping of a 3-D local reference grid point (X_L, Y_L, Z_L) to the image coordinate system (u,v) , for a calibrated system:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_u & sk & u_0 \\ 0 & f_v & v_0 \\ 0 & 0 & 1 \end{bmatrix} \left[\begin{array}{ccc|c} r_{11} & r_{12} & r_{13} & XCC \\ r_{21} & r_{22} & r_{23} & YCC \\ r_{31} & r_{32} & r_{33} & ZCC \end{array} \right] \begin{bmatrix} X_L \\ Y_L \\ Z_L \\ 1 \end{bmatrix} \tag{A.10}$$

A.3.3.3.4 Camera Parameter box definition

The type of the Camera Parameter box shall be 'lfcp' (0x6C66 6370) and the contents of the box shall have the format as in Figure A.8.

If the Camera Parameter box is not signalled, all parameters specified in Figure A.8 and Table A.7 shall be initialized to zero. The scaling values S_{GLX}, S_{GLY} and S_{GLZ} will be initialised to 1.



Key

Light field position in global reference grid

PP	precision of coordinates (Precision Prec = 16×2^{PP})
X_{LO}	position of the origin of the local reference grid in the global reference system along the X_G coordinate axis
Y_{LO}	position of the origin of the local reference grid in the global reference system along the Y_G coordinate axis
Z_{LO}	position of the origin of the local reference grid in the global reference system along the Z_G coordinate axis
θ_{XL}	rotation offset around the X_G axis (in rad)
θ_{YL}	rotation offset around the Y_G axis (in rad)
θ_{ZL}	rotation offset around the Z_G axis (in rad)
S_{GLX}	scaling of local reference grid system with respect to global reference grid system for the X-axes before rotation
S_{GLY}	scaling of local reference grid system with respect to global reference grid system for the Y-axes before rotation
S_{GLZ}	scaling of local reference grid system with respect to global reference grid system for the Z-axes before rotation

Extrinsic parameters for pinhole camera corresponding to subaperture view (t, s)

ExtInt	signals which extrinsic and intrinsic camera parameters are signalled
Baseline_X	horizontal camera baseline, used when XCC(t,s) = $s \times \text{Baseline}_X$ + XCC(0,0), and hence, XCC(0,1), XCC(0,2), …, XCC(T-1,S-1) do not need to be signalled
Baseline_Y	vertical camera baseline, used when YCC(t,s) = $t \times \text{Baseline}_Y$ + YCC(0,0), and hence, YCC(0,1), YCC(0,2), …, YCC(T-1,S-1) do not need to be signalled
XCC (*t, s*)	camera centre of subaperture view (t, s) in local reference grid along X_L coordinate axis

YCC(*t*, s)	camera centre of subaperture view (*t*, s) in local reference grid along Y_L coordinate axis
ZCC (*t*, s)	camera centre of subaperture view (*t*, s) in local reference grid along Z_L coordinate axis
$\theta_{X_{CAM}}$ **(*t*, s)**	camera rotation offset around the X_L axis (in rad)
$\theta_{Y_{CAM}}$ **(*t*, s)**	camera rotation offset around the Y_L axis (in rad)
$\theta_{Z_{CAM}}$ **(*t*, s)**	camera rotation offset around the Z_L axis (in rad)

Intrinsic parameters for pinhole camera corresponding to subaperture view (t, s)

f (*t*,*s*)	focal length (in mm)
sW (*t*,*s*)	sensor width (in mm)
sH (*t*,*s*)	sensor height (in mm)
sk (*t*,*s*)	sensor skew
u_0 **(*t*,*s*)**	horizontal principle point offset
v_0 **(*t*,*s*)**	vertical principle point offset

NOTE 1 **PP** indicates the floating-point precision issued for the coordinates.

NOTE 2 X_{LO}, Y_{LO}, Z_{LO}, θ_{XL}, θ_{YL}, θ_{ZL}, S_{GLX}, S_{GLY}, S_{GLZ}, $\theta_{X_{CAM}}$ **(*t*, s)**, $\theta_{Y_{CAM}}$ **(*t*, s)** and $\theta_{Z_{CAM}}$ **(*t*, s)** utilize the chosen floating-point precision.

Figure A.8 — Organization of the contents of the Camera Parameter box

The geometrical coordinates of the centre of the camera when acquiring the view (*t*, *s*) are denoted as

$XCC\left(t,s\right)$, camera centre on X_L

$YCC\left(t,s\right)$, camera centre on Y_L

$ZCC\left(t,s\right)$, camera centre on Z_L.

An example of camera centre coordinates for a planar camera array is given in Figure A.9, where both the horizontal and vertical coordinates are illustrated for five views in the camera array. The camera centres XCC and YCC are used together with the normalized disparity maps $\tilde{D}$ to obtain horizontal and vertical disparity maps between a pair of views in the light field. For usage examples see Annex D.4 and Annex E.4.

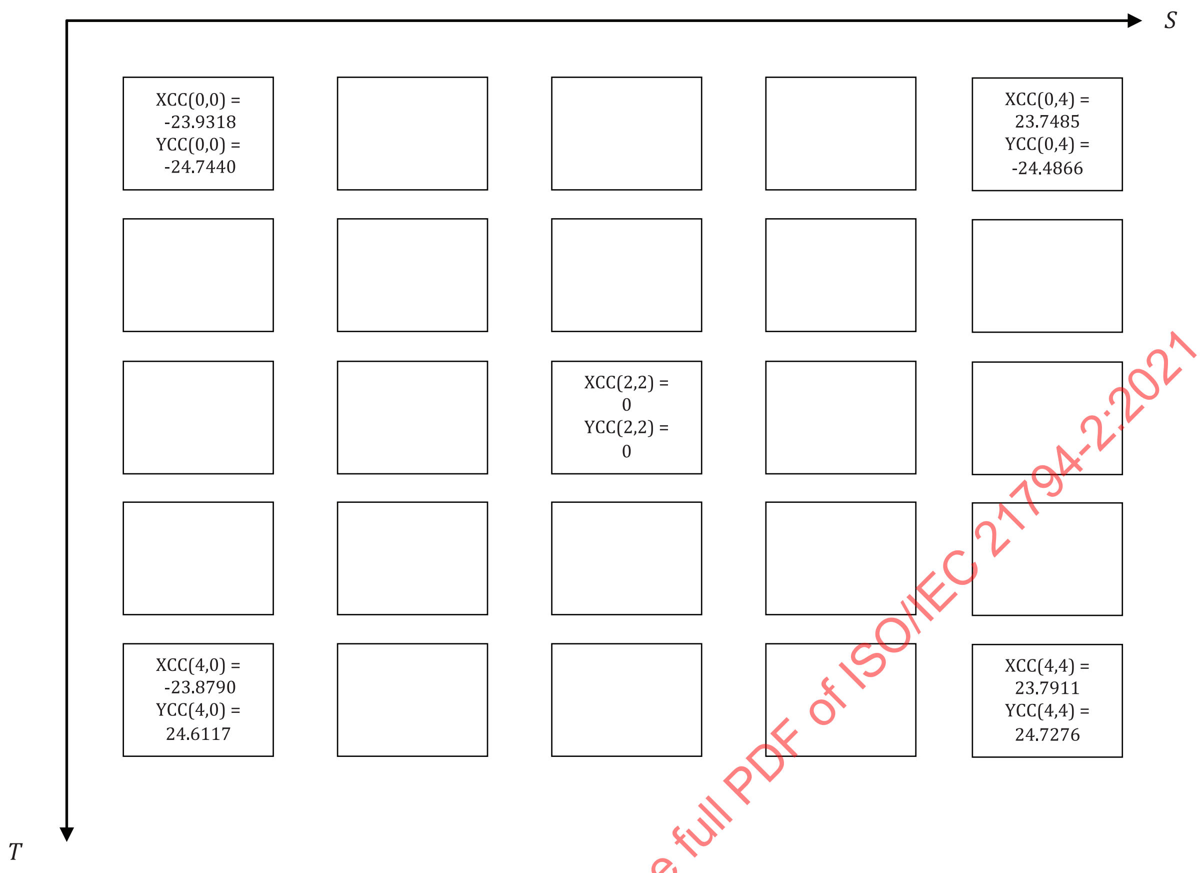

S
T
XCC(0,0) = -23.9318 YCC(0,0) = -24.7440
XCC(0,4) = 23.7485 YCC(0,4) = -24.4866
XCC(2,2) = 0 YCC(2,2) = 0
XCC(4,0) = -23.8790 YCC(4,0) = 24.6117
XCC(4,4) = 23.7911 YCC(4,4) = 24.7276

Figure A.9 — Organization of subaperture views and associated planar camera calibration information

Table A.7 — Format of the contents of the Camera Parameter box

Field name	Size (bits)	Value
PP	8	0 to (2^8-1)
X_{LO}	variable	big endian, floating point
Y_{LO}	variable	big endian, floating point
Z_{LO}	variable	big endian, floating point
θ_{XL}	variable	big endian, floating point
θ_{YL}	variable	big endian, floating point
θ_{ZL}	variable	big endian, floating point
S_{GLX}	variable	big endian, floating point
S_{GLY}	variable	big endian, floating point
S_{GLZ}	variable	big endian, floating point
ExtInt	16	See Table A.8
$Baseline_X$	32	single precision, big endian floating-point
$Baseline_Y$	32	single precision, big endian floating-point
XCC(0,0)	32	single precision, big endian floating-point
YCC(0,0)	32	single precision, big endian floating-point
ZCC(0,0)	32	single precision, big endian floating-point

Table A.7 *(continued)*

Field name	Size (bits)	Value
$\theta_{Xcam}(0,0)$	32	single precision, big endian floating-point
$\theta_{Ycam}(0,0)$	32	single precision, big endian floating-point
$\theta_{Zcam}(0,0)$	32	single precision, big endian floating-point
f(0,0)	32	single precision, big endian floating-point
sW(0,0)	32	single precision, big endian floating-point
sH(0,0)	32	single precision, big endian floating-point
sk(0,0)	32	single precision, big endian floating-point
$u_0(0,0)$	32	single precision, big endian floating-point
$v_0(0,0)$	32	single precision, big endian floating-point
XCC(0,1)	32	single precision, big endian floating-point
YCC(0,1)	32	single precision, big endian floating-point
...	...	...
$u_0(T\text{-}1,S\text{-}1)$	32	single precision, big endian floating-point
$v_0(T\text{-}1,S\text{-}1)$	32	single precision, big endian floating-point

Table A.8 — Meaning of ExtInt bits

Bit position	Value	Meaning
0 (LSB)	0	XCC(t,s) = s × BaselineX + XCC(0,0) Signal BaselineX and XCC(0,0), the remaining (T×S)-1 XCC entries in Table A.7 are not signalled
	1	The T×S XCC(t,s) entries in Table A.7 are signalled
1	0	YCC(t,s) = t × BaselineY + YCC(0,0) Signal BaselineY and YCC(0,0), the remaining (T×S)-1 YCC entries in Table A.7 are not signalled
	1	The T×S YCC(t,s) entries in Table A.7 are signalled
2	0	ZCC(t,s) = ZCC(0,0) and the remaining (T×S)-1 ZCC entries in Table A.7 are not signalled
	1	The T×S ZCC(t,s) entries in Table A.7 are signalled
3	0	$\theta_{Xcam}(t,s) = \theta_{Xcam}(0,0)$ and the remaining (T×S)-1 entries in Table A.7 are not signalled
	1	The T×S $\theta_{Xcam}(t,s)$ entries in Table A.7 are signalled
4	0	$\theta_{Ycam}(t,s) = \theta_{Ycam}(0,0)$ and the remaining (T×S)-1 entries in Table A.7 are not signalled
	1	The T×S $\theta_{Ycam}(t,s)$ entries in Table A.7 are signalled
5	0	$\theta_{Zcam}(t,s) = \theta_{Zcam}(0,0)$ and the remaining (T×S)-1 entries in Table A.7 are not signalled
	1	The T×S $\theta_{Zcam}(t,s)$ entries in Table A.7 are signalled
6	0	f(t,s) = f(0,0) and the remaining (T×S)-1 entries in Table A.7 are not signalled
	1	The T×S f(t,s) entries in Table A.7 are signalled
7	0	sW(t,s) = sW(0,0) and the remaining (T×S)-1 entries in Table A.7 are not signalled
	1	The T×S sW(t,s) entries in Table A.7 are signalled
8	0	sH(t,s) = sH(0,0) and the remaining (T×S)-1 entries in Table A.7 are not signalled
	1	The T×S sH(t,s) entries in Table A.7 are signalled
9	0	sk(t,s) = sk(0,0) and the remaining (T×S)-1 entries in Table A.7 are not signalled
	1	The T×S sk(t,s) entries in Table A.7 are signalled

Table A.8 *(continued)*

Bit position	Value	Meaning
10	0	u0(t,s) = $\lfloor U/2 \rfloor$ and the (T×S) entries in Table A.7 are not signalled
	1	The T×S u0(t,s) entries in Table A.7 are signalled.
11	0	0(t,s) = $\lfloor V/2 \rfloor$ and the (T×S) entries in Table A.7 are not signalled
	1	The T×S v0(t,s) entries in Table A.7 are signalled
12-15	0	Reserved for future ISO/IEC use

A.3.4 Contiguous Codestream box

The Contiguous Codestream box contains a JPEG Pleno codestream.

The type of a Contiguous Codestream box shall be 'jp2c' (0x6A70 3263). The contents of the box shall be as in Figure A.10 and Table A.9:

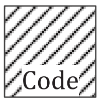


Figure A.10 — Organization of the contents of the Contiguous Codestream box

Table A.9 — Format of the contents of the Contiguous Codestream box

Field name	Size (bits)	Value
Code	Variable	Variable

Code This field contains valid and complete JPEG Pleno codestream components as specified in Annexes B, C, D and E.

Annex B
(normative)

4D transform mode

B.1 General

This annex describes an instantiation of 4D Transform mode encoder. Next, the codestream syntax is specified and subsequently the light field decoding process is detailed.

B.2 4D transform mode encoding

B.2.1 High-level coding architecture

In the 4D transform mode, the light field is encoded with a four-step process (Figure B.1). First, the 4D light field data is divided into fixed-sized 4D blocks that are independently encoded according to a predefined and fixed scanning order. However, if any light field dimensions are not multiple of such fixed-sized 4D blocks, the sizes of the 4D blocks at the light field boundaries shall be truncated to fit in the light field dimensions. The initial blocks can be further partitioned into a set of non-overlapping 4D sub-blocks, where the optimal partitioning parameters are derived based on a rate-distortion (R-D) criterion. Each sub-block is independently transformed by a variable block-size 4D DCT. Subsequently, the transformed blocks are quantized and entropy coded using hexadeca-tree bit plane decomposition and adaptive arithmetic encoding, producing a compressed representation of the light field. This coding procedure is applied to each colour component independently.

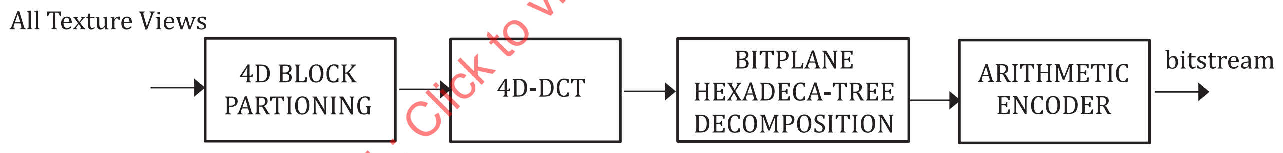


Figure B.1 — 4D transform mode encoding architecture

A sample (pixel) of the light field is referenced in a 4D coordinate system along the *t, s*, *v* and *u*-axes, where *t* and *s* are representing the coordinates of the addressed subaperture view, and *v* and *u* the sample coordinates within the subaperture images as illustrated in Figure B.2. The blocks are scanned in the directions *t, s*, *v* and *u*, with direction *u* corresponding to the inner loop of the scan. In Table B.1 a pseudo-code describes the scan of the 4D blocks. Each 4D block will generate a separate codestream embedded in the codestream, which can be independently decoded in support of random access.

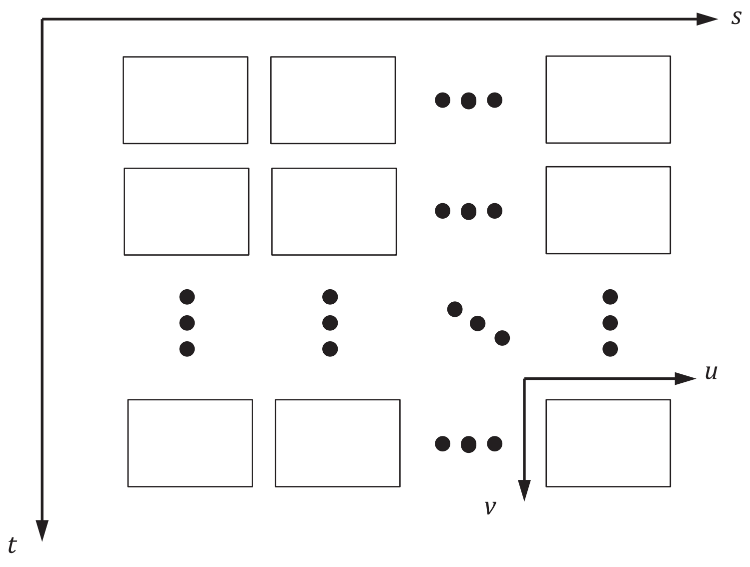


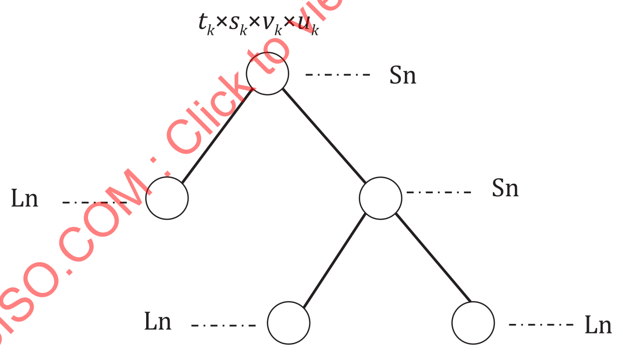
Figure B.2 — 4D structure

The 4D block partitioning, as well as the clustering of the bit-planes of transform coefficient – for efficient encoding – are signalled using tree structures (Figure B.3). The partitioning of the 4D blocks in sub-blocks is signalled with a binary tree using ternary flags. These flags signal whether:

— a block is transformed as is;

— is split into 4 blocks in the *s,t* (view) dimensions;

— is split into 4 blocks in the *u,v* (spatial) dimensions.



Key

Sn	split node
Ln	leaf node

Figure B.3 — Binary tree representing 4D-block partitioning of a $t_k \times s_k \times v_k \times u_k$ 4D block

Before subsequently applying the 4D-DCT on the sub-blocks, a level-shift operation is performed to reduce the dynamic range requirements of the DCT (Annex B.2.3.1). The deployed 4D DCT (Annex B.2.3.2) is separable, i.e. with 1D transforms computed separately in each of the 4 directions. An example of the computation flow of the 4D separable transform is depicted in Figure B.4. The order of the 1D transforms is arbitrary.

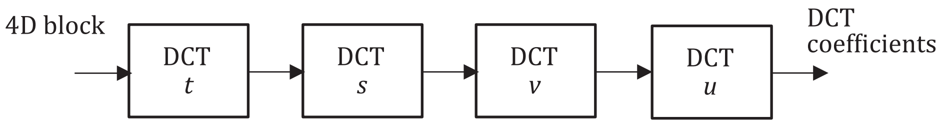


Figure B.4 — Separable forward 4D-DCT

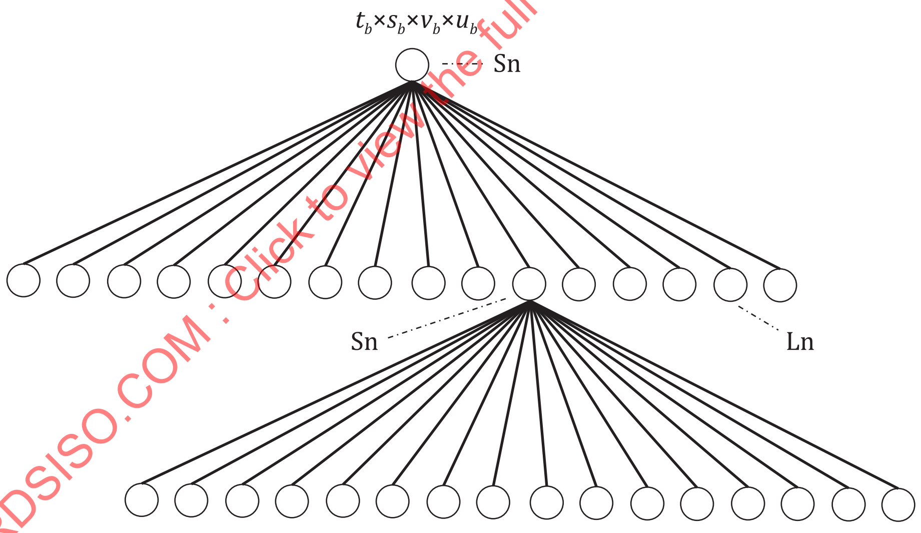
After the 4D-DCT is performed, the set of transform coefficients is sliced into 4D bit-planes. A transform coefficient is considered non-significant on a 4D bit-plane if its bits belonging to higher 4D bit-planes are all zero. Otherwise, the transform coefficient is considered to be significant. A hexadeca-tree with ternary flags is used to group the non-significant transform coefficients and thus localize the significant transform coefficients (Figure B.5). The ternary flags signal that:

— either a block of transform coefficients containing a significant coefficient at the current bit-plane is split into 16 blocks in the four *t*, *s*, *v*, *u* dimensions,

— or a block of transform coefficients not containing any significant coefficient at the current bit-plane is not split,

— or a block of transform coefficients containing a significant coefficient at the current bit-plane is discarded.

The 4D bit-planes are scanned from the most significant bit-plane to the least significant one, where the least significant 4D bit-plane being determined by the desired quantization level. Both the hexadeca-tree bits and the bits from the transform coefficients are encoded using an adaptive arithmetic coder.



Key

Sn	Split node
Ln	Leaf node

Figure B.5 — Hexadeca-tree representing the clustering of bit-planes of 4D transform coefficients of a $t_b \times s_b \times v_b \times u_b$ 4D block

The structure that clusters the non-significant 4D transform coefficients and thus localizes the significant ones is a hexadeca-tree, that is encoded using ternary flags. They signal that a block of transform coefficients containing a significant coefficient at the current bit-plane is split into 16 blocks in the four *t*, *s*, *v*, *u* dimensions, or that a block of transform coefficients not containing any significant coefficient at the current bit-plane is not split, or that a block of transform coefficients containing a significant coefficient at the current bit-plane is discarded. The 4D bit-planes are scanned from the most significant to the least significant one, where the least significant 4D bit-plane being determined by the desired quantization level. Both the hexadeca-tree bits and the bits from the coefficients are encoded using an adaptive arithmetic coder. The hexadeca-tree structure mentioned above is depicted in Figure B.5.

The following two sections provide more insight on the partitioning strategy and the entropy and quantization steps.

B.2.2 Optimize partitioning

The optimized partitioning for each block in Figure B.6 is obtained as follows: initially each block is transformed by a full-size DCT (B.2.3.2), and the Lagrangian encoding cost, J_0 is evaluated (Table B.5). This cost is defined as $J_0 = D + \lambda R$, where D is the distortion incurred when representing the original block by the quantized version and R is the rate needed to encode it. The procedure of Table B.5 is used to evaluate this cost.

