INTERNATIONAL ORGANISATION FOR STANDARDISATION
ORGANISATION INTERNATIONALE DE NORMALISATION
ISO/IEC JTC1/SC29/WG11
CODING OF MOVING PICTURES AND AUDIO

ISO/IEC JTC1/SC29/WG11 N2202

Tokyo, March 1998

INFORMATION TECHNOLOGY -

CODING OF AUDIO-VISUAL OBJECTS: VISUAL

ISO/IEC 14496-2

Committee Draft

Draft of 15 May, 1998

Contents

Foreword

(Foreword to be provided by ISO)

1. Introduction

1.1 Purpose

This part of this specification was developed in response to the growing need for a coding method that can facilitate access to visual objects in natural and synthetic moving pictures and associated natural or synthetic sound for various applications such as digital storage media, internet, various forms of wired or wireless communication etc. The use of this specification means that motion video can be manipulated as a form of computer data and can be stored on various storage media, transmitted and received over existing and future networks and distributed on existing and future broadcast channels.

1.2 Application

The applications of this specification cover, but are not limited to, such areas as listed below:

IMM	Internet Multimedia
IVG	Interactive Video Games
IPC	Interpersonal Communications (videoconferencing, videophone, etc.)
ISM	Interactive Storage Media (optical disks, etc.)
MMM	Multimedia Mailing
NDB	Networked Database Services (via ATM, etc.)
RES	Remote Emergency Systems
RVS	Remote Video Surveillance
WMM	Wireless Multimedia

1.3 Profiles and levels

This specification is intended to be generic in the sense that it serves a wide range of applications, bitrates, resolutions, qualities and services. Furthermore, it allows a number of modes of coding of both natural and synthetic video in a manner facilitating access to individual objects in images or video, referred to as content based access. Applications should cover, among other things, digital storage media, content based image and video databases, internet video, interpersonal video communications, wireless video etc. In the course of creating this specification, various requirements from typical applications have been considered, necessary algorithmic elements have been developed, and they have been integrated into a single syntax. Hence this specification will facilitate the bitstream interchange among different applications.

This specification includes one or more complete decoding algorithms as well as a set of decoding tools. Moreover, the various tools of this specification as well as that derived from ISO/IEC 13818-2 can be combined to form other decoding algorithms. Considering the practicality of implementing the full syntax of this specification, however, a limited number of subsets of the syntax are also stipulated by means of “profile” and “level”.

A “profile” is a defined subset of the entire bitstream syntax that is defined by this specification. Within the bounds imposed by the syntax of a given profile it is still possible to require a very large variation in the performance of encoders and decoders depending upon the values taken by parameters in the bitstream.

In order to deal with this problem “levels” are defined within each profile. A level is a defined set of constraints imposed on parameters in the bitstream. These constraints may be simple limits on numbers. Alternatively they may take the form of constraints on arithmetic combinations of the parameters.

1.4 Object based coding syntax

1.4.1 Video object

A *video object* in a scene is an entity that a user is allowed to access (seek, browse) and manipulate (cut and paste). The instances of video objects at a given time are called *video object planes* (VOPs). The encoding process generates a coded representation of a VOP as well as composition information necessary for display. Further, at the decoder, a user may interact with and modify the composition process as needed.

The full syntax allows coding of rectangular as well as arbitrarily shaped video objects in a scene. Furthermore, the syntax supports both nonscalable coding and scalable coding. Thus it becomes possible to handle normal scalabilities as well as object based scalabilities. The scalability syntax enables the reconstruction of useful video from pieces of a total bitstream. This is achieved by structuring the total bitstream in two or more layers, starting from a standalone base layer and adding a number of enhancement layers. The base layer can be coded using a non-scalable syntax, or in the case of picture based coding, even using a syntax of a different video coding standard.

To ensure the ability to access individual objects, it is necessary to achieve a coded representation of its shape. A natural video object consists of a sequence of 2D representations (at different points in time) referred to here as VOPs. For efficient coding of VOPs, both temporal redundancies as well as spatial redundancies are exploited. Thus a coded representation of a VOP includes representation of its shape, its motion and its texture.

1.4.2 Face object

A 3D (or 2D) *face object* is a representation of the human face that is structured for portraying the visual manifestations of speech and facial expressions adequate to achieve visual speech intelligibility and the recognition of the mood of the speaker. A face object is animated by a stream of *face animation parameters (FAP)* encoded for low-bandwidth transmission in broadcast (one-to-many) or dedicated interactive (point-to-point) communications. The FAPs manipulate key feature control points in a mesh model of the face to produce animated visemes for the mouth (lips, tongue, teeth), as well as animation of the head and facial features like the eyes. FAPs are quantized with careful consideration for the limited movements of facial features, and then prediction errors are calculated and coded arithmetically. The remote manipulation of a face model in a terminal with FAPs can accomplish lifelike visual scenes of the speaker in real-time without sending pictorial or video details of face imagery every frame.

A simple streaming connection can be made to a decoding terminal that animates a default face model. A more complex session can initialize a custom face in a more capable terminal by downloading *face definition parameters (FDP)* from the encoder. Thus specific background images, facial textures, and head geometry can be portrayed. The composition of specific backgrounds, face 2D/3D meshes, texture attribution of the mesh, etc. is described in ISO/IEC part 1. The FAP stream for a given user can be generated at the user's terminal from video/audio, or from text-to-speech. FAPs can be encoded at bitrates up to 2-3kbit/s at necessary speech rates. Optional temporal DCT coding provides further compression efficiency in exchange for delay. Using the facilities of ISO/IEC part 1, a composition of the animated face model and synchronized, coded speech audio (low-bitrate speech coder or text-to-speech) can provide an integrated low-bandwidth audio/visual speaker for broadcast applications or interactive conversation.

Limited scalability is supported. Face animation achieves its efficiency by employing very concise motion animation controls in the channel, while relying on a suitably equipped terminal for rendering of moving 2D/3D faces with non-normative models held in local memory. Models stored and updated for rendering in the terminal can be simple or complex. To support speech intelligibility, the normative specification of FAPs intends for their selective or complete use as signaled by the encoder. A masking scheme provides for selective transmission of FAPs according to what parts of the face are naturally active from moment to moment. A further control in the FAP stream allows face animation to be suspended while leaving face features in the terminal in a defined quiescent state for higher overall efficiency during multi-point connections.

The Face Animation specification is defined in parts 1 and 2 of the standard. This section is intended to facilitate finding various parts of specification. As a rule of thumb, FAP specification is found in the part 2, and FDP specification in the part 1. However, this is not a strict rule. For an overview of FAPs and their interpretation, read sections "6.1.5.2 Facial animation parameter set", "6.1.5.3 Facial animation parameter units", "6.1.5.4 Description of a neutral face" as well as the table 12-1 (annex C) of part 2. The viseme parameter is documented in section "7.12.3 Decoding of the viseme parameter fap 1" and the table 12-5 (annex C) of part 2. The expression parameter is documented in section "7.12.4 Decoding of the expression parameter fap 2" and the table 12-3 of part 2. FAP bitstream syntax is found in section "6.2.11 Face Object", semantics in "6.3.11 Face Object", and section "7.12 Face object decoding" of part 2 explains in more detail the FAP decoding process. FAP masking and interpolation is explained in sections "6.3.11.1 Face Object Plane", "7.12.1.1 Decoding of faps", "7.12.5 Fap masking" of part 2. The FIT interpolation scheme is documented in section "7.2.5.3.2.4 FIT" of part 1. The FDPs and their interpretation are documented in the section "7.2.5.3.2.6 FDP" of part 1. In particular, the FDP feature points are documented in the figure 12-1 in the annex C of part 2.

1.4.3 Mesh object

A 2D *mesh object* is a representation of a 2D deformable geometric shape, with which synthetic video objects may be created during a composition process at the decoder, by spatially piece-wise warping of existing video object planes or still texture objects. The instances of mesh objects at a given time are called *mesh object planes* (mops). The geometry of mesh object planes is coded losslessly. Temporally and spatially predictive techniques and variable length coding are used to compress 2D mesh geometry. The coded representation of a 2D mesh object includes representation of its geometry and motion.

1.4.4 Overview of the object based nonscalable syntax

The coded representation defined in the non-scalable syntax achieves a high compression ratio while preserving good image quality. Further, when access to individual objects is desired, the shape of objects also needs to be coded, and depending on the bandwidth available, the shape information can be coded lossy or losslessly.

The compression algorithm employed for texture data is not lossless as the exact sample values are not preserved during coding. Obtaining good image quality at the bitrates of interest demands very high compression, which is not achievable with intra coding alone. The need for random access, however, is best satisfied with pure intra coding. The choice of the techniques is based on the need to balance a high image quality and compression ratio with the requirement to make random access to the coded bitstream.