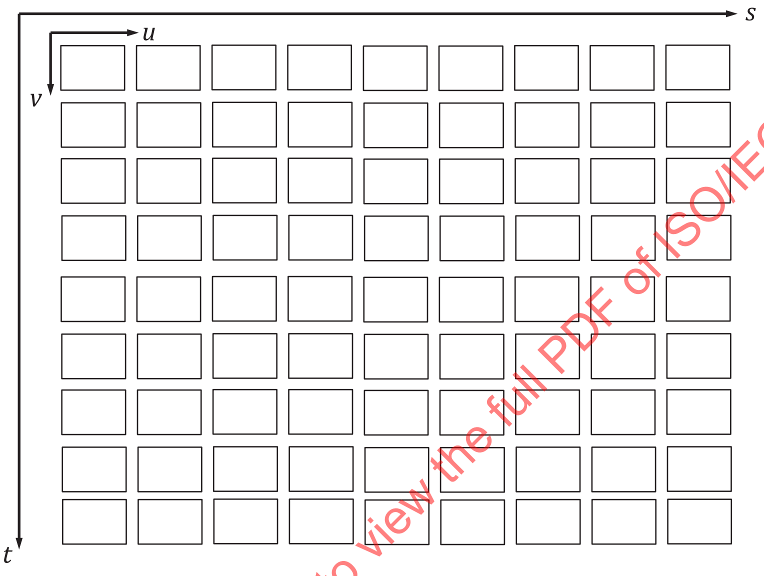


Figure B.6 — 4D Block of a light field

Next, the block can be partitioned in four sub-blocks each one with approximately a quarter of the pixels in the spatial dimensions. For example, let us consider a block B of dimensions $t_k \times s_k \times v_k \times u_k$. This block (pictured in Figure B.7 and Figure B.8) will be subdivided in four sub-blocks of sizes $t_k \times s_k \times \lfloor v_k/2 \rfloor \times \lfloor u_k/2 \rfloor$, $t_k \times s_k \times \lfloor v_k/2 \rfloor \times (u_k - \lfloor u_k/2 \rfloor)$, $t_k \times s_k \times (v_k - \lfloor v_k/2 \rfloor) \times \lfloor u_k/2 \rfloor$ and $t_k \times s_k \times (v_k - \lfloor v_k/2 \rfloor) \times (u_k - \lfloor u_k/2 \rfloor)$, respectively.

Figure B.8 shows a 4D block with dimensions $t_k \times s_k \times v_k \times u_k$ in the root node. When applying the spatial split (signalled with the *spatialSplit* flag) to the root node, the tree in Figure B.8 is obtained. Figure B.7 illustrates the four ways that a single view is partitioned using the *spatialSplit* flag, corresponding to four nodes of Figure B.8. The optimal partition for each sub-block is computed by means of the recursive procedure described in Table B.5 and the Lagrangian costs of the four sub-blocks are added to compute the Lagrangian cost J_S (Spatial R-D cost).

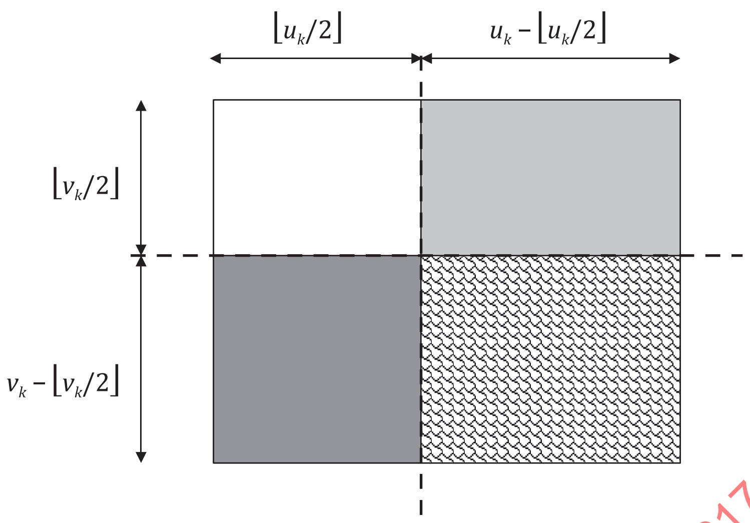
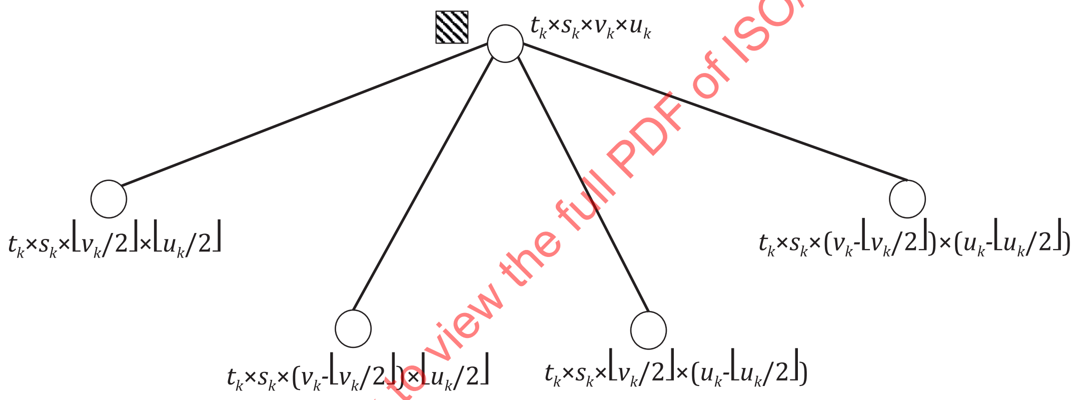


Figure B.7 — Spatial partitioning of block with spatial dimensions $v_k \times u_k$


$t_k \times s_k \times v_k \times u_k$
$t_k \times s_k \times \lfloor v_k/2 \rfloor \times \lfloor u_k/2 \rfloor$
$t_k \times s_k \times (v_k - \lfloor v_k/2 \rfloor) \times (u_k - \lfloor u_k/2 \rfloor)$
$t_k \times s_k \times (v_k - \lfloor v_k/2 \rfloor) \times \lfloor u_k/2 \rfloor$
$t_k \times s_k \times \lfloor v_k/2 \rfloor \times (u_k - \lfloor u_k/2 \rfloor)$

Key

○	Node
▨	*spatialSplit* flag

Figure B.8 — Hierarchical 4D partitioning of a 4D block with dimensions $t_k \times s_k \times v_k \times u_k$ using the *spatialSplit* flag

The block can be partitioned in four sub-blocks each one with approximately a quarter of the pixels in the view dimensions. For example, let us consider again a block B of dimensions $t_k \times s_k \times v_k \times u_k$. This block (pictured in Figure B.9 and Figure B.10) will be subdivided in four sub-blocks of sizes $\lfloor t_k/2 \rfloor \times \lfloor s_k/2 \rfloor \times v_k \times u_k$ $\lfloor t_k/2 \rfloor \times (s_k - \lfloor s_k/2 \rfloor) \times v_k \times u_k$, $(t_k - \lfloor t_k/2 \rfloor) \times \lfloor s_k/2 \rfloor \times v_k \times u_k$, $(t_k - \lfloor t_k/2 \rfloor) \times (s_k - \lfloor s_k/2 \rfloor) \times v_k \times u_k$, respectively Figure B.10). When applying the view split (signalled with the *viewSplit* flag) to the root node, the tree in Figure B.10 is obtained. Figure B.9 illustrates the four ways that a 4D block is partitioned using the *viewlSplit* flag, corresponding to four the nodes of Figure B.10. The optimal partition for each sub-block is computed by means of the recursive procedure described in Table B.5 and the Lagrangian costs of the four sub-blocks are added to compute the Lagrangian cost J_V (View R-D cost).

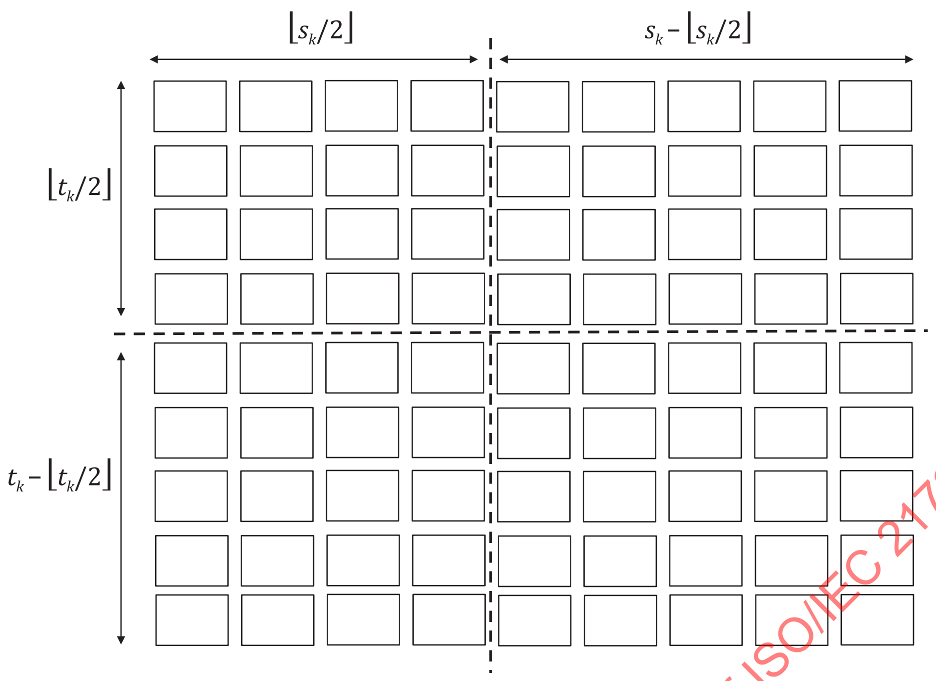
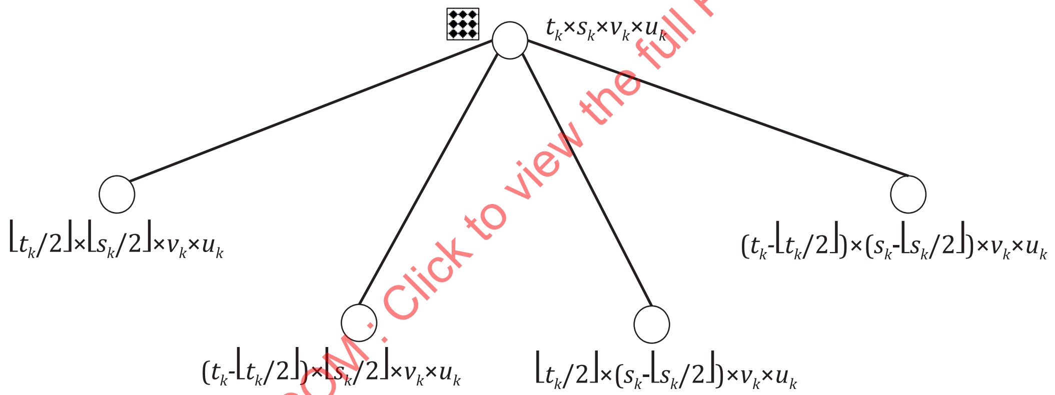


Figure B.9 — View partitioning of a 4D block of dimensions $t_k \times s_k \times v_k \times u_k$



$t_k \times s_k \times v_k \times u_k$
$\lfloor t_k/2 \rfloor \times \lfloor s_k/2 \rfloor \times v_k \times u_k$
$(t_k - \lfloor t_k/2 \rfloor) \times (s_k - \lfloor s_k/2 \rfloor) \times v_k \times u_k$
$(t_k - \lfloor t_k/2 \rfloor) \times \lfloor s_k/2 \rfloor \times v_k \times u_k$
$\lfloor t_k/2 \rfloor \times (s_k - \lfloor s_k/2 \rfloor) \times v_k \times u_k$

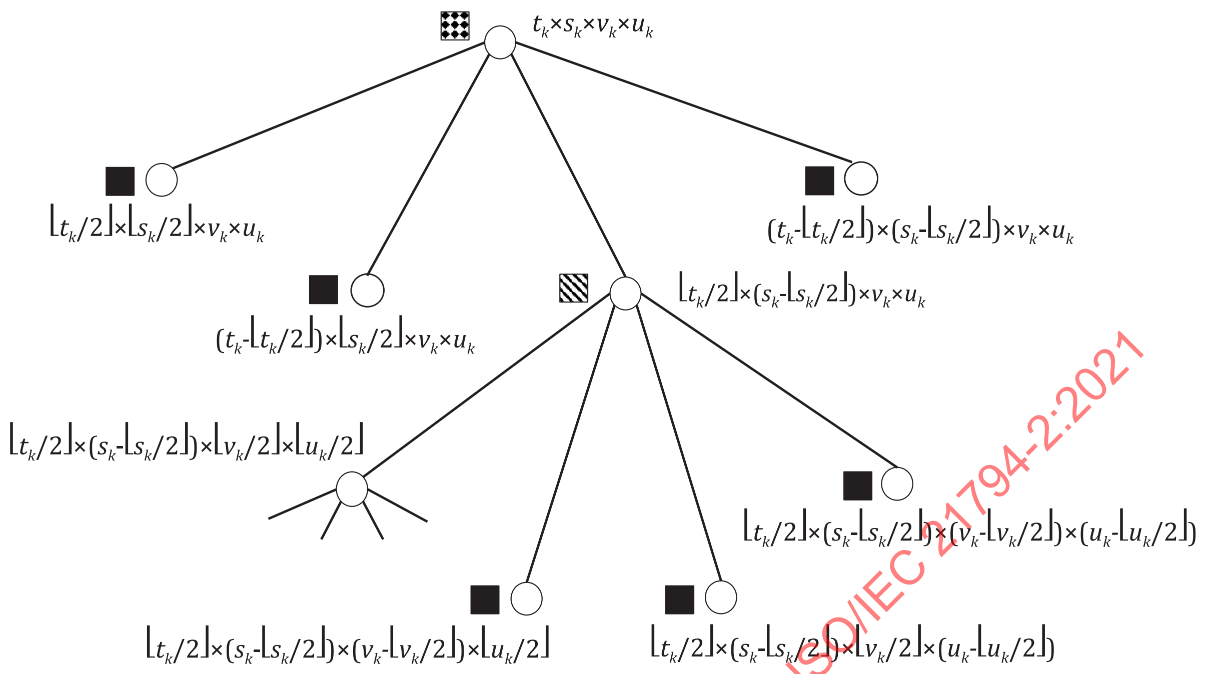
Key

○	Node
▨	*viewSplit* flag

Figure B.10 — Hierarchical 4D partitioning of a 4D block with dimensions $t_k \times s_k \times v_k \times u_k$, using the *viewSplit* flag

Finally, the three Lagrangian costs (J_0, J_S and J_V) are compared (Table B.5) and the one presenting the lowest value is chosen.

The recursive partition procedure (Table B.5) keeps track of this tree (Table B.11) and returns a *partitionString* (Table B.5) that represents the optimal tree. The string is obtained as follows: once the lowest cost is chosen, the current value of *partitionString* is augmented by appending to it the flag corresponding to the lowest cost chosen. Then, the string returned by the recursive call that leads to the minimum cost is also appended to the end of the *partitionString* and the procedure returns both the minimum cost J_0, J_S or J_V and the updated *partitionString*.


$t_k \times s_k \times v_k \times u_k$
$\lfloor t_k/2 \rfloor \times \lfloor s_k/2 \rfloor \times v_k \times u_k$
$(t_k - \lfloor t_k/2 \rfloor) \times (s_k - \lfloor s_k/2 \rfloor) \times v_k \times u_k$
$(t_k - \lfloor t_k/2 \rfloor) \times \lfloor s_k/2 \rfloor \times v_k \times u_k$
$\lfloor t_k/2 \rfloor \times (s_k - \lfloor s_k/2 \rfloor) \times v_k \times u_k$
$\lfloor t_k/2 \rfloor \times (s_k - \lfloor s_k/2 \rfloor) \times \lfloor v_k/2 \rfloor \times \lfloor u_k/2 \rfloor$
$\lfloor t_k/2 \rfloor \times (s_k - \lfloor s_k/2 \rfloor) \times (v_k - \lfloor v_k/2 \rfloor) \times (u_k - \lfloor u_k/2 \rfloor)$
$\lfloor t_k/2 \rfloor \times (s_k - \lfloor s_k/2 \rfloor) \times (v_k - \lfloor v_k/2 \rfloor) \times \lfloor u_k/2 \rfloor$
$\lfloor t_k/2 \rfloor \times (s_k - \lfloor s_k/2 \rfloor) \times \lfloor v_k/2 \rfloor \times (u_k - \lfloor u_k/2 \rfloor)$

Key

○	Node
▧	*spatialSplit* flag
▦	*viewSplit* flag
■	*transform* flag

NOTE The transform flag signals that the node is a leaf node and will be no further partitioned.

Figure B.11 — Hierarchical 4D partitioning using the *viewSplit* flag and the *spatialSplit* flag

After the optimal partition tree is found, the Encode Partition procedure (Table B.6) is called to encode it.

B.2.3 Forward 4D-DCT

B.2.3.1 Level shift

Subsequently, the subblocks are subject to a DCT. However, before processing the forward DCT for a block of source light field samples, if the samples of the component are unsigned, those samples shall be level shifted to a signed representation. if the MSB of Ssiz^i from the LFC marker segment (see Annex B.3.2.6.3) is zero, all samples *x(u,v,s,t)* of the ith component are level shifted by subtracting the same quantity from each sample as follows:

$$x(u,v,s,t) \leftarrow x(u,v,s,t) - 2^{Ssiz^i}$$

B.2.3.2 Forward 4D-DCT function

First all components have to be converted to the same precision (bit-depth). Each colour component *c* is by multiplied by $2 \times \left(BPC - Ssiz^c \right)$ before the forward 4D-DCT.

For a given light field *x(u,v,s,t)* the corresponding 4D-DCT representation *X(i,j,p,q)* is computed by transforming the block along each dimension as follows.

$$X^{(u)}(i,v,s,t)=\sqrt{u_k}\ \alpha(i)\sum_{u=0}^{u_k-1}X(u,v,s,t)\cos\left[\frac{\pi(2u+1)i}{2u_k}\right];\ \begin{matrix} i=0,1,\ldots,\ u_k-1; & v=0,1,\ldots,v_k-1 \\ s=0,1,\ldots,s_k-1; & t=0,1,\ldots,t_k-1\end{matrix}$$

$$X^{(uv)}(i,j,s,t)=\sqrt{v_k}\ \alpha(j)\sum_{v=0}^{v_k-1}X^{(u)}(i,v,s,t)\cos\left[\frac{\pi(2v+1)j}{2v_k}\right];\ \begin{matrix} i=0,1,\ldots,\ u_k-1; & j=0,1,\ldots,v_k-1 \\ s=0,1,\ldots,s_k-1; & t=0,1,\ldots,t_k-1\end{matrix}$$

$$X^{(uvs)}(i,j,p,t)=\sqrt{s_k}\ \alpha(p)\sum_{s=0}^{s_k-1}X^{(uv)}(i,j,s,t)\cos\left[\frac{\pi(2s+1)p}{2s_k}\right];\ \begin{matrix} i=0,1,\ldots,\ u_k-1; & j=0,1,\ldots,v_k-1 \\ p=0,1,\ldots,s_k-1; & t=0,1,\ldots,t_k-1\end{matrix}$$

$$X(i,j,p,q)=\sqrt{t_k}\ \alpha(q)\sum_{t=0}^{t_k-1}X^{(u,v,s)}(i,j,p,t)\cos\left[\frac{\pi(2t+1)q}{2t_k}\right];\ \begin{matrix} i=0,1,\ldots,\ u_k-1; & j=0,1,\ldots,v_k-1 \\ p=0,1,\ldots,s_k-1; & q=0,1,\ldots,t_k-1\end{matrix}$$

where $\alpha(0)=\sqrt{\frac{1}{N}}$ and $\alpha(n)=\sqrt{\frac{2}{N}};n=1,2,\ldots,\ N-1,$, where N is the size of the transform. Output coefficients are represented as 32-bit integers. As indicated earlier, the transform order is arbitrary.

B.2.4 Quantization and entropy encoding.

The quantization and entropy encoding rely on the R-D optimized hexadeca-tree structure, which is constructed based on the procedure listed in Table B.7 and which is further discussed in this paragraph. This tree is uniquely represented by a series of ternary flags: *lowerBitplane*, *splitBlock* and *zeroBlock* (Table B.44). The hexadeca-tree is built by recursively subdividing a 4D block until all sub-blocks reach a 1×1×1×1 4D block-size. The hexadeca-tree is built by recursively subdividing a 4D block. Starting from a 4D block of size t_b×s_b×v_b×u_b, and a bit-plane initially set to maxBitplane (Table B.11), 3 operations are performed:

i) **Lower the bit-plane:** in this case, the descendant of the node is another block with the same dimensions as the original one but represented with precision *bitplane*-1. This is used to indicate for all pixels of the block that the binary representation of their magnitudes at the current bitplane and above are zero. This situation is encoded by the ternary flag value *lowerBitPlane*.

ii) **Split the block**: in this case, the node will have up to 16 children, each one associated to a sub-block with approximately half the length of the original block in all four dimensions. For example, a block *B* of size t_b×s_b×v_b×u_b can be split in the following sub-blocks:

B_{0000} of size $(\lfloor t_b/2\rfloor \times \lfloor s_b/2\rfloor \times \lfloor v_b/2\rfloor \times \lfloor u_b/2\rfloor)$,
B_{0001} of size $(\lfloor t_b/2\rfloor \times \lfloor s_b/2\rfloor \times \lfloor v_b/2\rfloor \times u_b-\lfloor u_b/2\rfloor)$,

B_{0010} of size $(\lfloor t_b/2\rfloor \times \lfloor s_b/2\rfloor \times v_b-\lfloor v_b/2\rfloor \times \lfloor u_b/2\rfloor)$,
B_{0011} of size $(\lfloor t_b/2\rfloor \times \lfloor s_b/2\rfloor \times v_b-\lfloor v_b/2\rfloor \times u_b-\lfloor u_b/2\rfloor)$,

B_{0100} of size $(\lfloor t_b/2\rfloor \times s_b-\lfloor s_b/2\rfloor \times \lfloor v_b/2\rfloor \times \lfloor u_b/2\rfloor)$,
B_{0101} of size $(\lfloor t_b/2\rfloor \times s_b-\lfloor s_b/2\rfloor \times \lfloor v_b/2\rfloor \times u_b-\lfloor u_b/2\rfloor)$,
B_{0110} of size $(\lfloor t_b/2\rfloor \times s_b-\lfloor s_b/2\rfloor \times v_b-\lfloor v_b/2\rfloor \times \lfloor u_b/2\rfloor)$,

B_{0111} of size $(\lfloor t_b/2\rfloor \times s_b-\lfloor s_b/2\rfloor \times v_b-\lfloor v_b/2\rfloor \times u_b-\lfloor u_b/2\rfloor)$,
B_{1000} of size $(t_b-\lfloor t_b/2\rfloor \times \lfloor s_b/2\rfloor \times \lfloor v_b/2\rfloor \times \lfloor u_b/2\rfloor)$,
B_{1001} of size $(t_b-\lfloor t_b/2\rfloor \times \lfloor s_b/2\rfloor \times \lfloor v_b/2\rfloor \times u_b-\lfloor u_b/2\rfloor)$,

B_{1010} of size $(t_b - \lfloor t_b/2 \rfloor \times \lfloor s_b/2 \rfloor \times v_b - \lfloor v_b/2 \rfloor \times \lfloor u_b/2 \rfloor)$,

B_{1011} of size $(t_b - \lfloor t_b/2 \rfloor \times \lfloor s_b/2 \rfloor \times v_b - \lfloor v_b/2 \rfloor \times u_b - \lfloor u_b/2 \rfloor)$,

B_{1100} of size $(t_b - \lfloor t_b/2 \rfloor \times s_b - \lfloor s_b/2 \rfloor \times \lfloor v_b/2 \rfloor \times \lfloor u_b/2 \rfloor)$,

B_{1101} of size $(t_b - \lfloor t_b/2 \rfloor \times s_b - \lfloor s_b/2 \rfloor \times \lfloor v_b/2 \rfloor \times u_b - \lfloor u_b/2 \rfloor)$,

B_{1110} of size $(t_b - \lfloor t_b/2 \rfloor \times s_b - \lfloor s_b/2 \rfloor \times v_b - \lfloor v_b/2 \rfloor \times \lfloor u_b/2 \rfloor)$,

B_{1111} of size $(t_b - \lfloor t_b/2 \rfloor \times s_b - \lfloor s_b/2 \rfloor \times v_b - \lfloor v_b/2 \rfloor \times u_b - \lfloor u_b/2 \rfloor)$.

There are 16 possible sub-blocks, but depending on the size of the parent block, some of these descendant sub-blocks will have one or more of their lengths equal to zero and shall be skipped. All descendants have the same bit-depth as the parent. This situation is indicated by the flag value *splitBlock* (Table B.44).

iii) **Discard the block:** the node has no descendants and is represented by an all-zeros block. This situation is indicated by the flag value *zeroBlock* (Table B.44).

The procedure described in Table B.7 recursively subdivides the input block, as determined by the ternary flags in the segmentation string until a 1×1×1×1 4D block-size is reached.

Given a particular hexadeca-tree, specified by a unique *segmentationString* of ternary segmentation flags together with a particular block, the data can be encoded by means of the recursive procedure described in Table B.8. The inputs to this procedure are the transformed block to be encoded and the optimal partition string. It recursively subdivides the input block, as determined by the ternary flags in the segmentation string. Then the magnitude of the single coefficient of this block is encoded one bit at a time, ranging from the current bit plane to the *minimumBitplane* (Table B.8), using an arithmetic encoder with a different context information for each bit. If the coefficient is not zero valued, its signal is encoded as well.

The rate and the distortion achieved depend heavily on the choice of the segmentation tree as well as the data itself, and those should be matched. The procedure described in Table B.7, recursively chooses the optimal segmentation tree to use in the encoding of a given block in a rate-distortion sense. The optimization works as follows: it starts with *bitplane=maximumBitplane*, *segmentationString=null,* variables J_0 and J_1 both set to infinity and the full transformed input block. The transformed block is scanned and all its coefficients are compared to a threshold given by $2^{bitplane}$. If the magnitudes of all of them are less than the threshold, the optimization procedure is recursively called with the same block as input, but with a *bitplane* value decreased by one (*bitplane*-1). The values returned by this recursive call are the new Lagrangian cost J_0, and a rate-distortion optimized segmentation string *lowerSegmentationString*. However, if any coefficient is above the threshold, the transformed block is segmented into up to 16 sub-blocks as previously described. The optimization procedure is called recursively for each sub-block and the returned Lagrangian costs are added to obtain the new Lagrangian cost J_1. The segmentation strings returned from these calls are concatenated to form the *splitSegmentationString*.

Another Lagrangian cost J_2 is evaluated considering the resulting cost if the block was replaced by a block entirely composed of zeros. The lowest cost is chosen, and the segmentation string is updated as follows:

— If the minimum cost is J_0, the input segmentation string is augmented by appending a flag *lowerBitplane* followed by the *lowerSegmentationString.*

— If the minimum cost is J_1, the input segmentation string is augmented by appending a flag *splitBlock* followed by the *splitSegmentationString*.

— If the minimum cost is J_2, the input segmentation string is augmented by appending a flag *zeroBlock*.

The procedure returns the lowest cost and the resulting associated segmentation string.

The 4D coefficients, flags, and probability context information generated during the encoding process, are input to the arithmetic encoder, that generates the compressed representation of the light field (Table B.4). The adaptive statistical binary arithmetic coding is detailed in Table B.12, Table B.13 and Table B.14. The arithmetic coding requires transmitting only the information needed to allow a decoder to determine the particular fractional interval between 0 and 1 to which the sequence is mapped, adapting to changing statistics in the codestream.

B.2.5 Sample encoding procedure

In this subclause, a sample encoding algorithm is provided for informative purposes. The main procedure processing the individual 4D blocks is listed in Table B.1.

Table B.1 — 4D block scan procedure

`LightField() {`	
`SOC_marker()`	Write SOC marker
`LFC_marker()`	Write LFC marker
`Write SCC_marker() for every colour component not having a global scaling factor equal to 1.`	Write SCC marker for colour component of which the global scaling factor is different from 1
`PNT_marker()`	Write PNT marker
`for(t=0; t<T; t+=BLOCK-SIZE_t ){// scan order on t`	Scan order on t (T defined in Light Field Header box)
`  for(s=0; s<S; s+= BLOCK-SIZE_s){ // scan order on s`	Scan order on s (S defined in Light Field Header box)
`    for(v=0; v<V; v+= BLOCK-SIZE_v){ // scan order on v`	Scan order on v (V defined in Light Field Header box)
`      for(u=0; u<U; u+= BLOCK-SIZE_u){ // scan order  on u`	Scan order on u (U defined in Light Field Header box)
`        for(c=0; c<NC; c++){ // scan order on colour components`	Scan order on c (colour component)
`          SOB_marker();`	Write SOB marker

`        if ( (TRNC) && ((T - BLOCK-SIZE_t) < t < T) ) {`
`            tk = T umod BLOCK_SIZE_t; }`
`        else tk = BLOCK-SIZE_t;`
	Block size computation

`        if ( (TRNC) && ((S - BLOCK-SIZE_s) < s < S) ) {`
`            sk = S umod BLOCK_SIZE_s; }`
`        else sk = BLOCK-SIZE_s;`
	Block size computation

`        if ( (TRNC) && ((V - BLOCK-SIZE_v) < v < V) ) {`
`            vk = V umod BLOCK_SIZE_v; }`
`        else vk = BLOCK-SIZE_v;`
	Block size computation

`        if ( (TRNC) && ((U - BLOCK-SIZE_u) < u < U) ) {`
`            uk = U umod BLOCK_SIZE_u; }`
`        else uk = BLOCK-SIZE_u;`
	Block size computation

Table B.1 *(continued)*

`          Padding(LF.BlockAtPosition(t, s, v, u));`	Fills the pixels outside the light field if needed. If TRNC ==1, there will be no pixel outside the light field, and Padding (Table B.2) will have no effect. Note that the 4D array LF.Block-AtPosition is a local copy of the currently processed 4D block (for colour component c).
`          InitEncoder()`	Initializes the arithmetic coder for each coded codestream (Table B.13)
`          Encode(LF.BlockAtPosition(t, s, v, u), lambda)`	
`        } // end of scan order on colour components loop`	
`      }`	
`    }`	
`  }`	
`}`	

Table B.2 — 4D block padding procedure

`Procedure Padding(block) {`	When TRNC == 0 and a block exceeds a light field dimension, the exceeded block pixels are filled with block values at the light field border
`if (t+tk > T) {`	
`  for(ti = T; ti<t+tk; ++ti) {`	
`    for(si = s; si<s+sk; ++si) {`	
`      for(vi = v; vi<v+vk; ++vi) {`	
`        for(ui = u; ui<u+uk; ++ui) {`	

`          block[ti-t][si-s][vi-v][ui-u]`
`            = block[T-t-1][si-s][vi-v][ui-u]`	

`        }`	
`      }`	
`    }`	
`  }`	
`}`	
`if (s+sk > S) {`	
`  for(ti = t; ti<t+tk; ++ti) {`	
`    for(si = S; si< s+sk; ++si) {`	
`      for(vi = v; vi<v+vk; ++vi) {`	
`        for(ui = u; ui<u+uk; ++ui) {`	

`          block[ti-t][si-s][vi-v][ui-u]`
`            = block[ti-t][S-s-1][vi-v][ui-u]`	

`        }`	
`      }`	
`    }`	

Table B.2 *(continued)*

`  }`	
`}`	
`if (v+vk > V) {`	
`  for(ti = t; ti<t+tk; ++ti) {`	
`    for(si = s; si<s+sk; ++si) {`	
`      for(vi = V; vi<v+vk; ++vi) {`	
`        for(ui = u; ui<u+uk; ++ui) {`	

`          block[ti-t][si-s][vi-v][ui-u]`
`            = block[ti-t][si-s][V-v-1][ui-u]`	

`        }`	
`      }`	
`    }`	
`  }`	
`}`	
`if (u+uk > U) {`	
`  for(ti = t; ti<t+tk; ++ti) {`	
`    for(si = s; si< s+sk; ++si) {`	
`      for(vi = v; vi< v+vk; ++vi) {`	
`        for(ui = U; ui< u+uk; ++ui) {`	

`          block[ti-t][si-s][vi-v][ui-u]`
`            = block[ti-t][si-s][vi-v][U-u-1]`	

`        }`	
`      }`	
`    }`	
`  }`	
`}`	
`}`	

Table B.3 — 4D-DCT block coefficient component scaling

`Procedure ScaleBlock(block){`	Scaling of the 4D-DCT coefficients components by Spscc (see B.3.2.6.3)
`for(ti = 0; ti<tk; ++ti) {`	
`  for(si = 0; si< sk; ++si) {`	
`    for(vi = 0; vi< vk; ++vi) {`	
`      for(ui = 0; ui< uk; ++ui) {`	

`        block[ti][si][vi][ui] = ⌈Spscc[c] × block[ti][si]`
`[vi][ui]⌉;`
	The array Spscc contains the global scaling factors for each colour component.

`      }`	
`    }`	
`  }`	
`}`	

Table B.4 — 4D block encoding procedure

Procedure Encode(block, lambda) {	
OptimizePartition(partitionString, block, lambda);	Defined in Table B.5
EncodeMinimumBitPlane();	Defined in Table B.10
EncodePartition(partitionString, block);	Defined in Table B.6
FlushEncoder();	Defined in Table B.14
}	

Table B.5 — 4D block partition optimization procedure

Procedure OptimizePartition(block, lambda) {	Finds the optimal 4D-block partition
blockDCT = 4DDCT(block);	Transformed block (using Annex C.6.3.2 definitions)
ScaleBlock(blockDCT);	Scaling of the 4D-DCT coefficients components by Spscc (see Table B.3)
if((tb,sb,vb,ub) == (tk,sk,vk,uk)){	If the block size is equal to the maximum block size
EvaluateOptimalbitPlane(MinimumBitPlane, block, lambda);	Defined in Table B.11
set MinimumBitPlane to the optimal value found;	
set partititionString = "";	
}	
segmentationString = "";	
J0 = OptimizeHexadecaTree(block, maximumBitPlane, lambda, segmentationString);	Defined in Table B.7
JS = infinity;	
if((vb > 1)&&(ub > 1)) {	If spatial dimensions (ub, vb) of the block are greater than the predefined minimum then segments the block into 4 (four) nonoverlapping sub-blocks (spatialSplit flag – Figure B.7, Figure B.8 and Table B.30)
v'b = floor(vb/2);	
u'b = floor(ub/2);	
partitionStringS = "";	
JS = OptimizePartition(Block.GetSubblock(block, 0,0,0,0,tb,sb,v'b,u'b), lambda);	Points to the sub-block position; Returns the Spatial Lagrangian R-D cost
JS += OptimizePartition(Block.GetSubblock(block, 0,0,0,u'b,tb,sb,v'b,ub-u'b), lambda);	Points to the sub-block position; Returns the Spatial Lagrangian R-D cost
JS += OptimizePartition(Block.GetSubblock(block, 0,0,v'b,0,tb,sb,vb-v'b,u'b), lambda);	Points to the sub-block position; Returns the Spatial Lagrangian R-D cost
JS += OptimizePartition(Block.GetSubblock(block, 0,0,v'b,u'b,tb,sb,vb-v'b,ub-u'b),lambda);	Points to the sub-block position; Returns the Spatial Lagrangian R-D cost
}	
JV = infinity;	

Table B.5 *(continued)*

`  if((tb > 1)&&(sb > 1)) {`	If view dimensions (tb, sb) of the block are greater than the predefined minimum then segment the block into 4 (four) nonoverlapping sub-blocks (viewSplit flag – Figure B.9, Figure B.10 and Table B.30)
`    t'b = floor(tb/2);`	
`    s'b = floor(tb/2);`	
`    partitionStringV = "";`	
`    JV  = OptimizePartition(Block.GetSubblock(block, 0,0,0,0,t'b,s'b,vb,ub), lambda);`	Points to the sub-block position; Returns the View Lagrangian R-D cost
`    JV += OptimizePartition(Block.GetSubblock(block, 0,s'b,0,0,t'b,sb-s'b,vb,ub), lambda);`	Points to the sub-block position; Returns the View Lagrangian R-D cost
`    JV += OptimizePartition(Block.GetSubblock(block, t'b,0,0,0,tb-t'b,s'b,vb,ub), lambda);`	Points to the sub-block position; Returns the View Lagrangian R-D cost
`    JV += OptimizePartition(Block.GetSubblock(block, t'b,s'b,0,0,tb-t'b,sb-s'b,vb,ub), lambda);`	Points to the sub-block position; Returns the View Lagrangian R-D cost
`  }`	
`  if((J0 < JS)&&(J0 < JV)) {`	Returns: transform flag (Figure B.11, Table B.30)
`    partitionString = cat(partitionString, transformFlag);`	
`    return partitionString, J0;`	Returns the Lagrangian cost of transforming the block and the transform flag (Figure B.11, Table B.30)
`  }`	
`  if((JS < J0)&&(JS < JV)) {`	
`    partitionString = cat(partitionString, spatialSplitFlag);`	
`    return partitionString, JS;`	Returns the Lagrangian cost of the spatial segmentation and the spatialSplit flag (Figure B.11, Table B.30)
`  }`	
`  if((JV < JS)&&(JV < J0)) {`	
`    partitionString = cat(partitionString, viewSplitFlag);`	
`    return partitionString, JV;`	Returns the Lagrangian cost of the view segmentation and the viewSplit flag (Figure B.11, Table B.30)
`  }`	
`}`	

Table B.6 — 4D block encode partition procedure

Procedure EncodePartition(block, lambda) {	Encodes a 4D block
blockDCT = 4DDCT(block);	Defined in Annex B.2.3.2
ScaleBlock(blockDCT);	Scaling of the 4D-DCT coefficients components by Spscc (see Annex B.3.2.6.3 and Table B.3)
if((tb,sb,vb,ub) == (tk,sk,vk,uk)) {	If the block size is equal to the maximum block size
point to the start of the partitionString;	
}	
get the partitionFlag at the current position of the partitionString;	
advance the pointer to the partitionString by one position;	
if(partitionFlag == 0) EncodeBit(0,0);	Transmits the partitionFlag;
if(partitionFlag == 1){	
EncodeBit(1,0);	
EncodeBit(0,0);	
}	
if(partitionFlag == 2){	
EncodeBit(1,0);	
EncodeBit(1,0);	
}	
OptimizeHexadecaTree(block, lambda, maxBitPlane);	Defined in Table B.7
set the segmentationStringPointer to the start of the segmentationString;	
EncodeHexadecaTree(block, maxBitPlane);	Defined in Table B.8
}	
if(partitionFlag == spatialSplitFlag) {	
v'b = floor(vb/2);	
u'b = floor(ub/2);	
EncodePartition(Block.GetSubblock(block, 0,0,0,0,tb,sb,v'b,u'b));	
EncodePartition(Block.GetSubblock(block, 0,0,0,u'b,tb,sb,v'b,ub-u'b));	
EncodePartition(Block.GetSubblock(block, 0,0,v'b,0,tb,sb,vb-v'b,u'b));	
EncodePartition(Block.GetSubblock(block, 0,0,v'b,u'b,tb,sb,vb-v'b,ub-u'b));	
}	
if(partitionFlag == viewSplitFlag) {	
t'b = floor(tb/2);	
s'b = floor(tb/2);	
EncodePartition(Block.GetSubblock(block, 0,0,0,0,t'b,s'b,vb,ub));	
EncodePartition(Block.GetSubblock(block,	

Table B.6 *(continued)*

`    0,s'b,0,0,t'b,sb-s'b,vb,ub));`	

`  EncodePartition(Block.GetSubblock(block,`
`    t'b,0,0,0,tb-t'b,s'b,vb,ub));`	

`  EncodePartition(Block.GetSubblock(block,`
`    t'b,s'b,0,0,tb-t'b,sb-s'b,vb,ub));`	

`  }`	
`  return;`	
`}`	

Table B.7 — 4D block hexadeca-tree optimization procedure

`Procedure OptimizeHexadecaTree(block, bitplane, lambda,segmentationString) {`	Recursive Hexadeca-tree optimization procedure
`  if(bitplane < InferiorBitPlane) {`	

`    return the sum of the squared values of the coefficients`
`      of the block;`
	Energy of the block

`  }`	
`  if(the block is o size (tb,sb,vb,ub) == (1,1,1,1)) {`	If the block size is 1x1x1x1 then:

`    estimate the rate R to encode the remaining bits of the`
`      coefficient, from bitplane down to MinimumBitPlane;`
	estimate the rate (R) to encode the coefficient

`    evaluate the distortion D, as the squared error between the`
`      coefficient represented with minimumBitPlane precision`
`      and the full precision coefficient;`
	evaluate the distortion D;

`    return J = D + lambda × r`	return the Rate-Distortion cost
`  }`	
`  J0 = infinity;`	
`  J1 = infinity;`	

`  if (the magnitude of any coefficient of the block`
`    is less than 1 << bitplane) {`
	If the magnitude of all elements of the block are smaller than 2bitplane

`    lowerSegmentationString = "";`	The lower segmentation string is null

`    J0 = OptimizeHexadecaTree(block, bitplane-1, lambda,`
`          lowerSegmentationString);`	

`  }`	
`  else {`	Subdivides the block in at most 16 non-overlapping sub-blocks t'b, s'b, v'b and u'b (Figure B.11) by splitting in half at each dimension whenever the length at that dimension is greater than one
`    t'b = floor(tb/2);`	
`    s'b = floor(sb/2);`	
`    v'b = floor(vb/2);`	
`    u'b = floor(ub/2);`	
`  }`	
`  nseg_t = nseg_s = nseg_v = nseg_u = 1;`	

Table B.7 *(continued)*

`  if(t'b > 1) nseg_t++;`	
`  if(s'b > 1) nseg_s++;`	
`  if(v'b > 1) nseg_v++;`	
`  if(u'b > 1) nseg_u++;`	
`  splitSegmentationString = " ";`	The split segmentation string is null
`  J1 = 0;`	
`  for(t = 0; t < nseg_t; t++) {`	
`    for(s = 0; s < nseg_s; s++) {`	
`      for(v = 0; v < nseg_v; v++) {`	
`        for(u = 0; u < nseg_u; u++) {`	
`          new_t = t×t'b + (1-t)×(tb-t'b);`	
`          new_s = s×s'b + (1-s)×(sb-s'b);`	
`          new_v = v×v'b + (1-v)×(vb-v'b);`	
`          new_u = u×u'b + (1-u)×(ub-u'b);`	
`          get subBlock from block.`	The subBlock size is (new_t, new_s, new_v, new_u) the position is (t×t'b,s×s'b,v×v'b,u×u'b)
`          J1 += OptimizeHexadecaTree(subBlock, bitplane, lambda, splitSegmentationString);`	
`        }`	
`      }`	
`    }`	
`  }`	
`  J2 = 0;`	
`  J2 = sum of the squared values of the coefficients of the block + + lambda × rate to encode flag zeroBlock flag;`	Energy of the block plus the Lagrangian multiplier times the rate to encode the zeroBlock flag (Table B.30).
`  J0 += lambda × rate to encode the lowerBitPlane flag`	The cost to encode the lowerBitplane Flag (Table B.30).
`  J1 += lambda × rate to encode the splitBlock flag`	The cost to encode the splitBlock flag (Table B.30).
`  if(J0 < J1 && J0 < J2) {`	
`    segmentationString = cat(segmentationString, lowerBitplane, lowerSegmentationString);`	
`    return J0, segmentationString`	Returns the Lagrangian cost and the optimal segmentation string
`  }`	
`  if(J1 < J0 && J0 < J2) {`	
`    segmentationString = cat(segmentationString, splitBlock, splitSegmentationString);`	
`  return J1, segmentationString`	Returns the Lagrangian cost and the optimal segmentation string
`  }`	
`  if(J2 < J0 && J2 < J1) {`	
`    segmentationString = cat(segmentationString, zeroFlag);`	

Table B.7 *(continued)*

`    return J2, segmentationString`	Returns the Lagrangian cost and the optimal segmentation string
`  }`	
`}`	

Table B.8 — 4D block hexadeca-tree encoding procedure

`Procedure EncodeHexadecaTree(block, bitplane) {`	Encodes the resulting blocks from the hexadeca-tree structure and associated flags.
`  if(bitplane < InferiorBitPlane) return`	Below the lowest level
`  if(the block is of size (tb,sb,vb,ub) == (1,1,1,1)) {`	
`    EncodeCoefficient(block, bitplane);`	Defined in Table B.9
`  }`	
`  get the segmentationFlag at the current position of the segmentationString;`	
`  advance the pointer to the segmentationString by one position;`	
`  EncodeBit((segmentationFlag>>1) & 01, 33 + 2×bitplane);`	Transmits the segmentationFlag (Table B.12)
`  UpdateModel((segmentationFlag>>1) & 01, 33 + 2×bitplane);`	Defined in Table B.42
`  if((segmentationFlag>>1) & 01 == 0){`	
`    EncodeBit((segmentationFlag & 01), 34 + 2×bitplane);`	Defined in Table B.12
`    UpdateModel(segmentationFlag & 01), 34 + 2×bitplane);`	Defined in Table B.42
`  }`	
`  if(segmentationFlag == zeroBlock) return;`	
`  if (segmentationFlag == lowerBitPlane) {`	Lowers the bit-plane
`    EncodeHexadecaTree(block, bitplane-1);`	
`  }`	
`  if (segmentationFlag == splitBlock) {`	
`    t'b = floor(tb/2);`	
`    s'b = floor(sb/2);`	
`    v'b = floor(vb/2);`	
`    u'b = floor(ub/2);`	
`    nseg_t = nseg_s = nseg_v = nseg_u = 1;`	Number of segments in each 4D dimension
`    if(tb > 1) nseg_t++;`	
`    if(sb > 1) nseg_s++;`	
`    if(vb > 1) nseg_v++;`	
`    if(ub > 1) nseg_u++;`	
`    for(t = 0; t < nseg_t; t++) {`	
`      for(s = 0; s < nsg_s; s++) {`	
`        for(v = 0; v < nseg_v; v++) {`	
`          for(u = 0; u < nseg_u; u++) {`	

Table B.8 *(continued)*

`new_t = t×t'b + (1-t)×(tb-t'b);`	
`new_s = s×s'b + (1-s)×(sb-s'b);`	
`new_v = v×v'b + (1-v)×(vb-v'b);`	
`new_u = u×u'b + (1-u)×(ub-u'b);`	
`get subBlock from block at position (t×t'b,s×s'b,v×v'b,u×u'b)`	The subBlock size is (new_t, new_s, new_v, new_u) the position is (t×t'b,s×s'b,v×v'b,u×u'b)
`EncodeHexadecaTree(subBlock, bitplane);`	
`}`	
`}`	
`}`	
`}`	
`}`	
`}`	