A number of techniques are used to achieve high compression. The algorithm first uses block-based motion compensation to reduce the temporal redundancy. Motion compensation is used both for causal prediction of the current VOP from a previous VOP, and for non-causal, interpolative prediction from past and future VOPs. Motion vectors are defined for each 16-sample by 16-line region of a VOP or 8-sample by 8-line region of a VOP as required. The prediction error, is further compressed using the discrete cosine transform (DCT) to remove spatial correlation before it is quantised in an irreversible process that discards the less important information. Finally, the shape information, motion vectors and the quantised DCT information, are encoded using variable length codes.

1.4.4.1 Temporal processing

Because of the conflicting requirements of random access to and highly efficient compression, three main VOP types are defined. Intra coded VOPs (I-VOPs) are coded without reference to other pictures. They provide access points to the coded sequence where decoding can begin, but are coded with only moderate compression. Predictive coded VOPs (P-VOPs) are coded more efficiently using motion compensated prediction from a past intra or predictive coded VOPs and are generally used as a reference for further prediction. Bidirectionally-predictive coded VOPs (B-VOPs) provide the highest degree of compression but require both past and future reference VOPs for motion compensation. Bidirectionally-predictive coded VOPs are never used as references for prediction (except in the case that the resulting VOP is used as a reference for scalable enhancement layer). The organisation of the three VOP types in a sequence is very flexible. The choice is left to the encoder and will depend on the requirements of the application.

1.4.4.2 Coding of Shapes

In natural video scenes, VOPs are generated by segmentation of the scene according to some semantic meaning. For such scenes, the shape information is thus binary (binary shape). Shape information is also referred to as alpha plane. The binary alpha plane is coded on a macroblock basis by a coder which uses the context information, motion compensation and arithmetic coding.

For coding of shape of a VOP, a bounding rectangle is first created and is extended to multiples of 16×16 blocks with extended alpha samples set to zero. Shape coding is then initiated on a 16×16 block basis; these blocks are also referred to as binary alpha blocks.

1.4.4.3 Motion representation - macroblocks

The choice of 16×16 blocks (referred to as macroblocks) for the motion-compensation unit is a result of the trade-off between the coding gain provided by using motion information and the overhead needed to represent it. Each macroblock can further be subdivided to 8×8 blocks for motion estimation and compensation depending on the overhead that can be afforded.

Depending on the type of the macroblock, motion vector information and other side information is encoded with the compressed prediction error in each macroblock. The motion vectors are differenced with respect to a prediction value and coded using variable length codes. The maximum length of the motion vectors allowed is decided at the encoder. It is the responsibility of the encoder to calculate appropriate motion vectors. The specification does not specify how this should be done.

1.4.4.4 Spatial redundancy reduction

Both source VOPs and prediction errors VOPs have significant spatial redundancy. This specification uses a block-based DCT method with optional visually weighted quantisation, and run-length coding. After motion compensated prediction or interpolation, the resulting prediction error is split into 8×8 blocks. These are transformed into the DCT domain where they can be weighted before being quantised. After quantisation many of the DCT coefficients are zero in value and so two-dimensional run-length and variable length coding is used to encode the remaining DCT coefficients efficiently.

1.4.4.5 Chrominance formats

This specification currently supports the 4:2:0 chrominance format.

1.4.4.6 Pixel depth

This specification suppports pixel depths between 4 and 12 bits in luminance and chrominance planes.

1.4.5 Generalized scalability

The scalability tools in this specification are designed to support applications beyond that supported by single layer video. The major applications of scalability include internet video, wireless video, multi-quality video services, video database browsing etc. In some of these applications, either normal scalabilities on picture basis such as those in ISO/IEC 13818-2 may be employed or object based scalabilities may be necessary; both categories of scalability are enabled by this specification.

Although a simple solution to scalable video is the simulcast technique that is based on transmission/storage of multiple independently coded reproductions of video, a more efficient alternative is scalable video coding, in which the bandwidth allocated to a given reproduction of video can be partially re-utilised in coding of the next reproduction of video. In scalable video coding, it is assumed that given a coded bitstream, decoders of various complexities can decode and display appropriate reproductions of coded video. A scalable video encoder is likely to have increased complexity when compared to a single layer encoder. However, this standard provides several different forms of scalabilities that address non-overlapping applications with corresponding complexities.

The basic scalability tools offered are *temporal scalability* and *spatial scalability*. Moreover, combinations of these basic scalability tools are also supported and are referred to as *hybrid scalability*. In the case of basic scalability, two layers of video referred to as the *lower layer* and the *enhancement layer* are allowed, whereas in hybrid scalability up to four layers are supported.

1.4.5.1 Object based Temporal scalability

Temporal scalability is a tool intended for use in a range of diverse video applications from video databases, internet video, wireless video and multiview/stereoscopic coding of video. Furthermore, it may also provide a migration path from current lower temporal resolution video systems to higher temporal resolution systems of the future.

Temporal scalability involves partitioning of VOPs into layers, where the lower layer is coded by itself to provide the basic temporal rate and the enhancement layer is coded with temporal prediction with respect to the lower layer. These layers when decoded and temporally multiplexed yield full temporal resolution. The lower temporal resolution systems may only decode the lower layer to provide basic temporal resolution whereas enhanced systems of the future may support both layers. Furthermore, temporal scalability has use in bandwidth constrained networked applications where adaptation to frequent changes in allowed throughput are necessary. An additional advantage of temporal scalability is its ability to provide resilience to transmission errors as the more important data of the lower layer can be sent over a channel with better error performance, whereas the less critical enhancement layer can be sent over a channel with poor error performance. Object based temporal scalability can also be employed to allow graceful control of picture quality by controlling the temporal rate of each video object under the constraint of a given bit-budget.

1.4.5.2 Object based Spatial scalability

Spatial scalability is a tool intended for use in video applications involving multi quality video services, video database browsing, internet video and wireless video, i.e., video systems with the primary common feature that a minimum of two layers of spatial resolution are necessary. Spatial scalability involves generating two spatial resolution video layers from a single video source such that the lower layer is coded by itself to provide the basic spatial resolution and the enhancement layer employs the spatially interpolated lower layer and carries the full spatial resolution of the input video source.

An additional advantage of spatial scalability is its ability to provide resilience to transmission errors as the more important data of the lower layer can be sent over a channel with better error performance, whereas the less critical enhancement layer data can be sent over a channel with poor error performance. Further, it can also allow interoperability between various standards. Object based spatial scalability can allow better bit budgeting, complexity scalability and ease of decoding.

1.4.5.3 Hybrid scalability

There are a number of applications where neither the temporal scalability nor the spatial scalability may offer the necessary flexibility and control. This may necessitate use of temporal and spatial scalability simultaneously and is referred to as the hybrid scalability. Among the applications of hybrid scalability are wireless video, internet video, multiviewpoint/stereoscopic coding etc.

1.5 Error Resilience

ISO/IEC 14496-2 provides error robustness and resilience to allow accessing of image or video information over a wide range of storage and transmission media. The error resilience tools developed for ISO/IEC 14496-2 can be divided into three major categories. These categories include synchronization, data recovery, and error concealment. It should be noted that these categories are not unique to ISO/IEC 14496-2, and have been used elsewhere in general research in this area. It is, however, the tools contained in these categories that are of interest, and where ISO/IEC 14496-2 makes its contribution to the problem of error resilience.

COMMITTEE DRAFT OF ISO/IEC 14496-2

INFORMATION TECHNOLOGY -
CODING OF AUDIO-VISUAL OBJECTS: VIDEO

1. Scope

This committee draft of International Standard specifies the coded representation of picture information in the form of natural or synthetic visual objects like video sequences of rectangular or arbitrarily shaped pictures, moving 2D meshes, animated 3D face models and texture for synthetic objects. The coded representation allows for content based access for digital storage media, digital video communication and other applications. The International Standard specifies also the decoding process of the aforementioned coded representation. The representation supports constant bitrate transmission, variable bitrate transmission, robust transmission, content based random access (including normal random access), object based scalable decoding (including normal scalable decoding), object based bitstream editing, as well as special functions such as fast forward playback, fast reverse playback, slow motion, pause and still pictures. Synthetic objects and coding of special 2D/3D meshes, texture, and animation parameters are provided for use with downloadable models to exploit mixed media and the bandwidth improvement associated with remote manipulation of such models. This International Standard is intended to allow some level of interoperability with ISO/IEC 11172-2, ISO/IEC 13818-2 and ITU-T H.263.

2. Normative references

The following ITU-T Recommendations and International Standards contain provisions which through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All Recommendations and Standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards. The Telecommunication Standardisation Bureau maintains a list of currently valid ITU-T Recommendations.