Table B.9 — 4D block hexadeca-tree encode coefficient

`Procedure EncodeCoefficient(coefficient, bitplane){`	Uses the arithmetic encoder to encode the coefficient bit using the value of the bit-plane as context.
`Magnitude = |Coefficient|;`	
`for(bitplane_counter=bitplane ; bitplane_counter>=MinimumBitPlane; bitplane_counter--) {`	
`CoefficientBit = (Magnitude >> bitplane_counter-MinimumBitPlane) & 01H;`	
`EncodeBit(CoefficientBit,bitplane_counter);`	Transmits CoefficientBit (Table B.12)
`}`	
`if (Magnitude > 0) {`	
`if(Coefficient > 0) EncodeBit(0,0);`	Transmits signal (transmits '0') (Table B.12
`else EncodeBit(1,0);`	Transmits signal (transmits '1') (Table B.12)
`}`	
`}`	

Table B.10 — 4D-block hexadeca-tree encode minimum bit-plane

`Procedure EncodeMinimumBitPlane() {`	Encodes the 8-bit unsigned representation of the maximum number of bit-planes minus 1.
`for(bitplane_counter=7 ; bitplane_counter>=0; bitplane_counter--) {`	
`Bit = (MinimumBitPlane >> bitplane_counter) & 01H;`	
`EncodeBit(Bit,0);`	Transmits Bit (Table B.12)
`}`	

Table B.10 *(continued)*

}	

Table B.11 — 4D block optimal bitplane procedure

Procedure EvaluateOptimalbitPlane(MinimumBitPlane, block, lambda) {	Returns the minimal bit-plane
Jmin = infinity;	
MinimumBitPlane = MaximumBitPlane;	
AcumulatedRate = 0.0;	
for(bitplane = MaximumBitPlane; bitplane >= 0; bitplane--) {	
Distortion = 0.0	
for all pixels in block {	
AcumulatedRate += rate to encode bit at current bitplane for the current pixel;	
Distortion += distortion incurred by encoding the current pixel with bitplane precision	
J = Distortion + lambda × AcumulatedRate;	
if(J <= Jmin) {	
MinimumBitPlane = bitplane;	
Jmin = J;	
}	
}	
}	
return(MinimumBitPlane);	
}	

Table B.12 — Encode bit procedure

Procedure EncodeBit(inputbit, modelIndex) {	Encodes the bit that composes the codestream (for modelIndex see Table B.31)
threshold = floor(((tag - inferiorLimit + 1) × acumFreq_1[modelIndex])/(inferiorLimit - superiorLimit + 1))	
length = floor(((superiorLimit - inferiorLimit + 1) × acumFreq_0[modelIndex])/acumFreq_1[modelIndex])	
if(inputbit == 0) superiorLimit = inferiorLimit + length - 1;	
else inferiorLimit = inferiorLimit + length;	
while((MSB of inferiorLimit == MSB of superiorLimit) || ((inferiorLimit >= 4000H) and (superiorLimit < C000H)) {	
if(MSB of inferiorLimit == MSB of superiorLimit) {	
bit = MSB of inferiorLimit;	
transmits bit	
inferiorLimit = inferiorLimit << 1;	Shifts a zero into the LSB
superiorLimit = superiorLimit << 1;	Shifts a zero into the LSB
superiorLimit = superiorLimit+1	
while(ScalingsCounter > 0) {	
ScalingsCounter = ScalingsCounter - 1;	
transmits (1-bit)	

Table B.12 *(continued)*

`      }`	
`    }`	
`    if(inferiorLimit >= 4000H) and (superiorLimit < C000H ) {`	
`      inferiorLimit = inferiorLimit << 1;`	Shifts a zero into the LSB
`      superiorLimit = superiorLimit << 1;`	Shifts a zero into the LSB
`      superiorLimit = superiorLimit+1;`	
`      inferiorLimit = inferiorLimit ^ 8000H;`	
`      superiorLimit = superiorLimit ^ 8000H;`	
`      ScalingConter = ScalingCounter + 1;`	
`    }`	
`  }`	
`}`	

Table B.13 — Init encoder procedure

`Procedure InitEncoder() {`	Initializes the Arithmetic Encoder
`  inferiorLimit = 0;`	Inferior limit of the interval length
`  superiorLimit = FFFFH;`	Superior limit of the interval length
`  ScalingsCounter = 0;`	
`}`	

Table B.14 — Flush encoder procedure

`Procedure FlushEncoder() {`	When the encoding is complete, the bits in the buffer must be moved to output codestream before a terminating marker is generated.
`  mScalingsCounter++;`	
`  if(inferiorLimit >= 4000H)`	
`    bit = 1;`	
`  else`	
`    bit = 0;`	
`  transmit bit;`	
`  while(ScalingsCounter > 0) {`	
`    transmits (1-bit);`	
`    ScalingsCounter--;`	
`  }`	
`}`	

B.3 4D transform mode decoding

B.3.1 General

In this subclause, the decoding procedure of the codestreams for light field data encoded with the 4D Transform mode is specified. The codestream is signalled as payload of the Contiguous Codestream box defined in Annex A.3.4. Figure B.12 illustrates the decoder architecture.

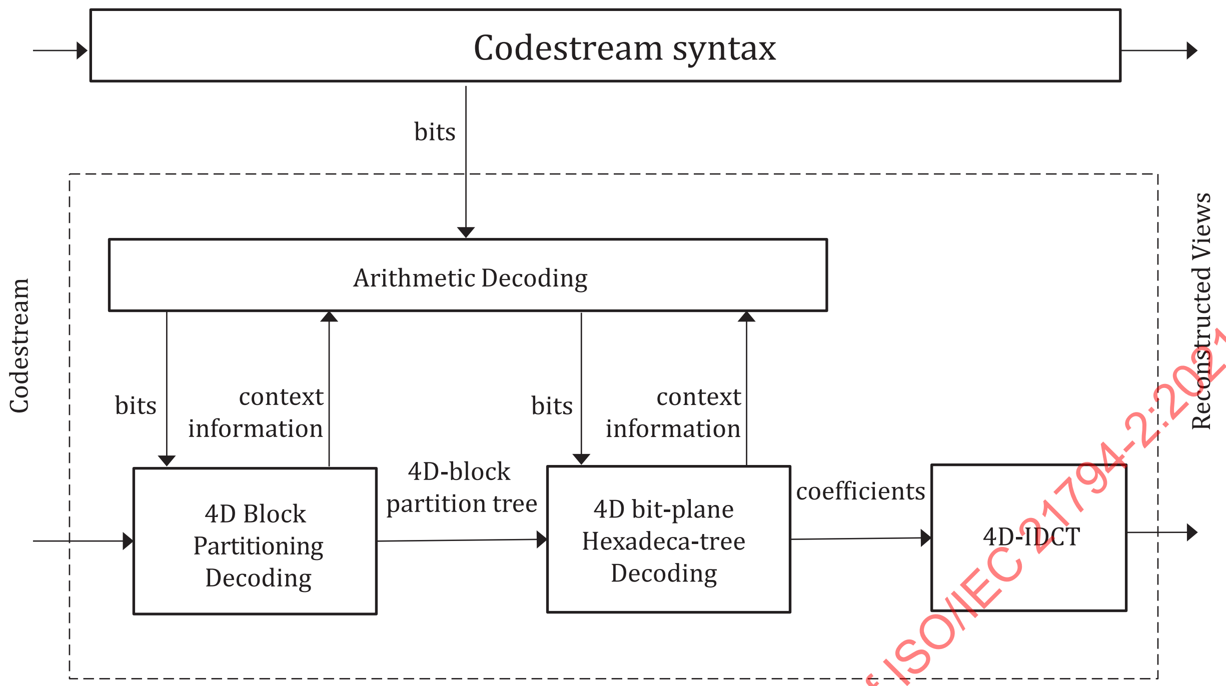


Figure B.12 — 4D transform mode light field decoder architecture

B.3.2 Codestream syntax

B.3.2.1 General

This section specifies the marker and marker segment syntax and semantics defined by this document. These markers and marker segments provide codestream information for this document. Further, this subclause provides a marker and marker segment syntax that is designed to be used in future specifications that include this document as a normative reference.

This document does not include a definition of conformance. The parameter values of the syntax described in this annex are not intended to portray the capabilities required to be compliant.

B.3.2.2 Markers, marker segments, and headers

This document uses markers and marker segments to delimit and signal the characteristics of the source image and codestream. This set of markers and marker segments is the minimal information needed to achieve the features of this document and is not a file format. A minimal file format is offered in Annexes A and B.

Main header is collections of markers and marker segments. The main header is found at the beginning of the codestream.

Every marker is two bytes long. The first byte consists of a single 0xFF byte. The second byte denotes the specific marker and can have any value in the range 0x01 to 0xFE. Many of these markers are already used in ITU-T Rec. T.81 | ISO/IEC 10918-1, ITU-T Rec. T.84 | ISO/IEC 10918-3, ITU-T Rec. T.800 | ISO/IEC 15444-1 and ITU-T Rec. T.801 | ISO/IEC 15444-2 and shall be regarded as reserved unless specifically used.

A marker segment includes a marker and associated parameters, called marker segment parameters. In every marker segment the first two bytes after the marker shall be an unsigned value that denotes the length in bytes of the marker segment parameters (including the two bytes of this length parameter but not the two bytes of the marker itself). When a marker segment that is not specified in the document appears in a codestream, the decoder shall use the length parameter to discard the marker segment.

B.3.2.3 Key to graphical descriptions

Each marker segment is described in terms of its function, usage, and length. The function describes the information contained in the marker segment. The usage describes the logical location and frequency of this marker segment in the codestream. The length describes which parameters determine the length of the marker segment.

These descriptions are followed by a figure that shows the order and relationship of the parameters in the marker segment. Figure B.13 shows an example of this type of figure. The marker segments are designated by the three-letter code of the marker associated with the marker segment. The parameter symbols have capital letter designations followed by the marker's symbol in lower-case letters. A rectangle is used to indicate a parameter's location in the marker segment. The width of the rectangle is proportional to the number of bytes of the parameter. A shaded rectangle (diagonal stripes) indicates that the parameter is of varying size. Two parameters with superscripts and a grey area between indicate a run of several of these parameters.

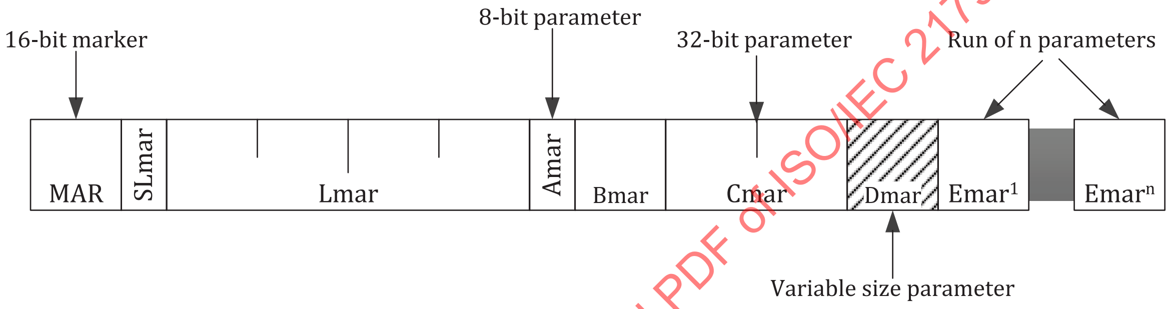


Figure B.13 — Example of the marker segment description figures

The figure is followed by a list that describes the meaning of each parameter in the marker segment. If parameters are repeated, the length and nature of the run of parameters is defined. As an example, in Figure B.13, the first rectangle represents the marker with the symbol MAR. The second rectangle represents the size of the length parameter SLmar (Table B.15). The third rectangle represents the length parameter Lmar. Parameters Amar, Bmar, Cmar, and Dmar are 8-, 16-, 32-bit and variable length respectively. The notation Emar^i implies that there are *n* different parameters, Emar^i, in a row.

After the list is a table that either describes the allowed parameter values or provides references to other tables that describe these values. Tables for individual parameters are provided to describe any parameter without a simple numerical value. In some cases, these parameters are described by a bit value in a bit field. In this case, an "x" is used to denote bits that are not included in the specification of the parameter or sub-parameter in the corresponding row of the table.

Table B.15 — Size parameters for the SLlfc, SLscc and SLpnt

Value (bits) MSB LSB	Parameter size
xxxx xx00	Length parameter is 16 bits.
xxxx xx01	Length parameter is 32 bits.
xxxx xx10	Length parameter is 64 bits.
	All other values reserved

B.3.2.4 Defined marker segments

Table B.16 lists the markers specified in this document.

Table B.16 — List of defined marker segments

	Symbol	Code	Main Header	4D Block Header
Start of codestream	SOC	0xFFA0	Required	Not allowed
Light field Configuration	LFC	0xFFA1	Required	Not allowed
Colour component scaling	SCC	0xFFA2	Optional	Optional
Codestream pointer set	PNT	0xFFA3	Optional	Not allowed
Start of block	SOB	0xFFA4	Not allowed	Required
End of codestream	EOC	0xFFD9	Not allowed	Not allowed

B.3.2.5 Construction of the codestream

Figure B.14 shows the construction of the codestream. All of the solid lines show required marker segments. The following markers and marker segments are required to be in a specific location: SOC, LFC, PNT, SOB and EOC. The dashed lines show optional or possibly not required marker segments.

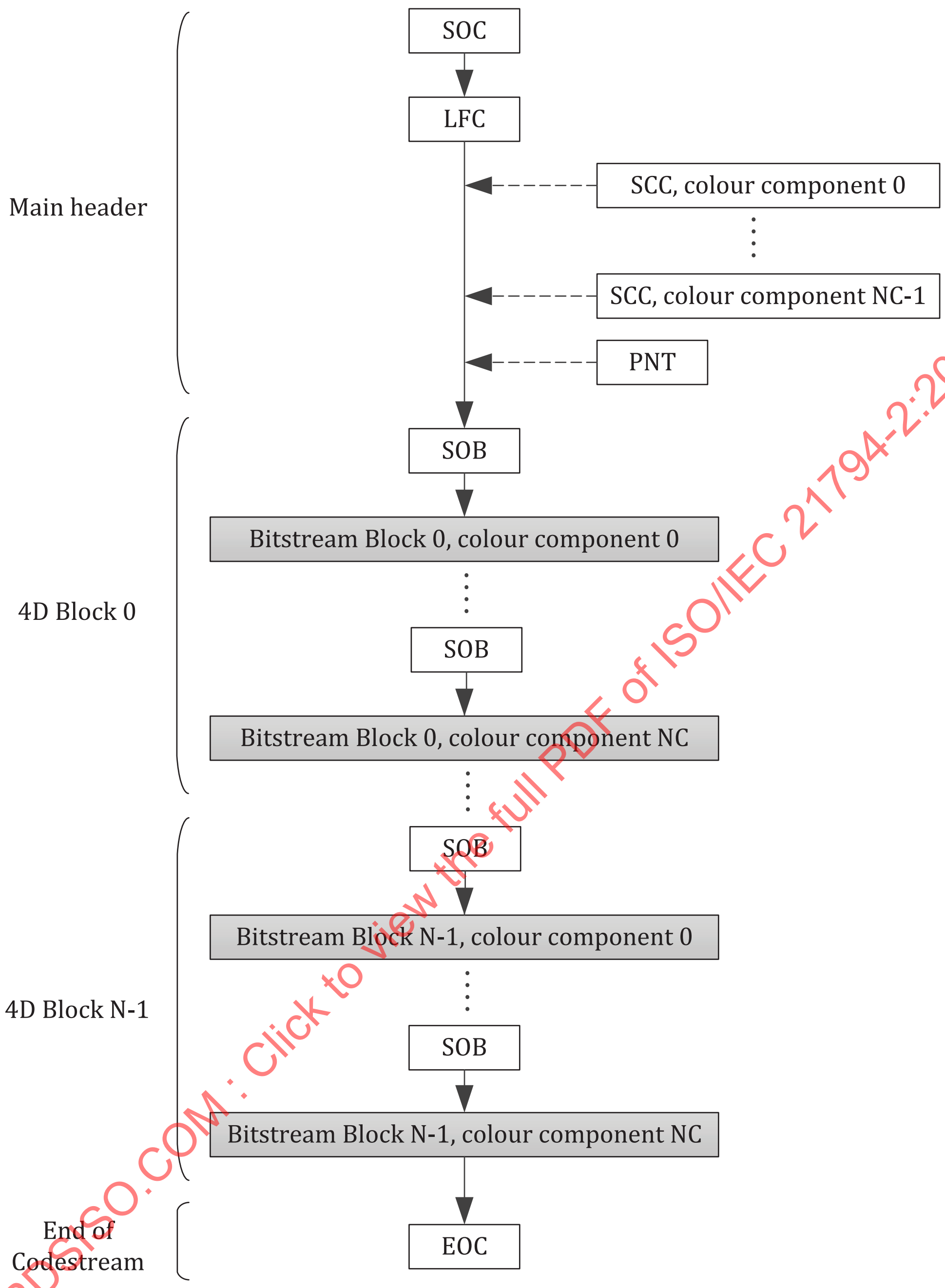

Main header
SOC
LFC
SCC, colour component 0
SCC, colour component NC-1
PNT
4D Block 0
SOB
Bitstream Block 0, colour component 0
SOB
Bitstream Block 0, colour component NC
4D Block N-1
SOB
Bitstream Block N-1, colour component 0
SOB
Bitstream Block N-1, colour component NC
End of Codestream
EOC

Figure B.14 — Codestream structure

B.3.2.6 Delimiting markers and marker segments

B.3.2.6.1 General

The delimiting marker and marker segments shall be present in all codestreams conforming to this document. Each codestream has only one SOC marker, one EOC marker, and contains at least one 4D block. Each 4D block has one SOB and one EOB marker. The SOC, SOB, and EOC are delimiting markers, not marker segments, and have no explicit length information or other parameters.

B.3.2.6.2 Start of codestream (SOC)

Function: Marks the beginning of a codestream specified in this document (Table B.17).

Usage: Main header. This is the first marker in the codestream. There shall be only one SOC per codestream.

Length: Fixed.

SOC marker code

Table B.17 — Start of codestream parameter values

Field name	Size (bits)	Value
SOC (Start of Codestream)	16	0xFFA0

B.3.2.6.3 Light field configuration (LFC)

Function: Provides information about the uncompressed light field such as the width and height of the subaperture views, number of subaperture views in rows and columns, number of components, component bit depth, number of 4D blocks and size of the 4D blocks, component bit depth for transform coefficients (Figure B.15):

— **Llfc:** The value of this parameter is determined as $Llfc = 40 + 2 \cdot NC$.

— **ROWS (T):** The value of this parameter indicates the number of rows of the subaperture view array. This field is stored as a 4-byte big-endian unsigned integer.

— **COLUMNS (S):** The value of this parameter indicates the number of columns of the subaperture view array. This field is stored as a 4-byte big-endian unsigned integer.

— **HEIGHT (V):** The value of this parameter indicates the height of the sample grid. This field is stored as a 4-byte big-endian unsigned integer.

— **WIDTH (U):** The value of this parameter indicates the width of the sample grid. This field is stored as a 4-byte big-endian unsigned integer.

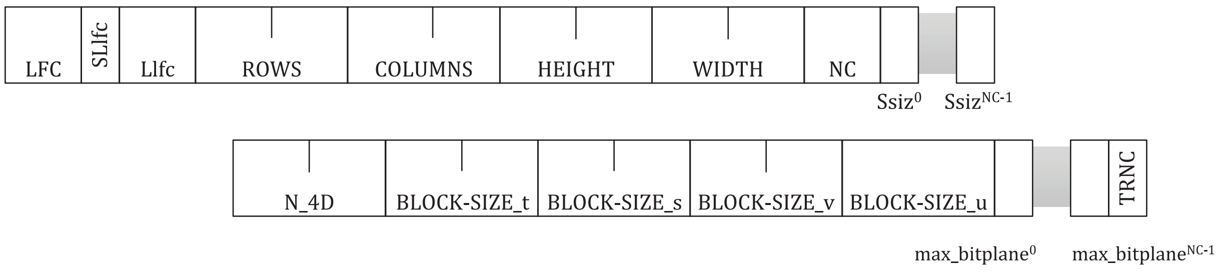
— **NC:** This parameter specifies the number of components in the codestream and is stored as a 2-byte big-endian unsigned integer. The value of this field shall be equal to the value of the NC field in the Light Field Header box. If no Channel Definition Box is available, the order of the components for colour images is R-G-B-Aux or Y-U-V-Aux.

— **Ssizi:** The precision is the precision of the component samples before DC level shifting is performed (i.e. the precision of the original component samples before any processing is performed). If the component sample values are signed, then the range of component sample values is −2(Ssiz+1 AND 0x7F)−1 ≤ component sample value ≤ 2(Ssiz+1 AND 0x7F)−1 − 1. There is one occurrence of this parameter for each component. The order corresponds to the component's index, starting with zero.

— **TRNC:** If unset (TRNC=0), all 4D blocks will have initially the same fixed-sizes and a padding procedure is applied.

Usage: Main header. There shall be one and only one in the main header immediately after the SOC marker segment. There shall be only one LFC per codestream.

Length: Fixed.



Key

LFC	marker code
SLlfc	size of Llfc parameter
Llfc	length of marker segment in bytes (not including the marker)
ROWS (*T*)	number of rows of the subaperture view array
COLUMNS (*S*)	number of columns of the subaperture view array
HEIGHT (*V*)	subaperture view height
WIDTH (*U*)	subaperture view width
NC	number of (colour) components
Ssiz^i	precision (depth) in bits and sign of the ith component samples
N_4D	number of 4D blocks in which the light field is segmented
BLOCK-SIZE_t	size of the 4D block in the t direction – number of rows of the view array
BLOCK-SIZE_s	size of the 4D block in the s direction – number of columns of the view array
BLOCK-SIZE_v	size of the 4D block in the v direction – height of the views
BLOCK-SIZE_u	size of the 4D block in the u direction – width of the views
max_bitplane^i	precision (depth) in bits of the ith component for the transform coefficients
TRNC	flag indicating that 4D blocks with dimensions spanning the full available light field dimensions have their sizes truncated to fit in the light field dimensions
LFC	marker code
SLlfc	size of Llfc parameter

Figure B.15 — Light field configuration syntax

Table B.18 — Format of the contents of configuration parameter set for the 4D coding

Field name	Size (bits)	Value
LFC	16	0xFFA1
SLlfc	8	0
Llfc	16	42 to 32808
ROWS	32	1 to (2^{32}– 1)
COLUMNS	32	1 to (2^{32}– 1)
HEIGHT	32	1 to (2^{32}– 1)
WIDTH	32	1 to (2^{32}– 1)
NC	16	1 to 16 384
Ssiz^i	8	See Table B.19
N_4D	32	1 to (2^{32}– 1)
BLOCK-SIZE_t	32	1 to (2^{32}– 1)
BLOCK-SIZE_s	32	1 to (2^{32}– 1)
BLOCK-SIZE_v	32	1 to (2^{32}– 1)
BLOCK-SIZE_u	32	1 to (2^{32}– 1)

Table B.18 *(continued)*

Field name	Size (bits)	Value
max_bitplane^i	8	Number less than or equal to (Ssiz^i AND 0x7F)+$\log_2$ (BLOCK-SIZE_t)+$\log_2$ (BLOCK-SIZE_s) +$\log_2$ (BLOCK-SIZE_v)+$\log_2$ (BLOCK-SIZE_u)
TRNC	8	0 or 1 All other values reserved for ISO/IEC use

Table B.19 — Component Ssiz parameter

Value (bits) MSB LSB	Component sample precision
x000 0000 to x010 0101	Component sample bit depth = value + 1. From 1 bit deep to 38 bits deep respectively (counting the sign bit, if appropriate).
0xxx xxxx	Component sample values are unsigned values.
1xxx xxxx	Component sample values are signed values.
	All other values reserved for ISO/IEC use.