- Recommendations and reports of the CCIR, 1990 XVIIth Plenary Assembly, Dusseldorf, 1990 Volume XI - Part 1 Broadcasting Service (Television) Recommendation ITU-R BT.601-3 “Encoding parameters of digital television for studios”.

- CCIR Volume X and XI Part 3 Recommendation ITU-R BR.648 “Recording of audio signals”.

- CCIR Volume X and XI Part 3 Report ITU-R 955-2 “Satellite sound broadcasting to vehicular, portable and fixed receivers in the range 500 - 3000Mhz”.

- ISO/IEC 11172-1 1993, *Information technology — Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s — Part 1: Systems.*

- ISO/IEC 11172-2 1993, *Information technology — Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s — Part 2: Video.*

- ISO/IEC 11172-3 1993, *Information technology — Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s — Part 3: Audio.*

- ISO/IEC 13818-1 1995, *Information technology — Generic Coding of moving pictures and associated audio — Part 1: Systems.*

- ISO/IEC 13818-2 1995, *Information technology — Generic Coding of moving pictures and associated audio— Part 2: Video.*

- ISO/IEC 13818-3 1995, *Information technology — Generic Coding of moving pictures and associated audio — Part 3: Audio.*

- IEEE Standard Specifications for the Implementations of 8 by 8 Inverse Discrete Cosine Transform, IEEE Std 1180-1990, December 6, 1990.

- IEC Publication 908:1987, *CD Digital Audio System*.

- IEC Publication 461:1986, *Time and control code for video tape recorder.*

- ITU-T Recommendation H.261 (Formerly CCITT Recommendation H.261) Codec for audiovisual services at px64 kbit/s Geneva, 1990.

- ITU-T Recommendation H.263 Video Coding for Low Bitrate Communication Geneva, 1996.

- ISO/IEC 10918-1:1994 | Recommendation ITU-T T.81 (JPEG) *Information Technology — Digital compression and coding of continuous-tone still images: Requirements and guidelines.*

3. Definitions

For the purposes of this International Standard, the following definitions apply.

3.1 **AC coefficient**: Any DCT coefficient for which the frequency in one or both dimensions is non-zero.

3.2 **B-VOP**; **bidirectionally predictive-coded video object plane (VOP)**: A VOP that is coded using motion compensated prediction from past and/or future reference VOPs

3.3 **backward compatibility**: A newer coding standard is backward compatible with an older coding standard if decoders designed to operate with the older coding standard are able to continue to operate by decoding all or part of a bitstream produced according to the newer coding standard.

3.4 **backward motion vector**: A motion vector that is used for motion compensation from a reference VOP at a later time in display order.

3.5 **backward prediction**: Prediction from the future reference VOP

3.6 **base layer:** An independently decodable layer of a scalable hierarchy

3.7 **binary alpha block:** A block of size 16x16 pels, colocated with macroblock, representing shape information of the binary alpha map; it is also referred to as a bab.

3.8 **binary alpha map:** A 2D binary mask used to represent the shape of a video object such that the pixels that are opaque are considered as part of the object where as pixels that are transparent are not considered to be part of the object.

3.9 **bitstream; stream**: An ordered series of bits that forms the coded representation of the data.

3.10 **bitrate**: The rate at which the coded bitstream is delivered from the storage medium or network to the input of a decoder.

3.11 **block**: An 8-row by 8-column matrix of samples, or 64 DCT coefficients (source, quantised or dequantised).

3.12 **byte aligned**: A bit in a coded bitstream is byte-aligned if its position is a multiple of 8-bits from the first bit in the stream.

3.13 **byte**: Sequence of 8-bits.

3.14 **context based arithmetic encoding:** The method used for coding of binary shape; it is also referred to as cae.

3.15 **channel**: A digital medium or a network that stores or transports a bitstream constructed according to this specification.

3.16 **chrominance format**: Defines the number of chrominance blocks in a macroblock.

3.17 **chrominance component**: A matrix, block or single sample representing one of the two colour difference signals related to the primary colours in the manner defined in the bitstream. The symbols used for the chrominance signals are Cr and Cb.

3.18 **coded B-VOP**: A B-VOP that is coded.

3.19 **coded VOP**: A coded VOP is a coded I-VOP, a coded P-VOP or a coded B-VOP.

3.20 **coded I-VOP**: An I-VOP that is coded.

3.21 **coded P-VOP**: A P-VOP that is coded.

3.22 **coded video bitstream**: A coded representation of a series of one or more VOPs as defined in this specification.

3.23 **coded order**: The order in which the VOPs are transmitted and decoded. This order is not necessarily the same as the display order.

3.24 **coded representation**: A data element as represented in its encoded form.

3.25 **coding parameters**: The set of user-definable parameters that characterise a coded video bitstream. Bitstreams are characterised by coding parameters. Decoders are characterised by the bitstreams that they are capable of decoding.

3.26 **component**: A matrix, block or single sample from one of the three matrices (luminance and two chrominance) that make up a picture.

3.27 **composition process**: The (non-normative) process by which reconstructed VOPs are composed into a scene and displayed.

3.28 **compression**: Reduction in the number of bits used to represent an item of data.

3.29 **constant bitrate coded video**: A coded video bitstream with a constant bitrate.

3.30 **constant bitrate**: Operation where the bitrate is constant from start to finish of the coded bitstream.

3.31 **conversion ratio:** The size conversion ratio for the purpose of rate control of shape.

3.32 **data element**: An item of data as represented before encoding and after decoding.

3.33 **DC coefficient**: The DCT coefficient for which the frequency is zero in both dimensions.

3.34 **DCT coefficient**: The amplitude of a specific cosine basis function.

3.35 **decoder input buffer**: The first-in first-out (FIFO) buffer specified in the video buffering verifier.

3.36 **decoder**: An embodiment of a decoding process.

3.37 **decoding (process)**: The process defined in this specification that reads an input coded bitstream and produces decoded VOPs or audio samples.

3.38 **dequantisation**: The process of rescaling the quantised DCT coefficients after their representation in the bitstream has been decoded and before they are presented to the inverse DCT.

3.39 **digital storage media; DSM**: A digital storage or transmission device or system.

3.40 **discrete cosine transform; DCT**: Either the forward discrete cosine transform or the inverse discrete cosine transform. The DCT is an invertible, discrete orthogonal transformation. The inverse DCT is defined in Annex A of this specification.

3.41 **display order**: The order in which the decoded pictures are displayed. Normally this is the same order in which they were presented at the input of the encoder.

3.42 **editing**: The process by which one or more coded bitstreams are manipulated to produce a new coded bitstream. Conforming edited bitstreams must meet the requirements defined in this specification.

3.43 **encoder**: An embodiment of an encoding process.

3.44 **encoding (process)**: A process, not specified in this specification, that reads a stream of input pictures or audio samples and produces a valid coded bitstream as defined in this specification.

3.45 **enhancement layer**: A relative reference to a layer (above the base layer) in a scalable hierarchy. For all forms of scalability, its decoding process can be described by reference to the lower layer decoding process and the appropriate additional decoding process for the enhancement layer itself.

3.46 **face animation parameter units, FAPU:** Special normalized units (e.g. translational, angular, logical) defined to allow interpretation of FAPs with any facial model in a consistent way to produce reasonable results in expressions and speech pronunciation.

3.47 **face animation parameters, FAP:** Coded streaming animation parameters that manipulate the displacements and angles of face features, and that govern the blending of visemes and face expressions during speech.

3.48 **face animation table, FAT:** A downloadable function mapping from incoming FAPs to feature control points in the face mesh that provides piecewise linear weightings of the FAPs for controlling face movements.

3.49 **face calibration mesh:** Definition of a 3D mesh for calibration of the shape and structure of a baseline face model.

3.50 **face definition parameters, FDP:** Downloadable data to customize a baseline face model in the decoder to a particular face, or to download a face model along with the information about how to animate it. The FDPs are normally transmitted once per session, followed by a stream of compressed FAPs. FDPs may include feature points for calibrating a baseline face, face texture and coordinates to map it onto the face, animation tables, etc.

3.51 **face feature control point:** A normative vertex point in a set of such points that define the critical locations within face features for control by FAPs and that allow for calibration of the shape of the baseline face.

3.52 **face interpolation transform, FIT:** A downloadable node type defined in ISO/IEC 14496-1 for optional mapping of incoming FAPs to FAPs before their application to feature points, through weighted rational polynomial functions, for complex cross-coupling of standard FAPs to link their effects into custom or proprietary face models.

3.53 **face model mesh:** A 2D or 3D contiguous geometric mesh defined by vertices and planar polygons utilizing the vertex coordinates, suitable for rendering with photometric attributes (e.g. texture, color, normals).

3.54 **feathering**: A tool that tapers the values around edges of binary alpha mask for composition with the background.