B.3.2.6.4 Colour component scaling (SCC)

Function: Describes the global scaling factor used for a colour component (Table B.20). If this marker segment is not signalled for a specific colour component, the value of the global scaling factor for this colour component shall be 1.

Usage: Main header. No more than one per any given component may be present. Optional.

Length: Variable depending on the number of colour components.

SCC	marker code Table B.20 shows the size and values of the symbol and parameters for the quantization component marker segment.
SLscc	size of Lscc parameter
Lscc	length of marker segment in bytes (not including the marker) Lscc = 6 for NC < 257; Lscc = 8 for NC ≥ 257
Cscc	index of the component to which this marker segment relates The components are indexed 0, 1, 2, etc. (either 8 or 16 bits depending on NC value defined in the LFC marker segment).
Spscc	Scaling factor used for the colour component Cscc (see Table B.21)

$$Spscc = 2^{Exponent-16} \times Mantissa$$

Table B.20 — Colour component scale parameter values

Field name	Size (bits)	Value
SCC	16	0xFFA2
SLscc	8	0
Lscc	16	6 or 8
Cscc	8 16	0 to 255; if NC < 257 0 to 16383; if NC ≥ 257
Spscc	16	Table B.21

Table B.21 — Quantization values for the Spscc parameter

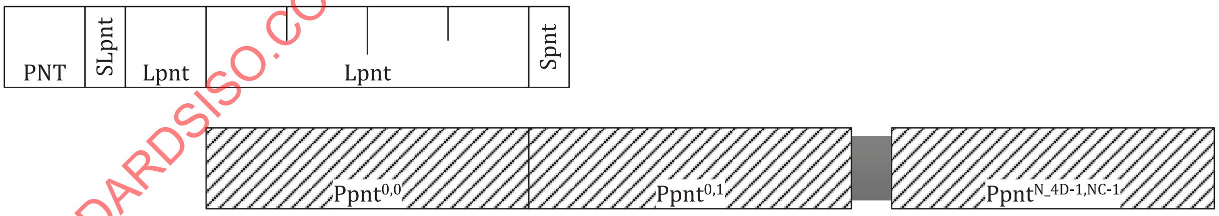
Value (bits)		Scaling factor values
MSB	LSB	
xxxx x000 0000 0000 to xxxx x111 1111 1111		mantissa of scaling factor value
0000 0xxx xxxx xxxx to 1111 1xxx xxxx xxxx		exponent of scaling factor value

B.3.2.6.5 Codestream pointer set (PNT)

Function: Provides pointers to the codestream associated each 4D block to facilitate efficient access (Table B.22).

Usage: Optional. If present, must be included in the main header between the LFC marker segment and the first SOB marker segment defined in this document. Only one PNT marker segment shall be embedded in the main header (Figure B.16).

Length: Variable.



Key

PNT	marker code
SLpnt	size of Lpnt parameter
Lpnt	length of marker segment in bytes (not including the marker)
Spnt	size of the PPnt parameter
$PPnt^{i,c}$	pointer to codestream of 4D block n and colour component c

NOTE 1 The value of the **Lpnt** parameter is determined as follows:

$$Lpnt = \begin{cases} 9+4\cdot N\_4D & \text{Spnt=0} \\ 9+8\cdot N\_4D & \text{Spnt=1} \end{cases}$$

where N_4D is defined in the LFC marker segment.

NOTE 2 The **PPnt**^{i,c} pointer indicates the position of the addressed 4D block codestream – its SOB marker – in the Contiguous Codestream box counting from the beginning of this box, i.e. the LBox field.

Figure B.16 — Codestream pointer set syntax

Table B.22 — Format of the contents of the codestream pointer set for the 4D coding

Field name	Size (bits)	Value
PNT	16	0xFFA3
SLpnt	8	2
Lpnt	64	$13 \text{ to } 9+8\cdot\left(2^{32}-1\right)$
Spnt	8	Table B.23
PPnt^i	32 if Spnt = 0 64 if Spnt = 1	1 to (2^{32}– 1) 1 to (2^{64}– 1)

Table B.23 — Size parameters for Spnt

Value (bits)		Parameter size
MSB	LSB	
xxxx xxx0		PPnt parameters are 32bits.
xxxx xxx1		PPnt parameter are 64bits.
		All other values reserved.

B.3.2.6.6 Start of block (SOB)

Function: Marks the beginning of a 4D block (Table B.24).

Usage: Every 4D block header. Shall be the first marker segment in a 4D Block header. There shall be at least one SOB in a codestream. There shall be only one SOB per 4D block.

Length: Fixed.

SOB marker code

Table B.24 — Format of the contents of the start of block

Field name	Size (bits)	Value
SOB	8	0xFFA4

B.3.2.6.7 End of codestream (EOC)

Function: Indicates the end of the codestream (Table B.25).

NOTE 1 This marker shares the same code as the EOI marker in ITU-T Rec. T.81 | ISO/IEC 10918-1 and the EOC marker in ITU-T Rec. T.800 | ISO/IEC 15444-1.

Usage: Shall be the last marker in a codestream. There shall be one EOC per codestream.

NOTE 2 In the case a file has been corrupted, it is possible that a decoder could extract much useful compressed image data without encountering an EOC marker.

Length: Fixed.

EOC marker code

Table B.25 — Format of the contents of end of codestream

Field name	Size (bits)	Value
EOC	8	0xFFD9

B.3.3 Codestream parsing

The procedure to parse and decode a light field codestream as contained by the Contiguous Codestream box (Annex A.3.4), with dimensions (max_t, max_s, max_v, max_u) defined as ROW, COLUMN, HEIGHT and WIDTH in Figure B.15 (Table B.18), with block dimensions (t_k, s_k, v_k, u_k) defined as BLOCK-SIZE_t, BLOCK-SIZE_s, BLOCK-SIZE_v and BLOCK-SIZE_u in Figure B.15 (Table B.18), with a maximum number of bit-planes of max_bitplane (defined in Figure B.15 (Table B.18)), is described in the pseudo-code (all variables are integers). The scan order is in the order *t, s, v, u*, as described in Table B.26.

The coordinate set (*t, s, v, u*) refers to the left, superior corner of the 4D block. The procedure "ResetArithmeticDecoder()" resets all the context model probabilities of the arithmetic decoder. The procedure "LocateContiguousCodestream" reads the pointer corresponding to position (*t,s,v,u*) and delivers the respective codestream to procedure "DecodeContiguous Codestream()". The procedure "DecodeContiguousCodestream()" decodes the components of the block contained in the codestream enabling the sequential decoding of light field. The codestreams of the components are decoded sequentially.

Table B.26 — JPEG Pleno (JPL) codestream structure (4D transform mode)

Defined syntax	**Simplified structure**
`LightField() {`	
`  SOC_marker()`	Codestream_Header()
`  LFC_marker()`	
`  Read all SCC_marker()`	
`  PNT_marker()`	
`  for(t=0; t<T; t+=BLOCK-SIZE_t ){// scan order on t`	Codestream_Body()
`    for(s=0; s<S; s+= BLOCK-SIZE_s){ // scan order on s`	
`      for(v=0; v<V; v+= BLOCK-SIZE_v){ // scan order on v`	
`        for(u=0; u<U; u+= BLOCK-SIZE_u){ // scan order on u`	
`          for(c=0; c<NC-1; c++){ // scan order on colour components`	
`            // Initializes the arithmetic decoder for each decoded 4D block codestream`	
`            ResetArithmeticDecoder();`	
`            // Finds the corresponding 4D block codestream for the desired position on the light field`	
`            LocateContiguousCodestream (t, s, v, u);`	
`            SOB_marker()`	
`           // Decodes contiguous 4D block codestream found by the previous procedure`	
`            if ( (TRNC) && ((T – BLOCK-SIZE_t) < t < T) ) { tk = T umod BLOCK-SIZE_t; } else tk = BLOCK-SIZE_t;`	

Table B.26 *(continued)*

`            if ( (TRNC) && ((S - BLOCK-SIZE_s) < s < S) ) {`
`                sk = S umod BLOCK-SIZE_s; }`
`             else sk = BLOCK-SIZE_s;`	

`            if ( (TRNC) && ((V - BLOCK-SIZE_v) < v < V) ) {`
`               vk = V umod BLOCK-SIZE_v; }`
`            else vk = BLOCK-SIZE_v;`	

`            if ( (TRNC) && ((U - BLOCK-SIZE_u) < u < U) ) {`
`               uk = U umod BLOCK-SIZE_u; }`
`            else uk = BLOCK-SIZE_u;`	

`            LF.BlockAtPosition(t, s, v, u) =`
`            DecodeContiguousCodestream(tk, sk, vk, uk);`	

`          } // end of scan order on colour components loop`	
`        } // end of scan order on u loop`	
`      } // end of scan order on v loop`	
`    } // end of scan order on s loop`	
`  } // end of scan order on t loop`	
`  EOC_marker()`	Codestream_End()
`}`	

B.3.4 4D partitioning general structure

B.3.4.1 General

The datasets (the all texture views in Figure B.1) are composed by 4D light fields of dimensions *t×s×v×u*. The views are addressed by the *s,t* coordinates pair, while the *u,v* pair addresses a pixel within each *s,t* view, as pictured in Figure B.6.

B.3.4.2 Partition tree decoding

The root node of the tree corresponds to a full *t×s×v×u* transform. The partition tree is represented by a series of ternary flags:

— A *spatialSplit* flag indicates that a $t_k \times s_k \times v_k \times u_k$ block is segmented into a set of four sub-blocks {spatialSubblock0, spatialSubblock1, spatialSubblock2 and spatialSubblock3}, of dimensions $t_k \times s_k \times \lfloor v_k/2 \rfloor \times \lfloor u_k/2 \rfloor$, $t_k \times s_k \times (v_k - \lfloor v_k/2 \rfloor) \times \lfloor u_k/2 \rfloor$, $t_k \times s_k \times \lfloor v_k/2 \rfloor \times (u_k - \lfloor u_k/2 \rfloor)$ and $t_k \times s_k \times (v_k - \lfloor v_k/2 \rfloor) \times (u_k - \lfloor u_k/2 \rfloor)$ respectively, where $D/2$ indicates the largest integer smallest than or equal to $D/2$. Figure B.7, Figure B.8 and Figure B.17 illustrate the results when applying the *spatialSplit* flag;

— A *viewSplit* flag indicates that a $t_k \times s_k \times v_k \times u_k$ block is segmented into a set of four sub-blocks {viewSubblock0, viewSubblock1, viewSubblock2 and viewSubblock3}, of dimensions $\lfloor t_k/2 \rfloor \times \lfloor s_k/2 \rfloor \times v_k \times u_k$, $(t_k - \lfloor t_k/2 \rfloor) \times \lfloor s_k/2 \rfloor \times v_k \times u_k$, $\lfloor t_k/2 \rfloor \times (s_k - \lfloor s_k/2 \rfloor) \times v_k \times u_k$ and $t_k \times s_k \times (v_k - \lfloor v_k/2 \rfloor) \times (u_k - \lfloor u_k/2 \rfloor)$, and. $(t_k - \lfloor t_k/2 \rfloor) \times (s_k - \lfloor s_k/2 \rfloor) \times v_k \times u_k$. Figure B.9, Figure B.10 and Figure B.18 illustrate the results when applying the *viewSplit* flag;

— The partition tree has its leaf nodes marked by a *transform* flag. Each sub-block that is not a leaf is recursively decoded in this fashion, from sub-block 0 to 3 of the sub-block set, and the decoded flags of the sub-trees of each sub-block are concatenated in this order.

NOTE The transform flag signals that the node is a leaf node and will be no further partitioned.

— Figure B.11 shows six nodes marked by a *transform* flag, two nodes split (segmented) accordingly the *spatialSplit* and the *viewSplit* flags, and the 9th node, $(\lfloor t_k/2 \rfloor \times (s_k - \lfloor s_k/2 \rfloor) \times \lfloor v_k/2 \rfloor \times \lfloor u_k/2 \rfloor$, which can be further segmented using either the *spatialSplit* flag or the *viewSplit* flag.

Figure B.17 depicts the result of a 4D block with dimensions of 9×9×434×625 ($t_k \times s_k \times v_k \times u_k$) partitioned, using the *spatialSplit* flag, into a sub-block with dimensions of 9×9×217×312, in grey, ($t_k \times s_k \times (v_k - \lfloor v_k/2 \rfloor) \times \lfloor u_k/2 \rfloor$). Please note that, in Figure B.17 the partitioned v and u dimensions are: $217 = 434 - \lfloor 434/2 \rfloor$ ($v_k - \lfloor v_k/2 \rfloor$) and $312 = \lfloor 625/2 \rfloor$ ($\lfloor u_k/2 \rfloor$).

Figure B.18 depicts the result of a 4D block with dimensions of 9×9×434×625 ($t_k \times s_k \times v_k \times u_k$) partitioned, using the *viewSplit* flag, into a sub-block with dimensions of 5×4×434×625, in grey, ($(t_k - \lfloor t_k/2 \rfloor) \times \lfloor s_k/2 \rfloor \times v_k \times u_k$). Please note that, the partitioned t and s dimensions are: $5 = 9 - \lfloor 9/2 \rfloor$ ($t_k - \lfloor t_k/2 \rfloor$) and $4 = \lfloor 9/2 \rfloor$ ($\lfloor s_k/2 \rfloor$).

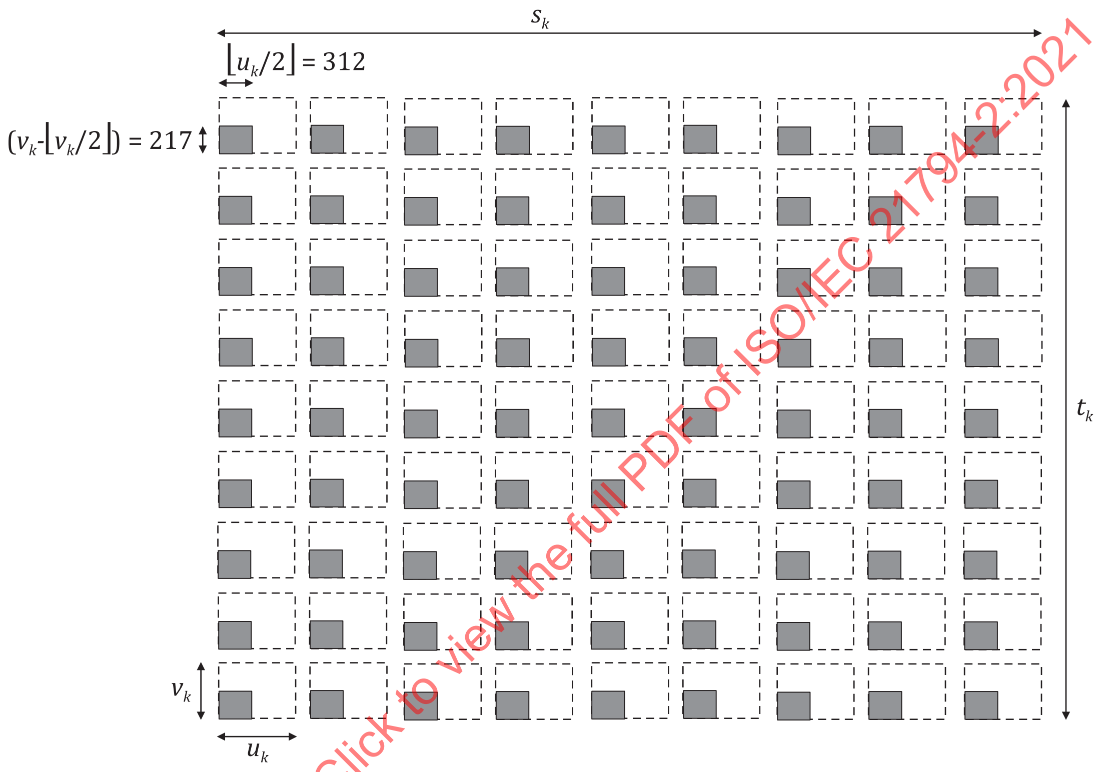

s_k
$\lfloor u_k/2 \rfloor = 312$
$(v_k - \lfloor v_k/2 \rfloor) = 217$
t_k
v_k
u_k

Figure B.17 — Example of a 4D block with dimensions $t_k \times s_k \times (v_k - \lfloor v_k/2 \rfloor) \times \lfloor u_k/2 \rfloor$ (in grey) superimposed to a 4D block with dimensions $t_k \times s_k \times v_k \times u_k$

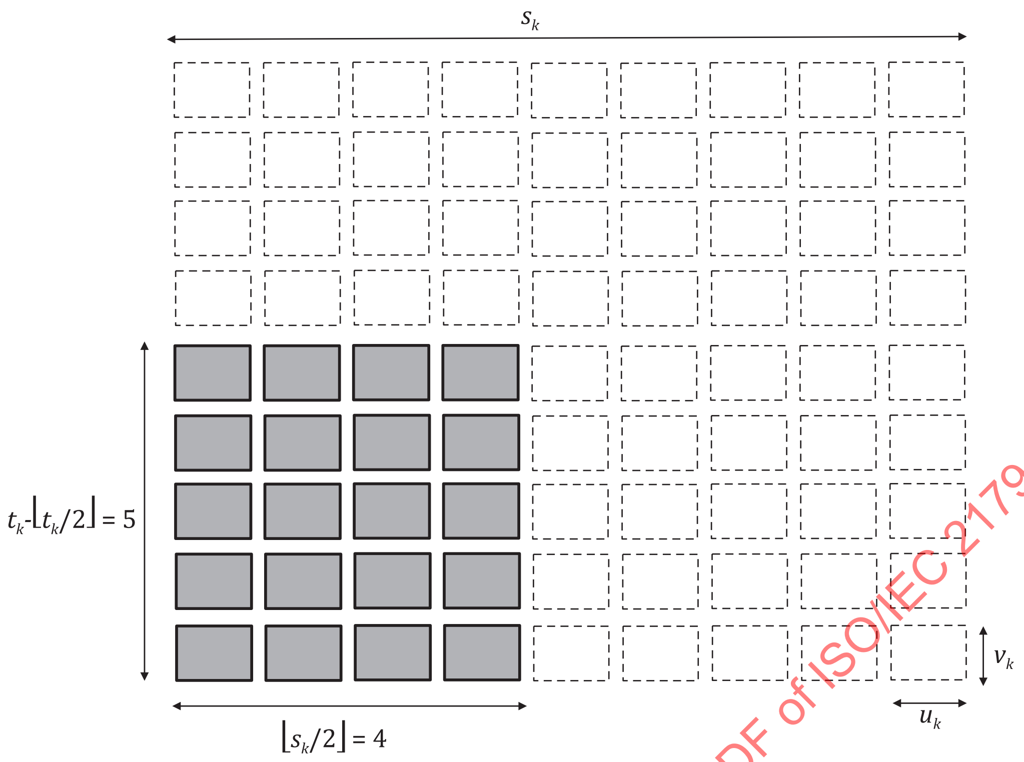

s_k
t_k
t_k-$\lfloor t_k/2 \rfloor = 5$
v_k
u_k
$\lfloor s_k/2 \rfloor = 4$

Figure B.18 — Example of a 4D block with dimensions $(t_k - \lfloor t_k/2 \rfloor) \times \lfloor s_k/2 \rfloor \times v_k \times u_k$ (in grey) superimposed on a 4D block with dimensions $t_k \times s_k \times v_k \times u_k$

B.3.4.3 Contiguous codestream of a 4D-block

For a contiguous codestream of a 4D-block with size tk×sk×vk×uk, a 4D-block partitioning decoding procedure is performed (DecodeContiguousCodestream), that uses the recursive procedure "Procedure DecodePartitionStep", both defined in Table B.27 and Table B.28.

Table B.27 — Decode contiguous codestream procedure

`Procedure DecodeContiguousCodestream(tk, sk, vk, uk){`	Decodes a 4D block of size (tk, sk, vk, uk)
`  ReadMinimumBitPlane;`	Reads an 8 bits integer that represents the lower bitplane of transform coefficient, as defined in Table B.34
`  Block=DecodePartitionStep(0,0,0,0,tk,sk,vk,uk);`	Defined in Table B.29
`  Return Block;`	
`}`	

Table B.28 — 4D-DCT block coefficient component inverse scaling

`Procedure InverseScaleBlock(block){`	Inverse scaling of the 4D-DCT coefficients components by Spscc of colour c (see B.3.2.6.3)
`  for(ti = 0; ti<tk; ++ti) {`	
`    for(si = 0; si< sk; ++si) {`	
`      for(vi = 0; vi< vk; ++vi) {`	
`        for(ui = 0; ui< uk; ++ui) {`	

Table B.28 *(continued)*

`block[ti][si][vi][ui] = block[ti][si][vi][ui] / Spscc[c];`	
`}`	
`}`	
`}`	
`}`	
`}`	

Table B.29 — Decode partition step procedure

`Procedure DecodePartitionStep(tpp, spp, vpp, upp, tk, sk, vk, uk){`	Recursively decodes the 4D block and the partition flags of a contiguous codestream
`ReadPartitionTreeFlag;`	Reads flag from arithmetic decoder – defined in Table B.37
`if (flag == transform) {`	Reached the Leaf Node (Transform flag; Figure B.11, Table B.30)
`Block=DecodeBlock(max_bitplane);`	Recursively decodes the DCT coefficients (Table B.43)
`InverseScaleBlock(Block);`	Inverse scaling of the 4D-DCT coefficients components by Spscc (see B.3.2.6.3)
`Blockdecoded=4D-IDCT(Block);`	Performs the inverse DCT of the decoded coefficients (Annex B.3.7)
`return Blockdecoded;`	
`}`	
`If (flag == spatialSplit){`	Figure B.7, Figure B.8, Figure B.17 and Table B.30
`Int new_tp, new_sp, new_vp, new_up, new_ tk, new_sk, new_vk, new_uk;`	
`new_tp = tpp;`	
`new_sp = spp;`	
`new_vp = vpp;`	
`new_up = upp;`	
`new_tk = tk;`	
`new_sk = sk;`	
`new_vk = floor(vk/2);`	
`new_uk = floor(uk/2);`	
`DecodePartitionStep(new_tp, new_sp, new_vp, new_up, new_tk, new_sk, new_vk, new_uk);`	
`new_up = upp +floor(uk/2);`	
`new_uk = uk – floor(uk/2);`	
`DecodePartitionStep(new_tp, new_sp, new_vp, new_up, new_tk, new_sk, new_vk, new_uk);`	
`new_vp = vpp +floor(vk/2);`	
`new_vk = vk – floor(vk/2);`	
`DecodePartitionStep(new_tp, new_sp, new_vp, new_up, new_tk, new_sk, new_vk, new_uk);`	
`new_up = upp ;`	
`new_uk = floor(uk/2);`	

Table B.29 *(continued)*

DecodePartitionStep(new_tp, new_sp, new_vp, new_up, new_tk, new_sk, new_vk, new_uk);	
};	
if (flag == viewSplit) {	Figure B.9, Figure B.10, Figure B.18 and Table B.30
Int new_tp, new_sp, new_vp, new_up, new_ tk, new_sk, new_vk, new_uk;	
new_tp = tpp;	
new_sp = spp;	
new_vp = vpp;	
new_up = upp;	
new_tk = floor(tk/2);	
new_sk = floor(sk/2);	
new_vk = vk;	
new_uk = uk;	
DecodePartitionStep(new_tp, new_sp, new_vp, new_up, new_tk, new_sk, new_vk, new_uk);	
new_sp = spp + floor(sk/2);	
new_sk = sk – floor(sk/2);	
DecodePartitionStep(new_tp, new_sp, new_vp, new_up, new_tk, new_sk, new_vk, new_uk);	
new_tp = tpp + floor(vk/2);	
new_tk = tk – floor(tk/2);	
DecodePartitionStep(new_tp, new_sp, new_vp, new_up, new_tk, new_sk, new_vk, new_uk);	
new_sp = spp ;	
new_sk = floor(sk/2);	
DecodePartitionStep(new_tp, new_sp, new_vp, new_up, new_tk, new_sk, new_vk, new_uk);	
}	
}	

Table B.30 — Lists of partition flags representations

Partition Flag	Representation
transform	0
spatialSplit	1
viewSplit	2

B.3.5 Arithmetic decoding procedure

B.3.5.1 General

Figure B.19 shows a simple block diagram of a binary adaptive arithmetic decoder. The compressed light field data cd and a context cx from the decoder's model unit (not shown) are input to the arithmetic decoder. The arithmetic decoder's output is the decision d. The encoder and decoder model units need to supply exactly the same context cx for each given decision. The decoding process should be initialized. The contexts (cx) and bytes of compressed light field data (as needed) are read and passed on to the decoder until all contexts have been read. The decoding process computes the binary decision d and returns a value of either 0 or 1. The estimation procedures for the probability models, which provide adaptive estimates of the probability for each context are part of the decoding process.