3.55 **flag**: A one bit integer variable which may take one of only two values (zero and one).

3.56 **forbidden**: The term “forbidden” when used in the clauses defining the coded bitstream indicates that the value shall never be used. This is usually to avoid emulation of start codes.

3.57 **forced updating**: The process by which macroblocks are intra-coded from time-to-time to ensure that mismatch errors between the inverse DCT processes in encoders and decoders cannot build up excessively.

3.58 **forward compatibility**: A newer coding standard is forward compatible with an older coding standard if decoders designed to operate with the newer coding standard are able to decode bitstreams of the older coding standard.

3.59 **forward motion vector**: A motion vector that is used for motion compensation from a reference frame VOP at an earlier time in display order.

3.60 **forward prediction**: Prediction from the past reference VOP.

3.61 **frame**: A frame contains lines of spatial information of a video signal. For progressive video, these lines contain samples starting from one time instant and continuing through successive lines to the bottom of the frame.

3.62 **frame period**: The reciprocal of the frame rate.

3.63 **frame rate**: The rate at which frames are be output from the composition process.

3.64 **future reference VOP**: A future reference VOP is a reference VOP that occurs at a later time than the current VOP in display order.

3.65 **VOP reordering**: The process of reordering the reconstructed VOPs when the coded order is different from the composition order for display. VOP reordering occurs when B-VOPs are present in a bitstream. There is no VOP reordering when decoding low delay bitstreams.

3.66 **hybrid scalability:** Hybrid scalability is the combination of two (or more) types of scalability.

3.67 **interlace**: The property of conventional television frames where alternating lines of the frame represent different instances in time. In an interlaced frame, one of the field is meant to be displayed first. This field is called the first field. The first field can be the top field or the bottom field of the frame.

3.68 **I-VOP; intra-coded VOP**: A VOP coded using information only from itself.

3.69 **intra coding**: Coding of a macroblock or VOP that uses information only from that macroblock or VOP.

3.70 **intra shape coding**: Shape coding that does not use any temporal prediction.

3.71 **inter shape coding**: Shape coding that uses temporal prediction.

3.72 **level**: A defined set of constraints on the values which may be taken by the parameters of this specification within a particular profile. A profile may contain one or more levels. In a different context, level is the absolute value of a non-zero coefficient (see “run”).

3.73 **layer**: In a scalable hierarchy denotes one out of the ordered set of bitstreams and (the result of) its associated decoding process.

3.74 **layered bitstream**: A single bitstream associated to a specific layer (always used in conjunction with layer qualifiers, e. g. “enhancement layer bitstream”)

3.75 **lower layer**: A relative reference to the layer immediately below a given enhancement layer (implicitly including decoding of *all* layers below this enhancement layer)

3.76 **luminance component**: A matrix, block or single sample representing a monochrome representation of the signal and related to the primary colours in the manner defined in the bitstream. The symbol used for luminance is Y.

3.77 **Mbit**: 1 000 000 bits

3.78 **macroblock**: The four 8×8 blocks of luminance data and the two (for 4:2:0 chrominance format) corresponding 8×8 blocks of chrominance data coming from a 16×16 section of the luminance component of the picture. Macroblock is sometimes used to refer to the sample data and sometimes to the coded representation of the sample values and other data elements defined in the macroblock header of the syntax defined in this part of this specification. The usage is clear from the context.

3.79 **mesh**: A 2D triangular mesh refers to a planar graph which tessellates a video object plane into triangular patches. The vertices of the triangular mesh elements are referred to as node points. The straight-line segments between node points are referred to as edges. Two triangles are adjacent if they share a common edge.

3.80 **mesh geometry**: The spatial locations of the node points and the triangular structure of a mesh.

3.81 **mesh motion**: The temporal displacements of the node points of a mesh from one time instance to the next.

3.82 **motion compensation**: The use of motion vectors to improve the efficiency of the prediction of sample values. The prediction uses motion vectors to provide offsets into the past and/or future reference VOPs containing previously decoded sample values that are used to form the prediction error.

3.83 **motion estimation**: The process of estimating motion vectors during the encoding process.

3.84 **motion vector**: A two-dimensional vector used for motion compensation that provides an offset from the coordinate position in the current picture or field to the coordinates in a reference VOP.

3.85 **motion vector for shape**: A motion vector used for motion compensation of shape.

3.86 **non-intra coding**: Coding of a macroblock or a VOP that uses information both from itself and from macroblocks and VOPs occurring at other times.

3.87 **opaque macroblock**: A macroblock with shape mask of all 255's.

3.88 **P-VOP; predictive-coded VOP**: A picture that is coded using motion compensated prediction from the past VOP.

3.89 **parameter**: A variable within the syntax of this specification which may take one of a range of values. A variable which can take one of only two values is called a flag.

3.90 **past reference picture**: A pas.t reference VOP is a reference VOP that occurs at an earlier time than the current VOP in composition order.

3.91 **picture**: Source, coded or reconstructed image data. A source or reconstructed picture consists of three rectangular matrices of 8-bit numbers representing the luminance and two chrominance signals. A "coded VOP" was defined earlier. For progressive video, a picture is identical to a frame.

3.92 **prediction**: The use of a predictor to provide an estimate of the sample value or data element currently being decoded.

3.93 **prediction error**: The difference between the actual value of a sample or data element and its predictor.

3.94 **predictor**: A linear combination of previously decoded sample values or data elements.

3.95 **profile**: A defined subset of the syntax of this specification.

3.96 **progressive**: The property of film frames where all the samples of the frame represent the same instances in time.

3.97 **quantisation matrix**: A set of sixty-four 8-bit values used by the dequantiser.

3.98 **quantised DCT coefficients**: DCT coefficients before dequantisation. A variable length coded representation of quantised DCT coefficients is transmitted as part of the coded video bitstream.

3.99 **quantiser scale**: A scale factor coded in the bitstream and used by the decoding process to scale the dequantisation.

3.100 **random access**: The process of beginning to read and decode the coded bitstream at an arbitrary point.

3.101 **reconstructed VOP**: A reconstructed VOP consists of three matrices of 8-bit numbers representing the luminance and two chrominance signals. It is obtained by decoding a coded VOP

3.102 **reference VOP**: A reference frame is a reconstructed VOP that was coded in the form of a coded I-VOP or a coded P-VOP. Reference VOPs are used for forward and backward prediction when P-VOPs and B-VOPs are decoded.

3.103 **reordering delay**: A delay in the decoding process that is caused by VOP reordering.

3.104 **reserved**: The term “reserved” when used in the clauses defining the coded bitstream indicates that the value may be used in the future for ISO/IEC defined extensions.

3.105 **scalable hierarchy**: coded video data consisting of an ordered set of more than one video bitstream.

3.106 **scalability**: Scalability is the ability of a decoder to decode an ordered set of bitstreams to produce a reconstructed sequence. Moreover, useful video is output when subsets are decoded. The minimum subset that can thus be decoded is the first bitstream in the set which is called the base layer. Each of the other bitstreams in the set is called an enhancement layer. When addressing a specific enhancement layer, “lower layer” refers to the bitstream that precedes the enhancement layer.

3.107 **side information**: Information in the bitstream necessary for controlling the decoder.

3.108 **run**: The number of zero coefficients preceding a non-zero coefficient, in the scan order. The absolute value of the non-zero coefficient is called “level”.

3.109 **saturation**: Limiting a value that exceeds a defined range by setting its value to the maximum or minimum of the range as appropriate.

3.110 **source**; **input**: Term used to describe the video material or some of its attributes before encoding.

3.111 **spatial prediction**: prediction derived from a decoded frame of the lower layer decoder used in spatial scalability

3.112 **spatial scalability**: A type of scalability where an enhancement layer also uses predictions from sample data derived from a lower layer without using motion vectors. The layers can have different VOP sizes or VOP rates.

3.113 **static sprite:** The luminance, chrominance and binary alpha plane for an object which does not vary in time.

3.114 **sprite-VOP:** A picture that is coded using information obtained by warping whole or part of a static sprite.

3.115 **start codes**: 32-bit codes embedded in that coded bitstream that are unique. They are used for several purposes including identifying some of the structures in the coding syntax.

3.116 **stuffing (bits)**; **stuffing (bytes)**: Code-words that may be inserted into the coded bitstream that are discarded in the decoding process. Their purpose is to increase the bitrate of the stream which would otherwise be lower than the desired bitrate.

3.117 **temporal prediction**: prediction derived from reference VOPs other than those defined as spatial prediction

3.118 **temporal scalability**: A type of scalability where an enhancement layer also uses predictions from sample data derived from a lower layer using motion vectors. The layers have identical frame size, and but can have different VOP rates.

3.119 **top layer**: the topmost layer (with the highest layer_id) of a scalable hierarchy.

3.120 **transparent macroblock**: A macroblock with shape mask of all zeros.

3.121 **variable bitrate**: Operation where the bitrate varies with time during the decoding of a coded bitstream.

3.122 **variable length coding**; **VLC**: A reversible procedure for coding that assigns shorter code-words to frequent events and longer code-words to less frequent events.

3.123 **video buffering verifier**; **VBV**: A hypothetical decoder that is conceptually connected to the output of the encoder. Its purpose is to provide a constraint on the variability of the data rate that an encoder or editing process may produce.

3.124 **video session**: The highest syntactic structure of coded video bitstreams. It contains a series of one or more coded video objects.

3.125 **viseme**: the physical (visual) configuration of the mouth, tongue and jaw that is visually correlated with the speech sound corresponding to a phoneme.

3.126 **warping**: Processing applied to extract a sprite VOP from a static sprite. It consists of a global spatial transformation driven by a few motion parameters (0,2,4,8), to recover luminance, chrominance and shape information.

3.127 **zigzag scanning order**: A specific sequential ordering of the DCT coefficients from (approximately) the lowest spatial frequency to the highest.

4. Abbreviations and symbols

The mathematical operators used to describe this specification are similar to those used in the C programming language. However, integer divisions with truncation and rounding are specifically defined. Numbering and counting loops generally begin from zero.

4.1 Arithmetic operators

+	Addition.
-	Subtraction (as a binary operator) or negation (as a unary operator).
++	Increment. i.e. *x*++ is equivalent to *x* = *x* + 1
- -	Decrement. i.e. *x*-- is equivalent to *x* = *x* - 1
$\left.\begin{matrix}\times \\ *\end{matrix}\right\}$	Multiplication.
^	Power.
/	Integer division with truncation of the result toward zero. For example, 7/4 and -7/-4 are truncated to 1 and -7/4 and 7/-4 are truncated to -1.
//	Integer division with rounding to the nearest integer. Half-integer values are rounded away from zero unless otherwise specified. For example 3//2 is rounded to 2, and -3//2 is rounded to -2.
///	Integer division with sign dependent rounding to the nearest integer. Half-integer values when positive are rounded away from zero, and when negative are rounded towards zero. For example 3///2 is rounded to 2, and -3///2 is rounded to -1.
////	Integer division with truncation towards the negative infinity.
÷	Used to denote division in mathematical equations where no truncation or rounding is intended.
%	Modulus operator. Defined only for positive numbers.
Sign( )	$\mathrm{Sign}(x) = \begin{cases} 1 & x >= 0 \\ -1 & x < 0 \end{cases}$
Abs( )	$\mathrm{Abs}(x) = \begin{cases} x & x >= 0 \\ -x & x < 0 \end{cases}$
$\sum_{i=a}^{i<b} f(i)$	The summation of the *f*(*i*) with *i* taking integral values from *a* up to, but not including *b*.

4.2 Logical operators

||	Logical OR.
&&	Logical AND.
!	Logical NOT.

4.3 Relational operators

>	Greater than.
>=	Greater than or equal to.
<	Less than.
<=	Less than or equal to.
==	Equal to.
!=	Not equal to.
max [, … ,]	the maximum value in the argument list.
min [, … ,]	the minimum value in the argument list.

4.4 Bitwise operators

&	AND
|	OR
>>	Shift right with sign extension.
<<	Shift left with zero fill.

4.5 Conditional operators

?:

$$(condition ?\ a : b) = \begin{cases} a & \text{if } condition \text{ is true,} \\ b & \text{otherwise.} \end{cases}$$

4.6 Assignment

=	Assignment operator.

4.7 Mnemonics

The following mnemonics are defined to describe the different data types used in the coded bitstream.

bslbf	Bit string, left bit first, where “left” is the order in which bit strings are written in this specification. Bit strings are generally written as a string of 1s and 0s within single quote marks, e.g. ‘1000 0001’. Blanks within a bit string are for ease of reading and have no significance. For convenience large strings are occasionally written in hexadecimal, in this case conversion to a binary in the conventional manner will yield the value of the bit string. Thus the left most hexadecimal digit is first and in each hexadecimal digit the most significant of the four bits is first.
uimsbf	Unsigned integer, most significant bit first.
simsbf	Signed integer, in twos complement format, most significant (sign) bit first.
vlclbf	Variable length code, left bit first, where “left” refers to the order in which the VLC codes are written. The byte order of multibyte words is most significant byte first.

4.8 Constants

P	3,141 592 653 58…
e	2,718 281 828 45…

5. Conventions

5.1 Method of describing bitstream syntax

The bitstream retrieved by the decoder is described in 6.2. Each data item in the bitstream is in bold type. It is described by its name, its length in bits, and a mnemonic for its type and order of transmission.

The action caused by a decoded data element in a bitstream depends on the value of that data element and on data elements previously decoded. The decoding of the data elements and definition of the state variables used in their decoding are described in 6.3. The following constructs are used to express the conditions when data elements are present, and are in normal type:

while (condition) {
    data_element
    . . .
}
	If the condition is true, then the group of data elements occurs next in the data stream. This repeats until the condition is not true.

do {
    data_element
    . . .
} while (condition)
	The data element always occurs at least once.
	
	The data element is repeated until the condition is not true.

if (condition) {
    data_element
    . . .
} else {
    data_element
    . . .
}
	If the condition is true, then the first group of data elements occurs next in the data stream.
	
	If the condition is not true, then the second group of data elements occurs next in the data stream.

for (i = m; i < n; i++) {
    data_element
    . . .
}
	The group of data elements occurs (n-m) times. Conditional constructs within the group of data elements may depend on the value of the loop control variable i, which is set to m for the first occurrence, incremented by one for the second occurrence, and so forth.

/* comment … */	Explanatory comment that may be deleted entirely without in any way altering the syntax.

This syntax uses the ‘C-code’ convention that a variable or expression evaluating to a non-zero value is equivalent to a condition that is true and a variable or expression evaluating to a zero value is equivalent to a condition that is false. In many cases a literal string is used in a condition. For example;

```
if ( video_object_layer_shape == “rectangular” ) …
```

In such cases the literal string is that used to describe the value of the bitstream element in 6.3. In this example, we see that “rectangular” is defined in a Table?? to be represented by the two bit binary number ‘00’.

As noted, the group of data elements may contain nested conditional constructs. For compactness, the brackets { } are omitted when only one data element follows.

data_element [n]	data_element [n] is the n+1th element of an array of data.
data_element [m][n]	data_element [m][n] is the m+1, n+1th element of a two-dimensional array of data.
data_element [l][m][n]	data_element [l][m][n] is the l+1, m+1, n+1th element of a three-dimensional array of data.

While the syntax is expressed in procedural terms, it should not be assumed that 6.2 implements a satisfactory decoding procedure. In particular, it defines a correct and error-free input bitstream. Actual decoders must include means to look for start codes in order to begin decoding correctly, and to identify errors, erasures or insertions while decoding. The methods to identify these situations, and the actions to be taken, are not standardised.

5.2 Definition of functions

Several utility functions for picture coding algorithm are defined as follows:

5.2.1 Definition of bytealigned() function

The function bytealigned () returns 1 if the current position is on a byte boundary, that is the next bit in the bitstream is the first bit in a byte. Otherwise it returns 0. . If the 8 bits following the next byte alignment position are ’01111111’, these bits shall be discarded.

5.2.2 Definition of nextbits_bytealigned() function

The function nextbits_bytealigned() permits comparison of a bit string with the next bits to be decoded in the bitstream at which the first bit is byte aligned.

5.2.3 Definition of next_start_code() function

The next_start_code() function removes any zero bit and a string of ‘1’ bits used for stuffing and locates the next start code.

next_start_code() {	**No. of bits**	**Mnemonic**
zero_bit	1	‘0’
while (!bytealigned())		
one_bit	1	‘1’
}		

This function checks whether the current position is byte aligned. If it is not, a zero stuffing bit followed by a number of one stuffing bits may be present before the start code.

5.2.4 Definition of next_resync_marker() function

The next_resync_marker() function removes any zero bit and a string of one bits used for stuffing and locates the next resync marker; it thus performs similar operation as next_start_code() but for resync_marker.

next_resync_marker() {	**No. of bits**	**Mnemonic**
zero_bit	1	‘0’
while (!bytealigned())		
one_bit	1	‘1’
}		

5.2.5 Definition of transparent_mb() function

The function transparent_mb() returns 1 if the current macroblock consists only of transparent pixels. Otherwise it returns 0.

5.2.6 Definition of transparent_block() function

The function transparent_block(j) returns 1 if the 8x8 with index j consists only of transparent pixels. Otherwise it returns 0. The index value for each block is defined in Figure 6-5.