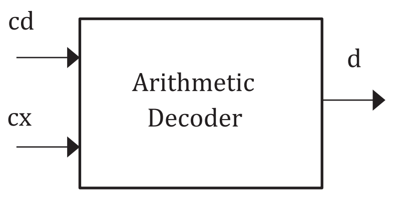


Figure B.19 — Arithmetic decoder inputs and output

B.3.5.2 Probability models

The contexts of the arithmetic coder are defined in Table B.31. When the adaptive flag is Off the probability model is fixed.

Table B.31 — Arithmetic coder contexts

Symbol	Context range	Adaptive Flag
DCT coefficient sign	0	Off
DCT coefficients bits	1-32	On
Hexadecatree flags	33-98	On
Partition flags	0	Off

B.3.5.3 Procedures to read symbols

Each call to the arithmetic decoder to read a variable is performed according to the following syntax:

```
symbol=read(Context, AdaptiveFlag);
```

Each call to read a symbol from stream, may update arithmetic decoder models if the adaptive flag is On. The procedure read(Context, AdaptiveFlag) is defined in Table B.32 and the variable *Context* is defined in Table B.31.

Table B.32 — Read bit using arithmetic decoder procedure

`Procedure Read(Context, AdaptiveFlag){`	Context defined in Table B.31
`  Bit =DecodeBit(Context);`	Defined in Table B.41
`  If (AdaptiveFlag == On) {`	
`    If Bit == 0 {`	
`      UpdateModel(0, Context);`	Defined in Table B.42
`    }`	
`    Else {`	
`      UpdateModel(1,Context);`	Defined in Table B.42
`    }`	
`  }`	
`  return Bit;`	
`}`	

Table B.33 — Read magnitude procedure

`Procedure ReadMagnitude(bitplane, MinimumBitPlane)`	Reads the magnitude of a DCT coefficient
`  Magnitude = 0;`	

`  for(bitplane_counter=bitplane ; bitplane_counter>=MinimumBitPlane;`
`    bitplane_counter--){`
	MinimumBitPlane denotes the lowest bit-plane used for encoding[a]

`    Magnitude = Magnitude << 1;`	
`    CoefficientBit =Read(bitplane_counter + 1, On);`	Decodes a coefficient bit
`    Magnitude +=  CoefficientBit;`	
`  }`	
`  Magnitude = Magnitude << MinimumBitPlane;`	
`  if (Magnitude > 0) {`	
`    Magnitude += (1 << MinimumBitPlane)/2;`	
`  }`	
`  return Magnitude;`	
`}`	

[a] Its value is encoded as an 8-bit unsigned integer and should be less than or equal to the variable max_bit-plane defined in Figure B.15. It must be read just before a DecodePartitionStep procedure, using the procedure "ReadMinimumBitPlane", defined in Table B.34.

Table B.34 — Read minimum bitplane procedure

`Procedure ReadMinimumBitPlane(){`	MinimumBitPlane denotes the lowest bit-plane used for encoding
`  MinimumBitPlane = 0;`	
`  for(counter=0 ; counter<8; counter++){`	
`    MinimumBitPlane = MinimumBitPlane << 1;`	
`    Bit =Read(0, Off);`	Defined in Table B.32
`    MinimumBitPlane +=  Bit;`	
`  }`	
`  return MinimumBitPlane;`	
`}`	

If a DCT coefficient is different from zero its sign must be read from the stream with the procedure in Table B.35.

Table B.35 — Read sign procedure

`Procedure ReadSign(){`	Reads DCT coefficient sign if ≠ 0
`  sign = read(0, Off)`	
`  return sign`	
`}`	

Every ternary hexadecatree flag shall be read using the procedure in Table B.36:

Table B.36 — Read hexadeca-tree flag procedure

`Procedure ReadHexadecatreeFlag(bitplane){`	Reads ternary hexadecatree flag: — lowerBitPlane — splitBlock — zeroBlock (Table B.44)
`  bit1 = read(33 + 2×bitplane, On);`	
`  if(bit1 == 0)`	
`    bit0 = read(34 + 2×bitplane, On);`	
`  flag = bit0+2×bit1;`	
`  return flag;`	
`}`	

Every ternary partition flag must be read using the procedure in Table B.37.

Table B.37 — Read partition tree flag procedure

`Procedure ReadPartitionTreeFlag(){`	Reads ternary partition flag: — spatialSplit; — viewSplit; — transform (Table B.30)
`  flag = read(0, Off);`	
`  if(flag ==1){`	
`    bit = read(0, Off);`	
`    if(bit == 1)`	
`      flag++`	
`  }`	
`  return flag`	
`}`	

B.3.5.4 Arithmetic decoder procedures and definitions

The arithmetic decoder employs probabilistic models to decode symbols from the codestream.

Definitions and procedures to reset the arithmetic decoder, to update probabilistic models and read information from the stream are detailed in this section.

The procedures use two vectors (acumFreq_0 and acumFreq_1), defined with length MAX_NUMBER_OF_MODELS.

The codec uses 99 models numbered from 0 to 98 (i.e. MAX_NUMBER_OF_MODELS=98).

At initialization the decoder limits are set and first bits are read from the stream as specified in the procedure in Table B.38.

Table B.38 — Initialize decoder procedure

`Procedure InitDecoder(){`	Arithmetic decoder initialization
`  inferiorLimit = 0;`	
`  superiorLimit = FFFFH1;`	
`  t = Read16bitsFromStream();`	
`  tag = 0;}`	
`  for(n=0; n<16; n++){`	Reads 16 bits From Stream

Table B.38 *(continued)*

`    tag = tag << 1;`	
`    bit = Read1BitFromStream();`	Reads one byte at a time, LSB first
`    tag = tag + bit`	Append bits to tag
`  }`	
`}`	

A probabilistic model is initialized executing the procedure in Table B.39.

Table B.39 — Initialize probabilistic model procedure

`Procedure InitProbabilisticModel(modelIndex)`	Probabilistic model initialization
`  acumFreq_0[modelIndex] = 1;`	
`  acumFreq_1[modelIndex] = 2;`	
`}`	

The arithmetic decoder is reset, for every model, by calling the “InitDecoder” procedure (Table B.38), followed by the “InitProbabilistModel” (Table B.39 and Table B.40).

Table B.40 — Reset arithmetic decoder procedure

`Procedure ResetArithmeticDecoder()`	
`  InitDecoder();`	Defined in Table B.38
`  for (counter=0; counter<99;counter++)`	
`    InitProbabilisticModel(counter);`	
`}`	

The procedure to decode a bit from the stream is described in Table B.41.

Table B.41 — Decode bit procedure

`bit = DecodeBit(modelIndex){`	Decodes a bit form the code-stream (modelIndex defined in Table B.31)

`  threshold = floor(((tag - inferiorLimit + 1) ×`
`    acumFreq_1[modelIndex]-1)/( superiorLimit -`
`inferiorLimit + 1));`	

`  length = floor(((superiorLimit - inferiorLimit + 1) ×`
`    acumFreq_0[modelIndex])/acumFreq_1[modelIndex]);`	

`  if(threshold < acumFreq_0[modelIndex]) {`	
`    bitDecoded = 0;`	
`    superiorLimit = inferiorLimit + length-1;`	
`  }`	
`  else {`	
`    bitDecoded = 1;`	
`    inferiorLimit = inferiorLimit + length;`	
`  }`	

`  while((MSB of inferiorLimit == MSB of superiorLimit) ||`
`    ((inferiorLimit >= 4000H) and (superiorLimit < C000H ))) {`	

`    if(MSB of inferiorLimit == MSB of superiorLimit) {`	
`      inferiorLimit = inferiorLimit << 1;`	Shifts a zero into the LSB
`      superiorLimit = superiorLimit << 1;`	Shifts a zero into the LSB
`      superiorLimit = superiorLimit+1`	

Table B.41 *(continued)*

`tag = tag << 1;`	Shifts a zero into the LSB
`bit = Read1bitFromStream();`	Reads one byte at a time, LSB first
`tag += bit;`	
`inferiorLimit = inferiorLimit & FFFFH;`	
`superiorLimit = superiorLimit & FFFFH;`	
`t = t & FFFFH;`	
`}`	
`if((inferiorLimit >= 4000H) && (superiorLimit < C000H )) {`	
`inferiorLimit = inferiorLimit << 1;`	Shifts a zero into the LSB
`superiorLimit = superiorLimit << 1;`	Shifts a zero into the LSB
`superiorLimit = superiorLimit+1;`	
`tag = tag<< 1; //Shifts a zero into the LSB`	Shifts a zero into the LSB
`bit = Read1bitFromStream();`	
`tag+= bit;`	
`inferiorLimit = inferiorLimit ^ 8000H;`	
`superiorLimit = superiorLimit ^ 8000H;`	
`tag = tag ^ 8000H;`	
`inferiorLimit = inferiorLimit & FFFFH;`	
`superiorLimit = superiorLimit & FFFFH;`	
`tag = tag & FFFFH;`	
`}`	
`}`	
`inferiorLimit = inferiorLimit & FFFFH;`	
`superiorLimit = superiorLimit & FFFFH;`	
`tag = tag & FFFFH;`	
`return bitDecoded;`	
`}`	

The procedure to update the statistic model is described in Table B.42.

Table B.42 — Update model procedure

`Procedure UpdateModel(bit, modelIndex){`	Probabilistic model update (modelIndex defined in Table B.31)
`if(bit == 0) {`	
`acumFreq_0[modelIndex]++;`	
`acumFreq_1[modelIndex]++;`	
`}`	
`else {`	
`acumFreq_1[modelIndex]++;`	
`}`	
`if(acumFreq_1[modelIndex] == 4095){`	
`acumFreq_1[modelIndex] = acumFreq_1[modelIndex]/2;`	
`acumFreq_0[modelIndex] = acumFreq_0[modelIndex]/2;`	
`if(acumFreq_0[modelIndex] == 0) {`	
`acumFreq_0[modelIndex]++;`	
`acumFreq_1[modelIndex]++;`	
`}`	
`}`	

B.3.6 4D bit-plane hexadeca-tree decoding

The MinimumBitPlane value indicates the minimum depth that the hexadeca-tree decoder will descend for the current block.

For each Block SB_k, of size $t_k \times s_k \times v_k \times u_k$, corresponding to the leaves of the partition tree, a hexadeca-tree decoding procedure is performed. It is described by the following recursive procedure:

Table B.43 — Decode block procedure

`SB = DecodeBlock(bitPlane) {`	Decodes the 4D blocks from the hexadeca-tree structure
`  if (bitPlane < MinimumBitPlane){`	
`    return SB = zero`	Returns Block = 0
`  }`	
`  if (SB is of size 1x1x1x1){`	
`    M = ReadMagnitude(bitplane, MinimumBitPlane);`	Reads a bitPlane-MinimumBitPlane precision positive integer M from input (Table B.33)
`    if (M > 0){`	
`      ReadSign;`	Reads a sign bit (Table B.35)
`      if(sign bit == 1) M = -M`	
`    }`	
`    return SB = M`	
`  }`	
`  ReadHexadecatreeFlag(bitplane);`	Reads hexadeca-tree flag (Table B.36)
`  if (flag == "zeroBlock") return SB = zero`	
`  if (flag == "lowerBitPlane"){`	
`    SB = DecodeBlock(bitPlane-1)`	
`    return SB`	
`  }`	
`  if (flag == "splitBlock"){`	
`    t'b = floor(tb/2);`	tb,sb,vb,ub are the dimensions of the original Block (SB)
`    s'b = floor(sb/2);`	
`    v'b = floor(vb/2);`	
`    u'b = floor(ub/2);`	
`    nseg_t = nseg_s = nseg_v = nseg_u = 1;`	
`    if(tb > 1) nseg_t++;`	
`    if(sb > 1) nseg_s++;`	
`    if(vb > 1) nseg_v++;`	
`    if(ub > 1) nseg_u++;`	
`    for(t = 0; t < nseg_t; t++) {`	
`      for(s = 0; s < nseg_s; s++) {`	
`        for(v = 0; v < nseg_v; v++) {`	
`          for(u = 0; u < nseg_u; u++) {`	
`            new_t = t×t'b + (1-t)×(tb-t'b);`	
`            new_s = s×s'b + (1-s)×(sb-s'b);`	
`            new_v = v×v'b + (1-v)×(vb-v'b);`	
`            new_u = u×u'b + (1-u)×(ub-u'b);`	

Table B.43 *(continued)*

`            SSB_t,s,v,u = DecodeBlock(subBlock, bitplane);`	
`          }`	
`        }`	
`      }`	
`    }`	
`    return SB = cat(SSB_t,s,v,u)`	The side-by-side concatenation of all sub-blocks composes the original block.
`  }`	
`}`	

Table B.44 lists the representations of the hexadeca-tree flags.

Table B.44 — Hexadeca-tree flags

Partition Flag	Representation
lowerBitPlane	0
splitBlock	1
zeroBlock	2

B.3.7 Inverse 4D-DCT procedure

B.3.7.1 General

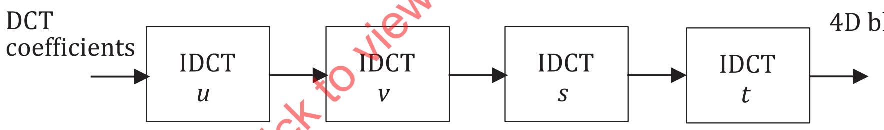


Figure B.20 — Inverse 4D-DCT

As with the direct transform, the inverse transform (4D-IDCT) is separable, i.e. with 1D-IDCTs computed separately in each of the 4 directions. An example of the computation flow of the *t×s×v×u* separable 4D-IDCT is depicted in Figure B.20 (the 4D inverse transform is the same irrespective of the order of application of the inverse 1D transform). After applying the inverse 4D-DCT a level shift operation is performed.

B.3.7.2 Inverse 4D-DCT

For a given light field 4D-DCT representation *X(i,j,p,q)* the corresponding light field *x(u,v,s,t)* can be computed by inverse transforming each dimension in sequence as follows.

$$X^{(uvs)}(i,j,p,t)=\frac{1}{\sqrt{t_k}}\sum_{q=0}^{t_k-1}\alpha(q)\,X(i,j,p,q)\cos\left[\frac{\pi(2t+1)q}{2t_k}\right];\quad \begin{matrix} i=0,1,\ldots,u_k-1; & j=0,1,\ldots,v_k-1 \\ p=0,1,\ldots,s_k-1; & t=0,1,\ldots,t_k-1 \end{matrix}$$

$$X^{(uv)}(i,j,s,t)=\frac{1}{\sqrt{s_k}}\sum_{p=0}^{s_k-1}\alpha(p)\,X^{(uvs)}(i,j,p,t)\cos\left[\frac{\pi(2s+1)p}{2s_k}\right];\quad \begin{matrix} i=0,1,\ldots,u_k-1; & j=0,1,\ldots,v_k-1 \\ s=0,1,\ldots,s_k-1; & t=0,1,\ldots,t_k-1 \end{matrix}$$

$$X^{(u)}\left(i,v,s,t\right)=\frac{1}{\sqrt{v_k}}\sum_{j=0}^{v_k-1}\alpha\left(j\right)\,X^{(uv)}\left(i,j,s,t\right)\cos\left[\frac{\pi\left(2v+1\right)j}{2v_k}\right];\quad \begin{matrix} i=0,1,\ldots,\ u_k-1; & v=0,1,\ldots,v_k-1 \\ s=0,1,\ldots,s_k-1; & t=0,1,\ldots,t_k-1 \end{matrix}$$

$$x\left(u,v,s,t\right)=\frac{1}{\sqrt{u_k}}\sum_{i=0}^{u_k-1}\alpha\left(i\right)\,X^{(u)}\left(i,v,s,t\right)\cos\left[\frac{\pi\left(2u+1\right)i}{2u_k}\right];\quad \begin{matrix} u=0,1,\ldots,\ u_k-1; & v=0,1,\ldots,v_k-1 \\ s=0,1,\ldots,s_k-1; & t=0,1,\ldots,t_k-1 \end{matrix}$$

where $\alpha\left(0\right)=\sqrt{\frac{1}{N}}$, and $\alpha\left(n\right)=\sqrt{\frac{2}{N}};n=1,2,\ldots,\ N-1$, where N is the size of the transform. Output pixels are represented as 32-bit integers. As indicated earlier, the transform order is arbitrary.

B.3.7.3 Inverse level shift

After processing the inverse DCT for a block of source light field samples, the reconstructed samples of the component that are unsigned shall be inversely level shifted. If the MSB of Ssiz^i from the LFC marker segment (see B.4.2.2) is zero, all reconstructed samples *x(u,v,s,t)* of the ith component are level shifted by adding the same quantity from each sample as follows:

$$x\left(u,v,s,t\right)\leftarrow x\left(u,v,s,t\right)+2^{Ssiz^i}$$

NOTE Due to quantization effects, the reconstructed samples *x(u,v,s,t)* can exceed the dynamic range of the original samples. There is no specified procedure for this overflow or underflow situation. However, clipping the value within the original dynamic range is a typical solution.

Annex C
(normative)

JPEG Pleno light field reference view decoding

C.1 General

This annex describes an instantiation of the reference view encoder for the 4D prediction mode. Next, the decoding process is detailed.

C.2 Organization of JPEG Pleno Light Field Reference View superbox

The JPEG Pleno Light Field Reference View box is a superbox that contains the following (see Figure C.1):

— a JPEG Pleno Light Field Reference View Description box, signalling the configuration of the reference view encoding;

— a Common Codestream Elements box, signalling redundant header information from individual codestreams of the reference views;

— a Contiguous Codestream box, containing as payload the individual codestreams of the reference views.

The type of the JPEG Pleno Reference View box shall be ‘lfrv’ (0x6C66 7276).

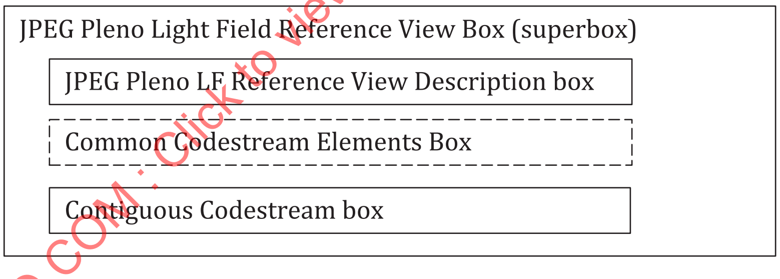

JPEG Pleno Light Field Reference View Box (superbox)
JPEG Pleno LF Reference View Description box
Common Codestream Elements Box
Contiguous Codestream box

Figure C.1 — Organization of the JPEG Pleno Light Field Reference View superbox

C.3 Defined boxes

C.3.1 JPEG Pleno Light Field Reference View Description box

The JPEG Pleno Light Field Reference View Description box contains information on the encoder issued to individually encode the reference views, the number of reference views, which views are encoded as reference view and pointers to the individual codestreams (see Figure C.2 and Table C.1).

The type of the JPEG Pleno Reference View Description box shall be ‘lfrd’ (0x6C66 7264).

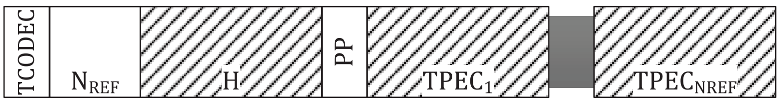


Figure C.2 — Organization of the contents of a Light Field Reference View Description box

Table C.1 — Format of the contents of the Light Field Reference View Description box (optional)

Field name	Size (bits)	Value
TCODEC	8	0 to $(2^8\text{-}1)$
N_{REF}	16	0 to $(2^{16}-1)$
$H(0,0)$ to $H(T\text{-}1,S\text{-}1)$	variable	1 bit per view [0, 1]
PP	8	0 (*Precision* = 32) or 1 (*Precision* = 64)
$TPEC_1$ = pointer to contiguous (EXTRDATA) codestream 1	*Precision*	1 to $(2^{Precision}-1)$
⋮		
$TPEC_{NREF}$ = pointer to contiguous (EXTRDATA) codestream N_{REF}	*Precision*	1 to $(2^{Precision}-1)$

TCODEC identifier for the codec deployed for reference views

TCODEC values shall correspond to the potential values for the coder type C defined in the JPX file format (ISO/IEC 15444-2) for the Image Header box. The default value of $TCODEC = 7$ corresponding to ISO/IEC 15444 (JPEG 2000).

N_{REF} number of reference views (t,s), for which $H(t,s)=1$

H this field specifies per subaperture image (t,s) the type of view as follows:

$H(t,s)==0$ specifies a intermediate view or non-existing view in the array H. In Annex E.3.1 it is specified how this value should be updated to reflect the hierarchical level for the intermediate views.

$H(t,s)==1$ specifies the view (t,s) as a reference view.

8 successive bits are packed as a byte. If $\boldsymbol{T}\times\boldsymbol{S}$ is not a multiple of 8, the remaining bits of the last byte are put to zero (zero padding to stuff last byte). The scan-order of the array H is illustrated by an example in the NOTE in Figure C.3.

PP pointer precision

0 indicates 32-bit precision (unsigned integer, default option) – 1 indicates 64-bit precision (unsigned integer). Other values are not valid.

$TPEC_l$ pointer to contiguous (EXTRDATA) codestream of texture data for texture reference view l, $l=1,2,\ldots,N_{REF}$

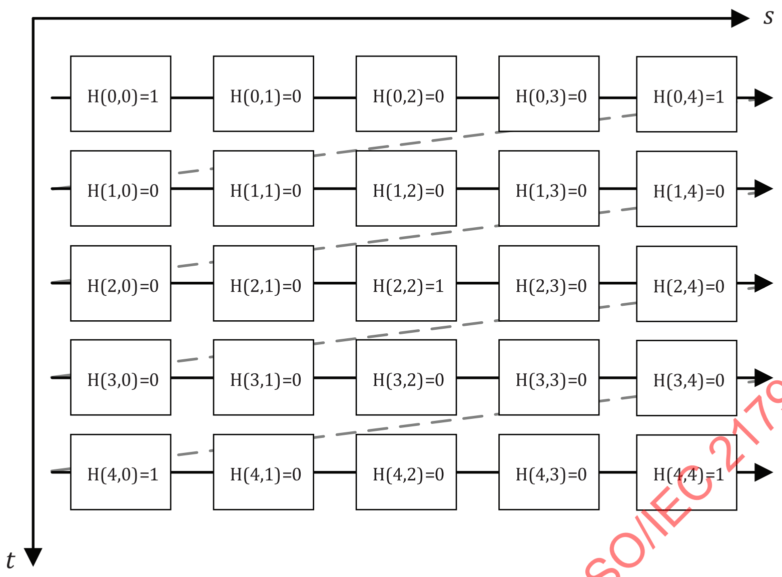
This pointer indicates the position of the EXTRDATA codestream in the Contiguous Codestream box counting from the beginning of this box, i.e. the LBox field.