5.3 Reserved, forbidden and marker_bit

The terms “reserved” and “forbidden” are used in the description of some values of several fields in the coded bitstream.

The term “reserved” indicates that the value may be used in the future for ISO/IEC defined extensions.

The term “forbidden” indicates a value that shall never be used (usually in order to avoid emulation of start codes).

The term “marker_bit” indicates a one bit integer in which the value zero is forbidden (and it therefore shall have the value ‘1’). These marker bits are introduced at several points in the syntax to avoid start code emulation.

5.4 Arithmetic precision

In order to reduce discrepancies between implementations of this specification, the following rules for arithmetic operations are specified.

(a) Where arithmetic precision is not specified, such as in the calculation of the IDCT, the precision shall be sufficient so that significant errors do not occur in the final integer values

(b) Where ranges of values are given by a colon, the end points are included if a bracket is present, and excluded if the ‘less than’ (<) and ‘greater than’ (>) characters are used. For example, [a : b> means from a to b, including a but excluding b.

6. Visual bitstream syntax and semantics

6.1 Structure of coded visual data

Coded visual data can be of several different types, such as video data, still texture data, 2D mesh data or facial animation parameter data.

Synthetic objects and their attribution are structured in a hierarchical manner to support both bitstream scalability and object scalability. ISO/IEC 14496-1 of the specification provides the approach to spatial-temporal scene composition including normative 2D/3D scene graph nodes and their composition supported by Binary Interchange Format Specification. At this level, synthetic and natural object composition relies on ISO/IEC 14496-1 with subsequent (non-normative) rendering performed by the application to generate specific pixel-oriented views of the models.

Coded video data consists of an ordered set of video bitstreams, called layers. If there is only one layer, the coded video data is called non-scalable video bitstream. If there are two layers or more, the coded video data is called a scalable hierarchy.

One of the layers is called base layer, and it can always be decoded independently. Other layers are called enhancement layers, and can only be decoded together with the lower layers (previous layers in the ordered set), starting with the base layer. The multiplexing of these layers is discussed in ISO/IEC 14496-1. The base layer of a scalable set of streams can be coded by other standards. The Enhancement layers shall conform to this specification. In general the visual bitstream can be thought of as a syntactic hierarchy in which syntactic structures contain one or more subordinate structures.

Visual texture, referred to herein as still texture coding, is designed for maintaining high visual quality in the transmission and rendering of texture under widely varied viewing conditions typical of interaction with 2D/3D synthetic scenes. Still texture coding provides for a multi-layer representation of luminance, color and shape. This supports progressive transmission of the texture for image build-up as it is received by a terminal. Also supported is the downloading of the texture resolution hierarchy for construction of image pyramids used by 3D graphics APIs. Quality and SNR scalability are supported by the structure of still texture coding.

Coded mesh data consists of just one non-scalable bitstream. This bitstream defines the structure and motion of the 2D mesh, the texture of the mesh has to be coded as a separate video object.

Coded face animation parameter data consists of one non-scaleable bitstream. It defines the animation of the facemodel of the decoder. Face animation data is structured as standard formats for downloadable models and their animation controls, and a single layer of compressed face animation parameters used for remote manipulation of the face model. The face is a node in a scene graph that includes face geometry ready for rendering. The shape, texture and expressions of the face are generally controlled by the bitstream containing instances of Facial Definition Parameter (FDP) sets and/or Facial Animation Parameter (FAP) sets. Upon initial or baseline construction, the face object contains a generic face with a neutral expression. This face can receive FAPs from the bitstream and be subsequently rendered to produce animation of the face. If FDPs are transmitted, the generic face is transformed into a particular face of specific shape and appearance. A downloaded face model via FDPs is a scene graph for insertion in the face node.

6.1.1 Visual object sequence

Visual object sequence is is the highest syntactic structure of the coded visual bitstream.

A visual object sequence commences with a visual_object_sequence_start_code which is followed by a one or more visual objects coded concurrently. The visual obbject sequence is terminated by a visual_object_sequence_end_code.

6.1.2 Visual object

A visual object commences with a visual_object_start_code, is followed by profile and level identification, and a visual object id, is followed by a video object, a still texture object, a mesh object, or a face object.

6.1.3 Video object

A video object commences with a video_object_start_code, and is followed by one or more video object layers.

6.1.3.1 Progressive and interlaced sequences

This specification deals with coding of both progressive and interlaced sequences although the tools of this specification can be combined with that of tools derived from ISO/IEC 13818-2 to form algorithms to code interlaced sequences.

The sequence, at the output of the decoding process, consists of a series of reconstructed VOPs separated in time and are readied for display via the compositor.

6.1.3.2 Frame

A frame consists of three rectangular matrices of integers; a luminance matrix (Y), and two chrominance matrices (Cb and Cr).

6.1.3.3 VOP

A reconstructed VOP is obtained by decoding a coded VOP. A coded VOP may have been derived from a either progressive or interlaced frame.

6.1.3.4 VOP types

There are four types of VOPs that use different coding methods:

1. An Intra-coded (I) VOP is coded using information only from itself.

2. A Predictive-coded (P) VOP is a VOP which is coded using motion compensated prediction from a past reference VOP.

3. A Bidirectionally predictive-coded (B) VOP is a VOP which is coded using motion compensated prediction from a past and/or future reference VOP(s).

4. A sprite (S) VOP is a VOP for a sprite object.

6.1.3.5 I-VOPs and group of VOPs

I-VOPs are intended to assist random access into the sequence. Applications requiring random access, fast-forward playback, or fast reverse playback may use I-VOPs relatively frequently.

I-VOPs may also be used at scene cuts or other cases where motion compensation is ineffective.

Group of VOP hearder is an optional header that can be used immediately before a coded I-VOP to indicate to the decoder if the first consecutive B-VOPs immediately following the coded I-frame can be reconstructed properly in the case of a random access. In the code bitstream, the first coded frame following a group of VOPs header shall be a coded I-VOP.

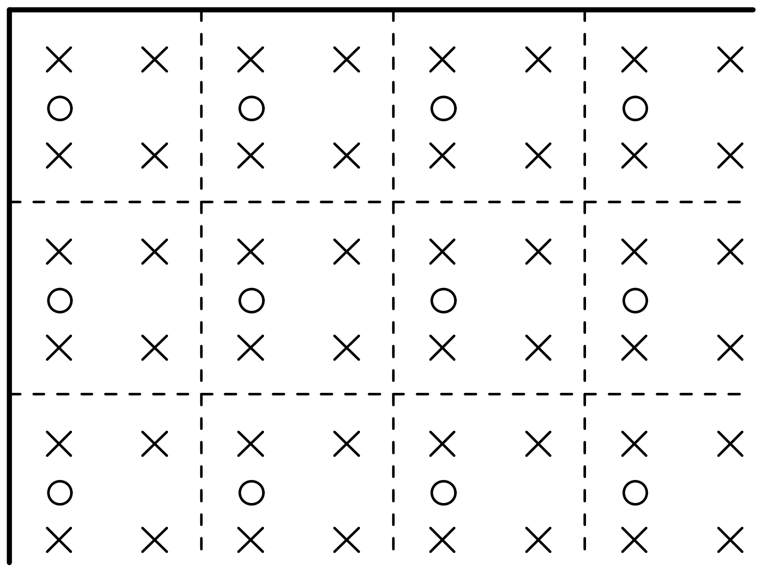
6.1.3.6 4:2:0 Format

In this format the Cb and Cr matrices shall be one half the size of the Y-matrix in both horizontal and vertical dimensions. The Y-matrix shall have an even number of lines and samples.

The luminance and chrominance samples are positioned as shown in Figure 6-1.

The two variations in the vertical and temporal positioning of the samples for interlaced VOPs are shown in Figure 6-2 and Figure 6-3.

Figure 6-4 shows the vertical and temporal positioning of the samples in a progressive frame.



× Represent luminance samples

○ Represent chrominance samples

Figure 6-1 -- The position of luminance and chrominance samples in 4:2:0 data.

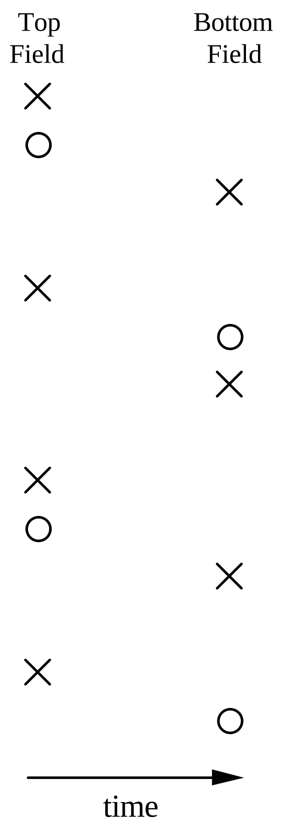


Figure 6-2 – Vertical and temporal positions of samples in an interlaced frame with top_field_first=1.

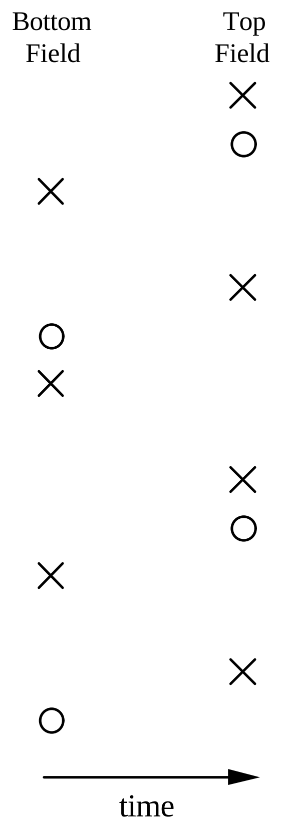


Figure 6-3 -- Vertical and temporal position of samples in an interlaced frame with top_field_first=0

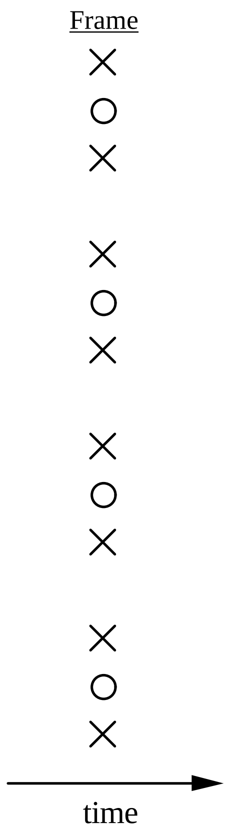


Figure 6-4– Vertical and temporal positions of samples in a progressive frame.

6.1.3.7 VOP reordering

When a video object layer contains coded B-VOPs, the number of consecutive coded B-VOPs is variable and unbounded. The first coded VOP shall not be a B-VOP.

A video object layer may contain no coded P-VOPs. A video object layer may also contain no coded I-VOPs in which case some care is required at the start of the video object layer and within the video object layer to effect both random access and error recovery.

The order of the coded VOPs in the bitstream, also called coded order, is the order in which a decoder reconstructs them. The order of the reconstructed VOPs at the output of the decoding process, also called the display order, is not always the same as the coded order and this section defines the rules of VOP reordering that shall happen within the decoding process.

When the video object layer contains no coded B-VOPs, the coded order is the same as the display order.

When B-VOPs are present in the video object layer re-ordering is performed according to the following rules:

If the current VOP in coded order is a B-VOP the output VOP is the VOP reconstructed from that B-VOP.

If the current VOP in coded order is a I-VOP or P-VOP the output VOP is the VOP reconstructed from the previous I-VOP or P-VOP if one exists. If none exists, at the start of the video object layer, no VOP is output.

The following is an example of VOPs taken from the beginning of a video object layer. In this example there are two coded B-VOPs between successive coded P-VOPs and also two coded B-VOPs between successive coded I- and P-VOPs. VOP ‘1I’ is used to form a prediction for VOP ‘4P’. VOPs ‘4P’ and ‘1I’ are both used to form predictions for VOPs ‘2B’ and ‘3B’. Therefore the order of coded VOPs in the coded sequence shall be ‘1I’, ‘4P’, ‘2B’, ‘3B’. However, the decoder shall display them in the order ‘1I’, ‘2B’, ‘3B’, ‘4P’.

At the encoder input,

1	2	3	4	5	6	7	8	9	10	11	12	13
I	B	B	P	B	B	P	B	B	I	B	B	P

At the encoder output, in the coded bitstream, and at the decoder input,

1	4	2	3	7	5	6	10	8	9	13	11	12
I	P	B	B	P	B	B	I	B	B	P	B	B

At the decoder output,

1	2	3	4	5	6	7	8	9	10	11	12	13

6.1.3.8 Macroblock

A macroblock contains a section of the luminance component and the spatially corresponding chrominance components. The term macroblock can either refer to source and decoded data or to the corresponding coded data elements. A skipped macroblock is one for which no information is transmitted. Presently there is only one chrominance format for a macroblock, namely, 4:2:0 format. The orders of blocks in a macroblock is illustrated below:

A 4:2:0 Macroblock consists of 6 blocks. This structure holds 4 Y, 1 Cb and 1 Cr Blocks and the block order is depicted in Figure 6-5.

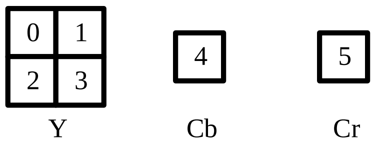


Figure 6-5 -- 4:2:0 Macroblock structure

The organisation of VOPs into macroblocks is as follows.

For the case of a progressive VOP, the interlaced flag (in the VOP header) is set to “0” and the organisation of lines of luminance VOP into macroblocks is called frame organization and is illustrated in Figure 6.4. In this case, frame DCT coding is employed.

For the case of interlaced VOP, the interlaced flag is set to “1” and the organisation of lines of luminance VOP into macroblocks can be either frame organization or field organization and thus both frame and field DCT coding may be used in the VOP.

- In the case of frame DCT coding, each luminance block shall be composed of lines from two fields alternately. This is illustrated in Figure 6-9.
- In the case of field DCT coding, each luminance block shall be composed of lines from only one of the two fields. This is illustrated in Figure 6-10.

Only frame DCT coding is applied to the chrominance blocks. It should be noted that field based predicitons may be applied for these chrominance blocks which will require predictions of 8x4 regions (after half-sample filtering).

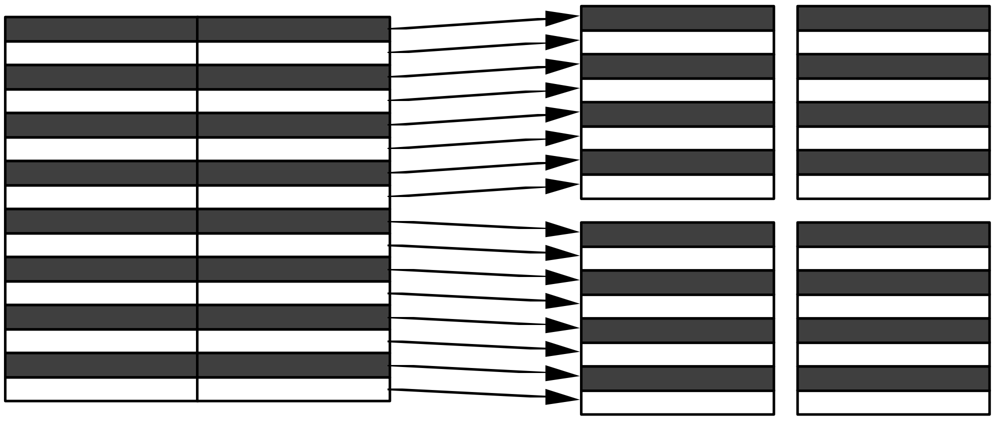


Figure 6-6 -- Luminance macroblock structure in field DCT coding

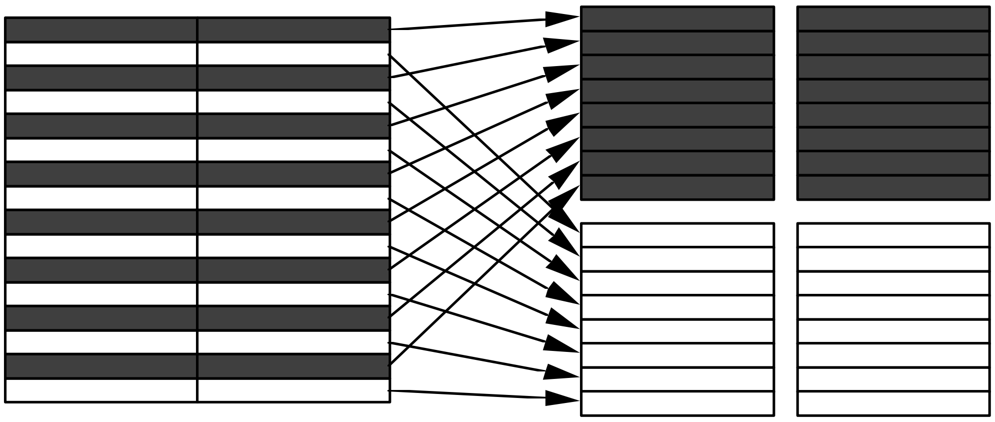


Figure 6-7 -- Luminance macroblock structure in field DCT coding

6.1.3.9 Block

The term **block** can refer either to source and reconstructed data or to the DCT coefficients or to the corresponding coded data elements.