EXTRDATA external encoded data payload

Contains texture data encoded with the external codec *TCODEC*. The header information produced by the external codec has been removed.

When $TCODEC = 7$ the codec deployed can be ISO/IEC 15444-1 or other parts, such as ISO/IEC 15444-2, which added support for multi-component transforms. The required capabilities are signalled to the JPEG 2000 decoder using the extended capabilities marker (CAP) which was introduced in

ISO/IEC 15444-1. The CAP marker is defined as 0xFF50 followed by a variable length field indicating the Parts of the ISO/IEC 15444 series containing extended capabilities that are used to encode the image.



NOTE Row-wise scan order illustrated with black arrows and grey arrows. Grey arrows indicate move to the beginning of the next row. The coordinates $\left(t_l^X, s_l^X\right)$, $l=1,2,3,4,5$, are the set of $\left(t,s\right)$ where $H\left(t,s\right)$=1.

Figure C.3 — Reference view configuration array H for $T=5, S=5$ with five reference views in centre plus corners configuration

C.3.2 Common Codestream Elements box

This box contains redundant codestream elements extracted from the reference view codestreams (Figure C.4 and Table C.2). If this box is signalled the contained codestream elements, representing codestream header information, shall be concatenated with every codestream fragment contained by the Contiguous Codestream box (Figure C.5 and Table C.3). This box is optional.

The type of the Common Codestream Elements box shall be ‘lfcc’ (0x6C66 6363).

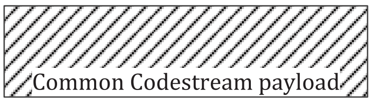


Figure C.4 — Organization of the contents of the Common Codestream Elements box

Table C.2 — Format of the contents of the Common Codestream Elements box

Field name	Size (bits)	Value
Common codestream payload	variable	variable

Common codestream payload	The common codestream element can be, for example, the header information from *TCODEC* encoded file.

C.3.3 Contiguous Codestream box

The Contiguous Codestream box is specified in Annex A.3.4. In this subclause, its payload is specified, which corresponds to the stripped codestreams of the reference views, i.e. codestreams with the redundant header information removed and stored in the Common Codestream Elements box in the JPEG Pleno Reference View box.

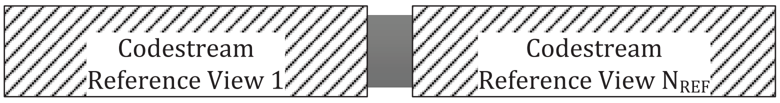


Figure C.5 — Organization of the contents of the Contiguous Codestream box for reference view signalling in the 4D Prediction mode

Table C.3 — Format of the contents of the JPEG Pleno Light Field Contiguous Codestream box for reference view signalling in the 4D Prediction mode

Field name	Size (bits)	Value	Comments
Codestream for reference view 1	variable	variable	External codec codestream
Codestream for reference view 2	variable	variable	External codec codestream
⋮			
Codestream for reference view N_{REF}	variable	variable	External codec codestream

C.4 Reference view encoding

This clause provides a description of encoding operations for the reference views. The following variables are elements obtained from matrix H, used for indexing the reference views:

t_l^X Subscript of the row index for the reference view $l = 1, \ldots, N_{REF}$ in the light field array in row-wise scanning order. Reference views are those views, for which $H(t,s) = 1$.

s_l^X Subscript of the column index for the reference view $l = 1, \ldots, N_{REF}$ in the light field array in row-wise scanning order. Reference views are those views, for which $H(t,s) = 1$.

Reference views are encoded with an external codec, such as $TCODEC$. Since all reference views share the same dimensions and bit depth, the header information of such encoding needs to be obtained only once. For this reason, the externally encoded files are split to two parts: 1) the header information, 2) the remaining codestream. The encoder will place the redundant header information as payload in the common codestream element box, and the decoder needs to concatenate the header to the remaining codestream part prior to the decoding with $TCODEC$.

The texture views of the light field are denoted as,

$$X(t,s,v,u,c).$$

The views are addressed by the (t,s) coordinates pair, while the (v,u) pair addresses a pixel within each (t,s) view and index c stands for colour component.

Similarly, the decoded texture views are denoted as,

$$X^{DEC}(t,s,v,u,c),$$

and the normalized disparity views of the light field as,

$$\tilde{D}(t,s,v,u),$$

and the decoded normalized disparity views as,

$$\tilde{D}^{DEC}(t,s,v,u),$$

where

$t=0,1,\ldots,T-1$

$s=0,1,\ldots,S-1$

$v=0,1,\ldots,V-1$

$u=0,1,\ldots,U-1$

$c=0,1,\ldots,NC-1$.

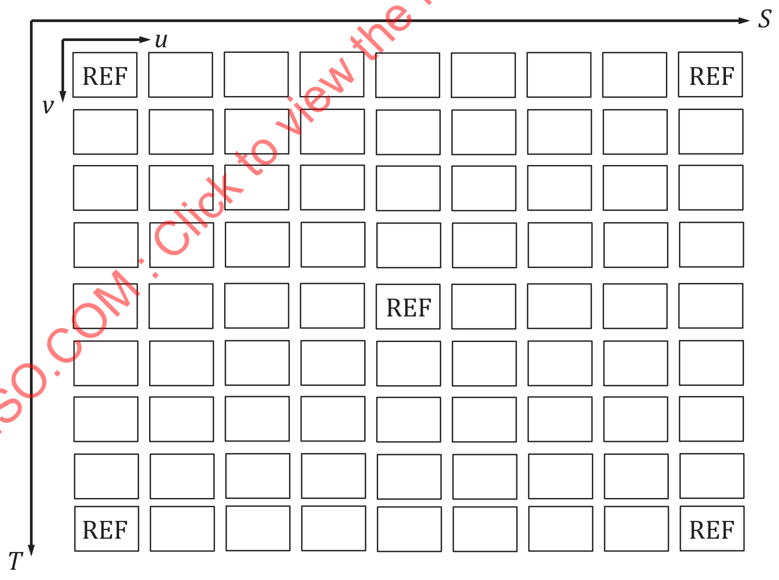


Figure C.6 — Light field with dimensions *T*×*S*×*V*×*U* displaying reference views at locations marked REF ($\left(t_l^X, s_l^X\right)$, $l=1,2,3,4,5$)

For a given reference view $\left(t_l^X, s_l^X\right)$, $l=1,2,\ldots,N_{REF}$ the encoding of texture is performed as in Table C.4. An example configuration of reference views is illustrated in Figure C.6. Note that the numbering follows the row-wise scanning pattern as illustrated in Figure C.3.

NOTE Row-wise scan order illustrated with black arrows and grey arrows. Grey arrows indicate move to the beginning of the next row. The coordinates $\left(t_l^X, s_l^X\right)$, $l=1,2,3,4,5$, are the set of (t,s) where $H(t,s)$=1.

Table C.4 — Procedure for encoding the reference view l

1	Read the l reference view as $\hat{X}_l = X\left(t_l^X, s_l^X\right)$, reference view has dimensions $V \times U \times NC$
2	Encode $\hat{X}_l$ with *TCODEC* at the requested rate
3	Repeat step 1 and 2 until all N_{REF} reference views have been encoded
4	Extract and remove common codestream element *THEADER* from all the *TCODEC* encoded files.
5	Output all stripped *TCODEC* codestreams (EXTRDATA) to the Contiguous Codestream box. The location of each stripped codestream is written to the codestream as pointer $TPEC_l$.
6	Output the common codestream element (*THEADER*) as the payload in the Common Codestream Element box.

The encoding procedure of a reference view is displayed as a flowchart in Figure C.7. In Table C.4, step 4 extracts the header information from the *TCODEC* encoded file. The header information is an array of bytes *THEADER*, and subsequently the bytes are removed from the codestream. The result is a codestream, that is not fully decodable with the *TCODEC* decoder. The header information array *THEADER* needs to be concatenated back prior to decoding. The decoder can obtain the header information from common codestream element box, detailed in Table C.2, append it to the beginning of the stripped codestream (EXTRADATA), and decode successfully. This stripping of header information saves bytes, and makes the encoding less redundant, since only a single unique header is required.

When obtaining the common codestream element, the encoder must use the correct markers for identifying the header section in the encoded codestream. In the case of $TCODEC == 7$, the deployed codec is JPEG 2000, and the marker for identifying the last two bytes of the header is the SOT marker (0xFF90), i.e. the beginning of a tile marker. For other *TCODEC* types, the marker will depend on the chosen codec. Note that the decoding process of the common codestream element is transparent to the codec type. Hence, no normative constraints need to be imposed on the exact cutting point of the *TCODEC* codestream.

The process of Table C.4 applies to all reference views. The encoder will output all the reference view data and common codestream element information according to the format of Light Field Reference View Description box.

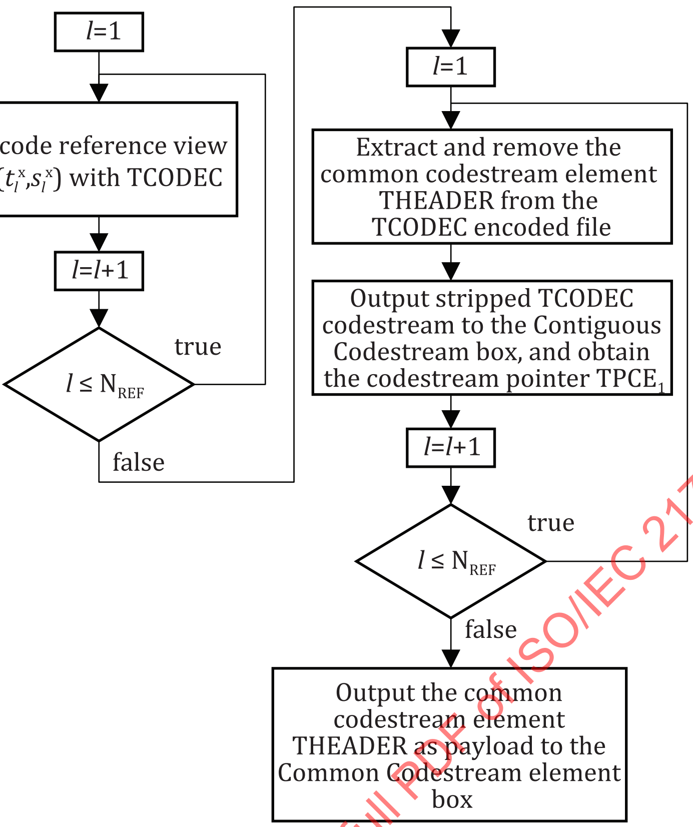

l=1
Encode reference view X(t_l^x,s_l^x) with TCODEC
l=l+1
l ≤ N_REF
true
false
l=1
Extract and remove the common codestream element THEADER from the TCODEC encoded file
Output stripped TCODEC codestream to the Contiguous Codestream box, and obtain the codestream pointer TPCE_l
l=l+1
l ≤ N_REF
true
false
Output the common codestream element THEADER as payload to the Common Codestream element box

Figure C.7 — Overview of reference view encoding of view $\left(t_l^X, s_l^X\right)$, $l=1,2,\ldots,N_{REF}$

C.5 Reference view decoding

This clause provides a description of decoding operations for the reference views. Reference views are views that use no prediction. These views are the views on the lowest hierarchical level where $H(t,s)=1$.

Since, the externally encoded files are split to two parts 1) the header information 2) the remaining codestream, the decoder needs to concatenate the header to the remaining codestream part prior to the decoding with $TCODEC$.

See Figure C.8 for an overview of the reference view decoding process. The detailed steps for decoding the reference view $\left(t_l^X, s_l^X\right)$, $l=1,2,\ldots,N_{REF}$ are given in Table C.5.

Table C.5 — Procedure for decoding $X^{DEC}\left(t_l^X, s_l^X\right)=\hat{X}_l^{DEC}$

1	Obtain the header payload from Common Codestream Element box, and store the payload as an array of bytes $THEADER$.
2	Obtain the reference view *i* data from the contiguous codestream box, pointed to by $TPEC_l$, and store the data as an array of bytes $TDATA$.
3	Concatenate THEADER and TDATA as $TENCODING=\left[THEADER, TDATA\right]$.
4	Decode the $V\times U\times NC$ image $\hat{X}_l^{DEC}$ from $TENCODING$ with the external codec $TCODEC$.
5	$X^{DEC}\left(t_l^X, s_l^X\right)=\hat{X}_l^{DEC}$.
6	Repeat from step 2 until all reference views have been decoded.

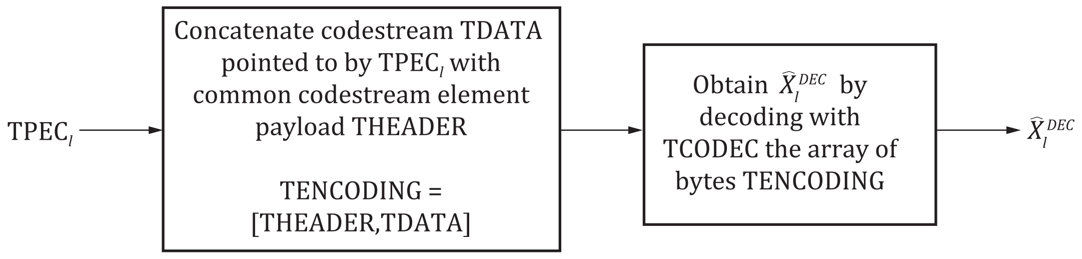

TPEC$_l$
Concatenate codestream TDATA pointed to by TPEC$_l$ with common codestream element payload THEADER
TENCODING = [THEADER,TDATA]
Obtain $\widehat{X}_l^{DEC}$ by decoding with TCODEC the array of bytes TENCODING
$\widehat{X}_l^{DEC}$

Figure C.8 — Overview of decoding reference view $X^{DEC}\left(t_l^X, s_l^X\right) = \hat{X}_l^{DEC}$ $, l = 1, 2, \ldots, N_{REF}$

Annex D
(normative)

JPEG Pleno light field normalized disparity view decoding

D.1 General

This annex describes an instantiation of the normalized disparity view encoder for the 4D prediction mode. Thereafter, the decoding process is detailed.

D.2 Organization of JPEG Pleno Light Field Normalized Disparity View Superbox

The JPEG Pleno Light Field Normalized Disparity View box is a superbox that contains the following (see Figure D.1):

— a JPEG Pleno Light Field Normalized Disparity View Description box, signalling the configuration of the normalized disparity view encoding;

— a Common Codestream Elements box, signalling redundant header information from individual codestreams of the normalized disparity views;

— a Contiguous Codestream box, containing has payload the individual codestreams of the normalized disparity views.

The type of JPEG Pleno Light Field Normalized Disparity View box shall be ‘lfdv’ (0x6C66 6476).

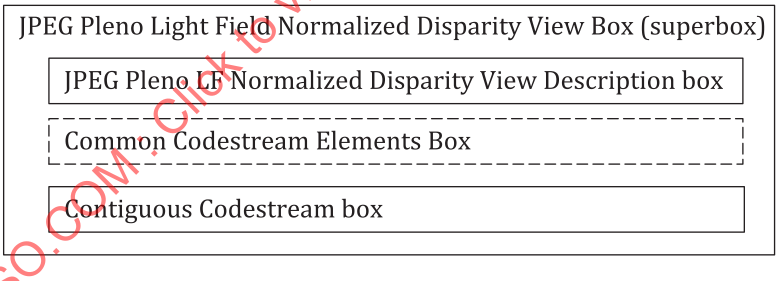

JPEG Pleno Light Field Normalized Disparity View Box (superbox)
JPEG Pleno LF Normalized Disparity View Description box
Common Codestream Elements Box
Contiguous Codestream box

Figure D.1 — Organization of the JPEG Pleno Light Field Normalized Disparity View superbox

D.3 Defined boxes

D.3.1 JPEG Pleno Light Field Normalized Disparity View Description box

The JPEG Pleno Light Field Normalized Disparity View Description box contains information on the encoder issued to individually encode the normalized disparity views, the number of normalized disparity views, which views are encoded as normalized disparity view, and pointers to the individual codestreams.

The type of JPEG Pleno Light Field Normalized Disparity View Description box shall be ‘lfdd’ (0x6C66 6464).

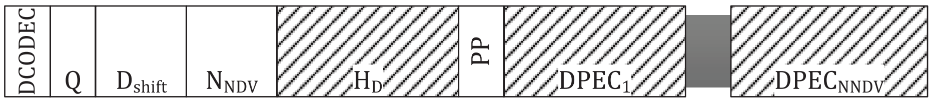


Figure D.2 — Organization of the contents of a Light Field Normalized Disparity View Description box

Table D.1 — Format of the contents of the Light Field Normalized Disparity View Description box

Field name	Size (bits)	Value
DCODEC	8	0 to $(2^8\text{-}1)$
Q	8	0 to $(2^8\text{-}1)$
D_{shift}	16	0 to $(2^{16}-1)$
N_{NDV}	16	0 to $(2^{16}-1)$
$H_D(0,0)$ *to* $H_D(T-1,S-1)$	variable	1 bit per view [0, 1]
PP	8	0 (*Precision* = 32) or 1 (*Precision* = 64)
$DPEC_1$ *= Pointer to contiguous (EXTRDATA) codestream 1*	*Precision*	1 to $(2^{Precision}-1)$
⋮		
$DPEC_{NNDV}$ *= Pointer to contiguous (EXTRDATA) codestream* N_{NDV}	*Precision*	1 to $(2^{Precision}-1)$

DCODEC identifier for the codec deployed for normalized disparity views

DCODEC values shall correspond to the potential values for the coder type C defined in the JPX file format (ISO/IEC 15444-2) for the Image Header box. The default value of *DCODEC* = 7 corresponding to ISO/IEC 15444 (JPEG 2000).

Q depth quantization parameter

D_{shift} minimum normalized disparity value (optional)

The integer value is represented in unsigned 16 bit. The D_{shift} quantity is subtracted from the encoded normalized disparity data to level-shift the normalized disparity data to cover the negative and positive range of the normalized disparity field. Intended to be used with DCODECs which do not support negative data.

$\boldsymbol{N_{NDV}}$ number of normalized disparity (reference) views (t,s), for which $H_D(t,s)=1$

$\boldsymbol{H_D(t,s)}$ this field specifies per normalized disparity view (t,s) the type of view as follows:

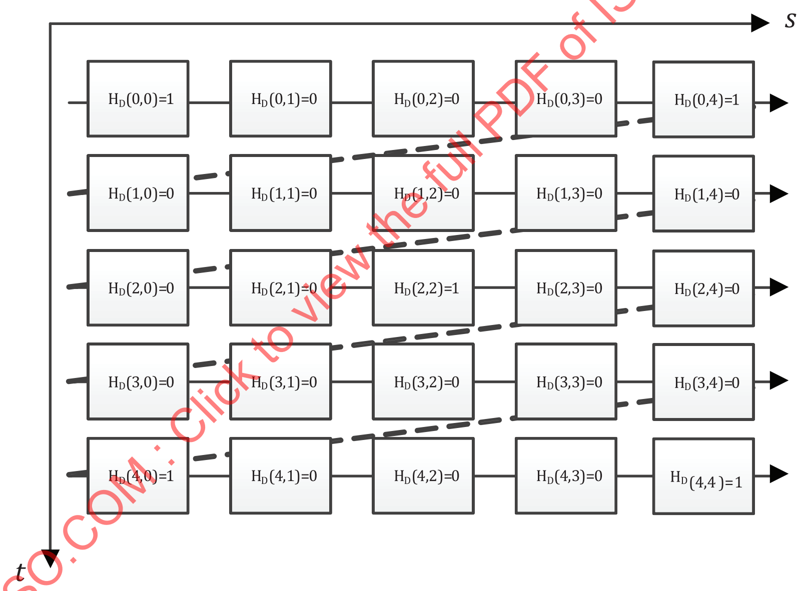
$H_D(t,s)==0$, normalized disparity view not supplied.

$H_D(t,s)==1$, normalized disparity view supplied.

8 successive bits are packed as a byte. If $T \times S$ is not a multiple of 8, the remaining bits of the last byte are put to zero (zero padding to stuff last byte). The scan-order of the normalized disparity views is illustrated by an example in the NOTE in Figure D.3.

PP	pointer precision 0 indicates 32-bit precision (unsigned integer, default option) – 1 indicates 64-bit precision (unsigned integer). Other values are not valid.
$DPEC_k$	pointer to contiguous (EXTRDATA) codestream of normalized disparity data for normalized disparity reference view k, $k = 1, 2, \ldots, N_{IDV}$ This pointer indicates the position of the EXTRDATA codestream in the Contiguous Codestream box counting from the beginning of this box, i.e. the LBox field.
EXTRDATA	externally encoded data payload Contains disparity data encoded with the external codec DCODEC.

When $DCODEC = 7$ the codec deployed can be ISO/IEC 15444-1 or other parts , such as ISO/IEC 15444-2, which added support for multi-component transforms. The required capabilities are signalled to the JPEG 2000 decoder using the extended capabilities marker (CAP) which was introduced in ISO/IEC 15444-1. The CAP marker is defined as 0xFF50 followed by a variable length field indicating the Parts in the ISO/IEC 15444 series containing extended capabilities that are used to encode the image.



NOTE Row-wise scan order illustrated with black arrows and grey arrows. Grey arrows indicate move to the beginning of the next row. The coordinates $\left(t_k^D, s_k^D\right)$, $k = 1, 2, 3, 4, 5$, are the set of (t, s) where $H_D(t, s) = 1$.

Figure D.3 — Normalized disparity view configuration array H_D for $T = 5, S = 5$ with five reference views in centre plus corners configuration

D.3.2 Common Codestream Elements box

The redundant codestream syntax from the individual codestreams of the normalized disparity data is extracted and signalled as specified in Annex C.3.2 for the reference views. This box is optional.

D.3.3 Contiguous Codestream box

The Contiguous Codestream box has been specified in Annex A.3.4. In this subclause, its payload is specified, which is corresponding to the stripped codestreams of the normalized disparity views, i.e. codestreams with the redundant header information removed and stored in the Common Codestream Elements box in the JPEG Pleno Normalized Disparity View box (see Figure D.4 and Table D.2).

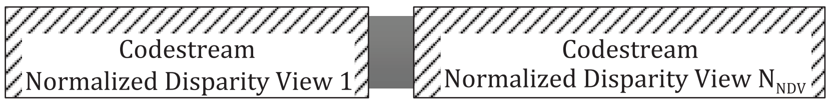


Figure D.4 — Organization of the contents of the Contiguous Codestream box for normalized disparity view signalling in the 4D Prediction mode

Table D.2 — Format of the contents of the JPEG Pleno Light Field Contiguous Codestream box for normalized disparity view signalling in the 4D Prediction mode

Field name	Size (bits)	Value	Comments
Codestream for normalized disparity view 1	variable	variable	External codec codestream
Codestream for normalized disparity view 2	variable	variable	External codec codestream
Codestream for normalized disparity view N_{NDV}	variable	variable	External codec codestream

D.4 Representation of the normalized disparity data

The horizontal disparity map D_x and vertical disparity map D_y between the views (t_1, s_1) and (t_2, s_2) are expressing the correspondence of the pixel (t_1, s_1, v_1, u_1) with the pixel $t_2, s_2, \hat{v}, \hat{u}$, where

$$\hat{u} = \lfloor u_1 + D_x(v_1, u_1) \rceil$$

$$\hat{v} = \lfloor v_1 + D_y(v_1, u_1) \rceil.$$

If the pixel $(t_2, s_2, \hat{v}, \hat{u})$ is not occluded by another pixel $(t_2, s_2, \hat{v}', \hat{u}')$, then the colour attributes $X(t_1, s_1, v_1, u_1, c)$ and $X(t_2, s_2, \hat{v}, \hat{u}, c)$ are very similar one to another, allowing to predict one from the other.

The disparity maps can be estimated from the texture views (t_1, s_1) and (t_2, s_2) using optical flow stereo estimation methods.[2] For the case of light fields there exist more specialized methods utilizing the whole light field in the estimation.[3][4] In the latter category, the disparity maps between each pairs of views can be obtained using the normalized disparity map, defined as,

$$\tilde{D}(t_1, s_1, v_1, u_1) = \frac{D_x(v_1, u_1)}{B_x} = \frac{D_y(v_1, u_1)}{B_y} \in \mathbb{R},$$

where

$$B_x = XCC(t_1, s_1) - XCC(t_2, s_2),$$

$$B_y = YCC(t_1, s_1) - YCC(t_2, s_2).$$

The normalized disparity of pixel (v,u) in view (t,s) is denoted as $\tilde{D}(t,s,v,u)$. For a pair of two arbitrary views (t_1,s_1) and (t_2,s_2) the normalized disparity map can be used to find corresponding pixels by,

$$X(t_2,s_2,\hat{v},\hat{u},c) \approx X(t_1,s_1,v_1,u_1,c) \text{ for } \forall c \in \{0,\ldots,NC-1\},$$

where

$$\hat{u} = \lfloor u_1 + \tilde{D}(t_1,s_1,v_1,u_1)\cdot B_x \rceil,$$

$$\hat{v} = \lfloor v_1 + \tilde{D}(t_1,s_1,v_1,u_1)\cdot B_y \rceil.$$

D.5 Encoding of normalized disparity data

This clause provides details how the normalized disparity data is encoded using *DCODEC* encoder. Normalized disparity data is used to predict intermediate views from reference views using disparity-based warping, see Annex E.

The normalized disparity views are real-valued floating-point quantities. Denote the floating-point normalized disparity view as $\tilde{D}$. A quantized integer precision normalized disparity view is obtained as,

$$\tilde{D}^{quant} = \lceil \tilde{D}\cdot 2^Q \rfloor,$$

where Q is the quantization factor.

For codecs which do not support encoding of negative data, the quantized normalized disparity is level-shifted to positive range using the constant D_{shift}. The shifting constant D_{shift} is common to all normalized disparity maps in the light field and is obtained over all quantized normalized disparity maps. In this case of level-shifting to positive range, the quantized normalized disparity data becomes,

$$\tilde{D}^{quant} = \tilde{D}\cdot 2^Q + D_{shift}.$$

The normalized disparity views are usually provided at lowest hierarchical level (see the example in Figure D.5), and are used to predict subsequent intermediate views at higher hierarchies. Normalized disparity views are encoded with an external codec, such as *DCODEC*. Since all normalized disparity views share the same dimensions and bit depth, the header information of such encoding needs to be obtained only once. For this reason, the externally encoded files are split to two parts: 1) the header information, 2) the remaining codestream. The encoder will place the redundant header information as payload in the common codestream element box, and the decoder needs to concatenate the header to the remaining codestream part prior to the decoding with *DCODEC*.

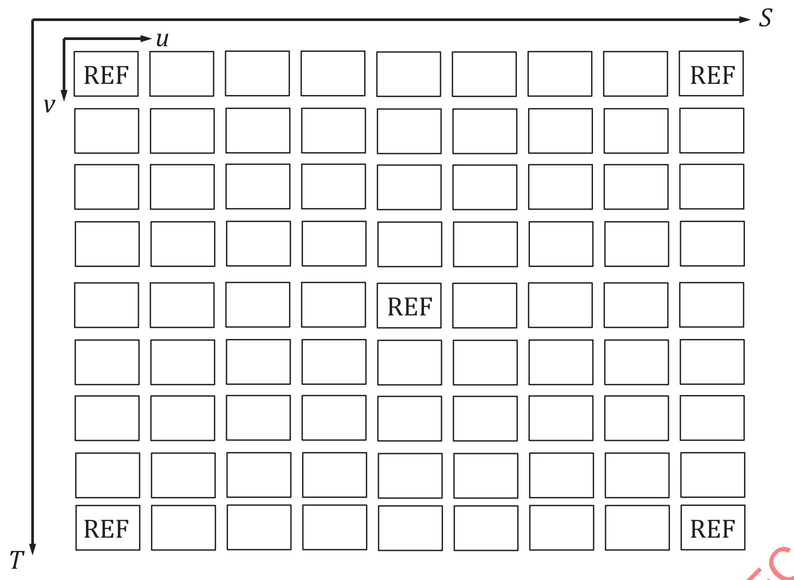


Figure D.5 — Light field with dimensions $T \times S \times V \times U$ displaying normalized disparity reference views at locations marked REF (in row-wise scan order $\left(t_k^D, s_k^D\right)$, $k = 1, 2, 3, 4, 5$)

The following variables are elements obtained from matrix H_D, used for indexing the normalized disparity views,

t_k^D	Subscript of the row index for the normalized disparity view $k = 1, \ldots, N_{NDV}$ in the light field array in row-wise scanning order. Normalized disparity views are those views, for which $H_D\left(t_l^x, s_l^x\right) = 1$, for $l = 1, \ldots, N_{REF}$.
s_k^D	Subscript of the column index for the normalized disparity view $\mathrm{k} = 1, \ldots, N_{NDV}$ in the light field array in row-wise scanning order. Normalized disparity views are those views, for which $H_D\left(t_l^x, s_l^x\right) = 1$, for $l = 1, \ldots, N_{REF}$.

An example configuration of normalized disparity views is given in Figure D.5. For a given normalized disparity view $\left(t_k^D, s_k^D\right)$, $k = 1, \ldots, \mathrm{N}_{NDV}$ the encoding of normalized disparity using *DCODEC* is performed as in Table D.3.

Table D.3 — Normalized disparity view encoding procedure for normalized disparity view k when using $DCODEC$

1	Obtain reference normalized disparity view as $\hat{D}_k = \tilde{D}\left(t_k^D, s_k^D\right)$, reference normalized disparity view has dimensions $V \times U$
2	Quantize by rounding to nearest integer after multiplication by 2^Q, $\hat{D}_k^{quant} = \lfloor \hat{D}_k \cdot 2^Q \rceil$
3	Level-shift by adding the quantity D_{shift}: $\hat{D}_k^{quant} = \lfloor \hat{D} \cdot 2^Q \rceil + D_{shift}$
4	Encode $\hat{D}_k^{quant}$ with $DCODEC$ at the requested rate
5	Extract and remove common codestream element $DHEADER$ from the $DCODEC$ encoded files.
6	Output to codestream the headerless $DCODEC$ encodings (EXTRDATA). The location of this data in the codestream is written to the codestream as pointer $DPEC_k$.
7	Output the common codestream element as the payload in the Common Codestream Element box, see Table C.2.
8	Obtain reference normalized disparity view as $\hat{D}_k = \tilde{D}\left(t_k^D, s_k^D\right)$, reference normalized disparity view has dimensions $V \times U$

Step 5 extracts the header information from the $DCODEC$ encoded file. The header information is an array of bytes $DHEADER$, and subsequently the bytes are removed from the encoded file. The result is an encoding, that is not fully decodable with the $DCODEC$ decoder. The header information array $DHEADER$ needs to be concatenated back prior to decoding. The decoder can obtain the header information from common codestream element box, append it to the beginning of the headerless encoding, and decode successfully. This stripping of header information saves bytes, and makes the encoding less redundant, since only a single unique header is required.

When obtaining the common codestream element, the encoder must use the correct markers for identifying the header section in the encoded codestream. In the case of $DCODEC == 7$, the deployed codec is JPEG 2000, and the marker for identifying the last two bytes of the header is 0xFF90, i.e. the beginning of a tile marker. For other $DCODEC$ types, the marker will depend on the chosen codec. Note that the decoding process of the common codestream element is transparent to the codec type.

D.6 Decoding of normalized disparity data

This clause provides details on decoding the normalized disparity data using $DCODEC$ decoder. The normalized disparity views are usually provided at lowest hierarchical level and used to predict subsequent intermediate views at higher hierarchical levels, see Annex E.

The normalized disparity views are real-valued floating-point quantities. During encoding they are quantized into integer range by a multiplication with 2^Q followed by rounding to nearest integer and encoded using 16 bits. Optionally, for normalized disparity views with negative values the data is level shifted by subtracting the quantity D_{shift}, see Table D.1. The D_{shift} value is obtained only once over all normalized disparity views. Level shifting is not necessary for codecs which support negative input data.

Consider an encoded normalized disparity view $\breve{D}_k^{quantDEC}$ obtained from decoding a codestream encoded with $DCODEC$. First, the decoder level-shifts the data using,

$$\hat{D}_k^{quantDEC} = \breve{D}_k^{quantDEC} - D_{shift},$$

Next, the inverse quantization step is applied,

$$\hat{D}_k^{DEC} = (\hat{D}_k^{quantDEC})/2^Q,$$

where $\hat{D}_k^{DEC}$ is truncated to 16 fractional bits and is representing the final decoded normalized disparity view. It is assigned to $\tilde{D}^{DEC}\left(t_k^D, s_k^D\right) = \hat{D}_k^{DEC}$.

The steps for decoding the normalized disparity view $\left(t_k^D, s_k^D\right)$, $k = 1, 2, \ldots, \mathrm{N}_{NDV}$ using $DCODEC$ are given in Table D.4.

Table D.4 — Normalized disparity view decoding procedure for normalized disparity view *k* using *DCODEC*

1	Obtain header payload from Common Codestream Element box as an array of bytes $DHEADER$.
2	Obtain the normalized disparity view *i* data from the contiguous codestream box, pointed to by $DPEC_l$, and store the data as an array of bytes $DDATA$.
3	Concatenate DHEADER and DDATA as $DENCODING = [DHEADER, DDATA]$.
4	Decode $DENCODING$ with external codec $DCODEC$ to obtain $V \times U$ image $\hat{D}_k^{quantDEC}$
5	Level shift by subtracting D_{shift}, $\hat{D}_k^{quantDEC} = \breve{D}_k^{quantDEC} - D_{shift}$ (optional step)
6	Dequantize to floating-point values by division with 2^Q: $\hat{D}_k^{DEC} = (\hat{D}_k^{quantDEC})/2^Q$
7	$\tilde{D}^{DEC}\left(t_k^D, s_k^D\right) = \hat{D}_k^{DEC}$.

Annex E
(normative)

JPEG Pleno Light Field Intermediate View superbox

E.1 General

This annex specifies the decoding process for intermediate view decoding in the 4D prediction mode. It also specifies the superbox containing all information necessary to reconstruct and to decode the intermediate views, such as prediction parameters and encoded residual prediction data.

However, for informative purposes, it first describes an instantiation of the intermediate view encoder for the 4D prediction mode.

E.2 Organization of JPEG Pleno Light Field Intermediate View superbox

The JPEG Pleno Light Field Intermediate View box is a superbox that contains the following (see Figure E.1):

— a JPEG Pleno Light Field Prediction Parameter box, signalling the parameters of the intermediate view prediction;

— a JPEG Pleno Light Field Residual View Description box, signalling the configuration of the residual view encoding;

— a Common Codestream Elements box, signalling redundant header information from individual codestreams of the residual views;

— a Contiguous Codestream box, containing has payload the individual (stripped) codestreams of the residual views.

The type of JPEG Pleno Light Field Intermediate View box shall be ‘lfiv’ (0x6C66 6976).

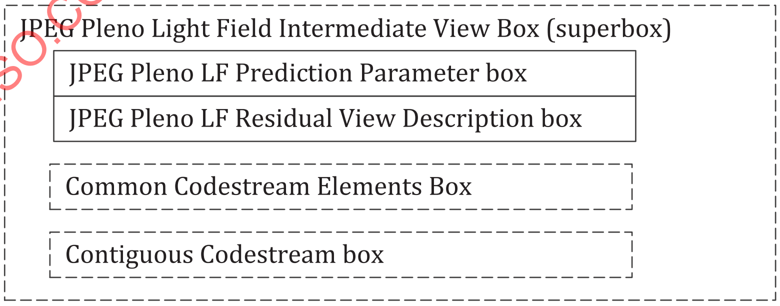

JPEG Pleno Light Field Intermediate View Box (superbox)
JPEG Pleno LF Prediction Parameter box
JPEG Pleno LF Residual View Description box
Common Codestream Elements Box
Contiguous Codestream box

Figure E.1 — Organization of the JPEG Pleno Light Field Intermediate View superbox

E.3 Defined boxes

E.3.1 JPEG Pleno Light Field Prediction Parameter box

The prediction parameters box contains prediction parameters for the N_I intermediate views $\left(t_p^I, s_p^I\right)$, for which $H\left(t_p^I, s_p^I\right)=0$.

For each intermediate view $\left(t_p^I, s_p^I\right)$ this box contains updated hierarchy information for the intermediate view configuration, view merging mode options and sparse filter parameters. For each intermediate view $\left(t_p^I, s_p^I\right)$, the prediction parameters are contained in a separate prediction parameter block (see Figure E.2 and Table E.1).

The type of JPEG Pleno Light Field Prediction Parameter box shall be 'lfpp' (0x6C66 7070).

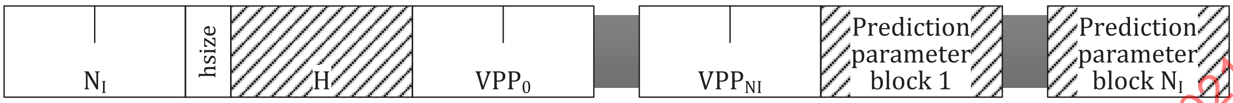


Figure E.2 — Organization of the contents of a Light Field Prediction Parameter box

Table E.1 — Format of the contents of the Light Field Prediction Parameters Description box

Field name	Size (bits)	Value
N_I	32	0 to $(2^{32}-1)$
hsize	8	1 to (2^8-1)
$H\left(t_1^I, s_1^I\right)$ to $H\left(t_{N_I}^I, s_{N_I}^I\right)$	variable	$[0, (2^{hsize}-1)]$ in *hsize* bit per view
VPP_1	32	1 to $(2^{32}-1)$
⋮		
VPP_{N_I}	32	1 to $(2^{32}-1)$
Prediction parameter block$_1$	variable	variable
⋮		
Prediction parameter block$_{NI}$	variable	variable

N_I number of intermediate views, $N_I = N - N_{REF}$, where N equals the total number of views in the light field, and N_{REF} equals the total number of reference views

hsize precision in bit for specifying the number of hierarchical levels in the array H (up to 3 levels requires 2 bits, from 4 to 7 levels 3 bits, etc.)

$H(t,s)$ the value specifying the type of view (t,s) as follows:

$H(t,s) == 0$ is reserved for non-existing views in the array H.

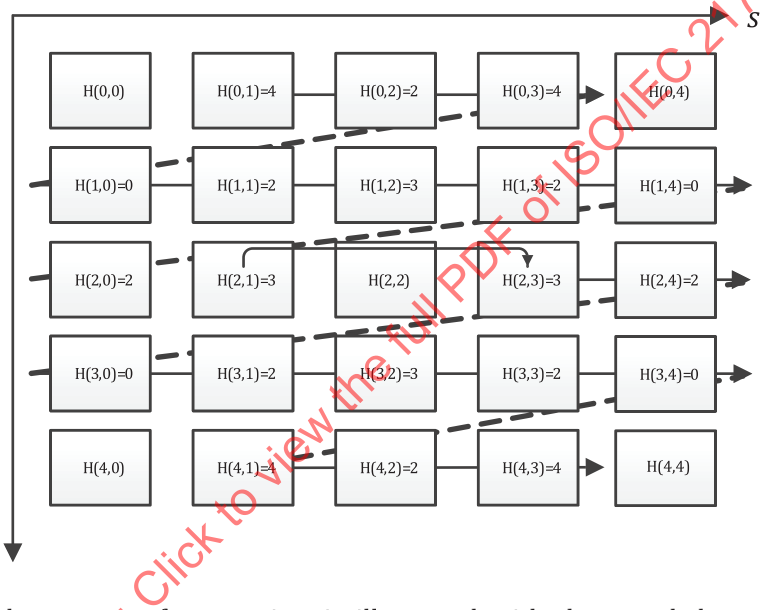
$H(t,s) == 1$ is not allowed (reserved for reference views).

$H(t,s) > 1$ signals the hierarchy level of the intermediate view (t,s).

Please note that the indexes t_p^I and s_p^I are ordered according to a row-wise scan (see NOTE to Figure E.3).

The *hsize* bit per view are concatenated to one stream. 8 Successive bits are packed as a byte. If $N_I \times hsize$ is not a multiple of 8, the remaining bits of the last byte are put to zero (zero padding to stuff last byte). The scan-order for H is illustrated in the NOTE to Figure E.3.

t_p^I	subscript of the row index in row-wise scanning order for the intermediate view $p=1,\ldots,N_I$ in the light field array
s_p^I	subscript of the column index in row-wise scanning order for the intermediate view $p=1,\ldots,N_I$ in the light field array
VPP_p	pointer to Prediction Parameters block for intermediate view $\left(t_p^I, s_p^I\right)$, $p=1,\ldots,N_I$ in the light field array in row-wise scanning order This pointer indicates the position of this Prediction Parameters block in the Prediction Parameters Description box counting from the beginning of this box, i.e. the LBox field.



NOTE Skipping of the centre reference view is illustrated with the rounded arrow. Hierarchy level is indicated when $H(t,s)\geq 1$. In this example there are four hierarchical levels $\{1,2,3,4\}$ and four non-existing views with $H(t,s)=0$

Figure E.3 — Subaperture views are scanned in row-wise order skipping the reference views $H(t,s)=1$ that have been signalled earlier (see Annex C.3.1)

For each intermediate view $\left(t_p^I, s_p^I\right)$, $p=1,\ldots,N_I$ the prediction parameters are contained in a separate prediction parameter block, see Table E.2.

Table E.2 — Format of the contents of the Prediction Parameter block

Field name	Size (bits)	Value
N_p^T	8	0 to (2^8-1)
N_p^D	8	0 to (2^8-1)
t_1^{Tr}	16	0 to (2^{16}– 1)
s_1^{Tr}	16	0 to (2^{16}– 1)

Table E.2 *(continued)*

Field name	Size (bits)	Value
	$\vdots$	
$t^{Tr}_{N^T_p}$	16	0 to (2^{16}– 1)
$s^{Tr}_{N^T_p}$	16	0 to (2^{16}– 1)
t^{Dr}_1	16	0 to (2^{16}– 1)
s^{Dr}_1	16	0 to (2^{16}– 1)
	$\vdots$	
$t^{Dr}_{N^D_p}$	16	0 to (2^{16}– 1)
$s^{Dr}_{N^D_p}$	16	0 to (2^{16}– 1)
$MMODE_p$	8	{0,1,2}
SF_p	8	[0,1]
	$MMODE_p == 0$ *(least-squares optimal view merging)*	
$LSW^{p,0}_1$	16	-(2^{15}-1) to (2^{15}-1)
$LSW^{p,0}_2$	16	-(2^{15}-1) to (2^{15}-1)
	$\vdots$	
$LSW^{p,N_c-1}_{NLS_p}$	16	-(2^{15}-1) to (2^{15}-1)
	$MMODE_p == 1$ *(geometric distance based view merging)*	
FPW_p	32	single precision, big endian floating-point
	$SF_p == 1$ *(sparse filter enabled)*	
NRT_p	8	2^8-1
MSP_p	8	2^8-1
$SPW^{p,0}_1$	32	-(2^{31}-1) to (2^{31}-1)
$SPW^{p,0}_2$	32	-(2^{31}-1) to (2^{31}-1)
	$\vdots$	
$SPW^{p,N_c-1}_{M_{SP}}$	32	-(2^{31}-1) to (2^{31}-1)
$KR_{p,0}$	$(N_{sp}+1)$	$0 \text{ to } \left(2^{(N_{sp}+1)}-1\right)$
	$\vdots$	
KR_{p,N_c-1}	$(N_{sp}+1)$	$0 \text{ to } \left(2^{(N_{sp}+1)}-1\right)$

$\boldsymbol{N_p^T}$	number of reference views for intermediate view $\left(t_p^I, s_p^I\right)$, $p = 1, \ldots, N_I$
$\boldsymbol{N_p^D}$	number of normalized disparity views for intermediate view $\left(t_p^I, s_p^I\right)$, $p = 1, \ldots, N_I$
$\boldsymbol{t_{ii}^{Tr}}$	subscript of the row index of the reference view, $ii = 1, 2, \ldots, N_p^T$ in the subaperture image array in row-wise scanning order
$\boldsymbol{s_{ii}^{Tr}}$	subscript of the column index of the reference view, $ii = 1, 2, \ldots, N_p^T$ in the subaperture image array in row-wise scanning order The set of reference views for intermediate view p is defined as $\Omega_p^{Tr} = \{(t_1^{Tr}, s_1^{Tr}), (t_2^{Tr}, s_2^{Tr}), \ldots, (t_{N_P^T}^{Tr}, s_{N_P^T}^{Tr})\}$
$\boldsymbol{t_{jj}^{Dr}}$	subscript of the row index of the normalized disparity view, $jj = 1, 2, \ldots, N_p^D$ in the subaperture image array in row-wise scanning order
$\boldsymbol{s_{jj}^{Dr}}$	subscript of the column index of the normalized disparity view, $jj = 1, 2, \ldots, N_P^D$ in the subaperture image array in row-wise scanning order The set of normalized disparity reference views for intermediate view p is defined as $\Omega_p^{Dr} = \{(t_1^{Dr}, s_1^{Dr}), (t_2^{Dr}, s_2^{Dr}), \ldots, (t_{N_p^D}^{Dr}, s_{N_p^D}^{Dr})\}$
$\boldsymbol{MMODE_p}$	view merging mode for texture view p, $p = 1, 2, \ldots, N_I$

$\boldsymbol{MMODE_p}$	**View merging mode for view** $\left(\boldsymbol{t_p^I}, \boldsymbol{s_p^I}\right)$
0	Least-squares merging, Annex E.5.4.1
1	Fixed-weight merging, Annex E.5.4.2
2	Median merging, Annex E.7

$\boldsymbol{SF_p}$	sparse filter enabled/disabled at view p, $p = 1, 2, \ldots, N_I$, $SF_p == 1$ sparse filter enabled, $SF_p == 0$ sparse filter disabled.
$\boldsymbol{NLS_p}$	number of LS merging coefficients for view p, computed as $\left(N_p^T \cdot 2^{N_p^T}\right)/2$
$\boldsymbol{LSW_j^{p,c}}$	least-squares merging weight of component c for view p, $j = 1, 2, \ldots, NLS_p$
$\boldsymbol{FPW_p}$	fixed-weight merging parameter for view p
$\boldsymbol{NRT_p}$	regressor template parameter of sparse filter for view p Defines the size of the sparse predictor neighbourhood from which the regressors are selected.