When the block refers to source and reconstructed data it refers to an orthogonal section of a luminance or chrominance component with the same number of lines and samples. There are 8 lines and 8 samples/line in the block.

6.1.4 Mesh object

A 2D triangular *mesh* refers to a tessellation of a 2D visual object plane into triangular patches. The vertices of the triangular patches are called *node points*. The straight-line segments between node points are called *edges*. Two triangles are *adjacent* if they share a common edge.

A *dynamic* 2D mesh consists of a temporal sequence of 2D triangular meshes, where each mesh has the same topology, but node point locations may differ from one mesh to the next. Thus, a dynamic 2D mesh can be specified by the geometry of the initial 2D mesh and motion vectors at the node points for subsequent meshes, where each motion vector points from a node point of the previous mesh in the sequence to the corresponding node point of the current mesh. The dynamic 2D mesh can be used to create 2D animations by mapping texture from e.g. a video object plane onto successive 2D meshes.

A 2D dynamic mesh with *implicit structure* refers to a 2D dynamic mesh of which the initial mesh has either *uniform* or *Delaunay* topology. In both cases, the topology of the initial mesh does not have to be coded (since it is implicitly defined), only the node point locations of the initial mesh have to be coded. Note that in both the uniform and Delaunay case, the mesh is restricted to be *simple*, i.e. it consists of a single connected component without any holes, topologically equivalent to a disk.

A *mesh object* represents the geometry and motion of a 2D triangular mesh. A mesh object consists of one or more *mesh object planes*, each corresponding to a 2D triangular mesh at a certain time instance. An example of a mesh object is shown in the figure below.

A sequence of mesh object planes represents the piece-wise deformations to be applied to a video object plane or still texture object to create a synthetic animated video object. Triangular patches of a video object plane are to be warped according to the motion of corresponding triangular mesh elements. The motion of mesh elements is specified by the temporal displacements of the mesh node points.

The syntax and semantics of the mesh object pertains to the mesh geometry and mesh motion only; the video object to be used in an animation is coded separately. The warping or texture mapping applied to render visual object planes is handled in the context of scene composition. Furthermore, the syntax does not allow explicit encoding of other mesh properties such as colors or texture coordinates.

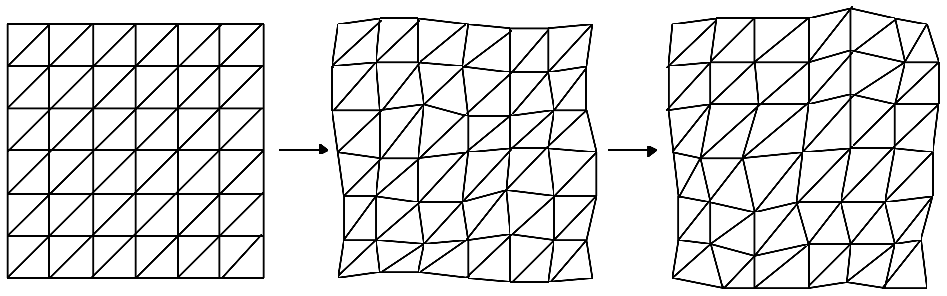


Figure 6-8: Mesh object with uniform triangular geometry.

6.1.4.1 Mesh object plane

There are two types of mesh object planes that use different coding methods.

An *intra-coded* mesh object plane codes the geometry of a single 2D mesh. An intra-coded mesh is either of uniform or Delaunay type. In the case of a mesh of uniform type, the mesh geometry is coded by a small set of parameters. In the case of a mesh of Delaunay type, the mesh geometry is coded by the locations of the node points and boundary edge segments. The triangular mesh structure is specified implicitly by the coded information.

A *predictive-coded* mesh object plane codes a 2D mesh using temporal prediction from a past reference mesh object plane. The triangular structure of a predictive-coded mesh is identical to the structure of the reference mesh used for prediction; however, the locations of node points may change. The displacements of node points represent the motion of the mesh and are coded by specifying the motion vectors of node points from the reference mesh towards the predictive-coded mesh.

Note that each coded mesh is restricted to be *simple*, i.e. it consists of a single connected component without any holes, topologically equivalent to a disk.

6.1.5 Face object

Conceptually the face object consists of a collection of nodes in a scene graph which are animated by the facial object bitstream. The shape, texture and expressions of the face are generally controlled by the bitstream containing instances of Facial Definition Parameter (FDP) sets and/or Facial Animation Parameter (FAP) sets. Upon construction, the Face object contains a generic face with a neutral expression. This face can already be rendered. It is also immediately capable of receiving the FAPs from the bitstream, which will produce animation of the face: expressions, speech etc. If FDPs are received, they are used to transform the generic face into a particular face determined by its shape and (optionally) texture. Optionally, a complete face model can be downloaded via the FDP set as a scene graph for insertion in the face node.

The FDP and FAP sets are designed to allow the definition of a facial shape and texture, as well as animation of faces reproducing expressions, emotions and speech pronunciation. The FAPs, if correctly interpreted, will produce reasonably similar high level results in terms of expression and speech pronunciation on different facial models, without the need to initialize or calibrate the model. The FDPs allow the definition of a precise facial shape and texture in the setup phase. If the FDPs are used in the setup phase, it is also possible to produce more precisely the movements of particular facial features. Using a phoneme to FAP conversion it is possible to control facial models accepting FAPs via TTS systems. The translation from phonemes to FAPs is not standardized. It is assumed that every decoder has a default face model with default parameters. Therefore, the setup stage is not necessary to create face animation. The setup stage is used to customize the face at the decoder.

6.1.5.1 Structure of the face object bitstream

A face object is formed by a temporal sequence of face object planes. This is depicted as follows in Figure 6-9.

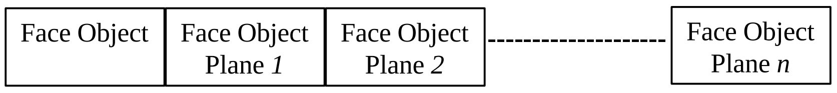
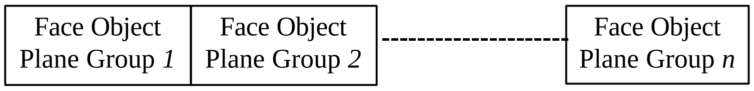


Figure 6-9 Structure of the face object bitstream

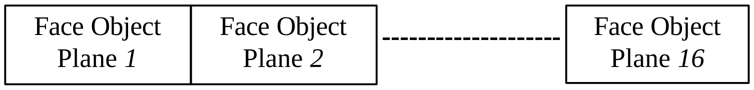
A face object represents a node in an MPEG4 scene graph. An MPEG-4 scene is understood as a composition of Audio-Visual objects according to some spatial and temporal relationships. The scene graph is the hierarchical representation of the MPEG-4 scene structure (see ISO/IEC 14496-1).

Alternatively, a face object can be formed by a temporal sequence of face object plane groups (called segments for simplicity), where each face object plane group itself is composed of a temporal sequence of 16 face object planes, as depicted in the following:

face object:


Face Object Plane Group *1*
Face Object Plane Group *2*
Face Object Plane Group *n*

face object plane group:



When the alternative face object bitstream structure is employed, the bitstream is decoded by DCT-based face object decoding as described in Section 7.11.2. Otherwise, the bitstream is decoded by the frame-based face object decoding.

6.1.5.2 Facial animation parameter set

The FAPs are based on the study of minimal facial actions and are closely related to muscle actions. They represent a complete set of basic facial actions, and therefore allow the representation of most natural facial expressions. Exaggerated values permit the definition of actions that are normally not possible for humans, but could be desirable for cartoon-like characters.

The FAP set contains two high level parameters visemes and expressions. A viseme is a visual correlate to a phoneme. The viseme parameter allows viseme rendering (without having to express them in terms of other parameters) and enhances the result of other parameters, insuring the correct rendering of visemes. Only static visemes which are clearly distinguished are included in the standard set. Additional visemes may be added in future extensions of the standard. Similarly, the expression parameter allows definition of high level facial expressions. The facial expression parameter values are defined by textual descriptions. To facilitate facial animation, FAPs that can be used together to represent natural expression are grouped together in FAP groups, and can be indirectly addressed by using an expression parameter. The expression parameter allows for a very efficient means of animating faces. In Annex C, a list of the FAPs is given, together with the FAP grouping, and the definitions of the facial expressions.

6.1.5.3 Facial animation parameter units

All the parameters involving translational movement are expressed in terms of the *Facial Animation Parameter Units (FAPU)*. These units are defined in order to allow interpretation of the FAPs on any facial model in a consistent way, producing reasonable results in terms of expression and speech pronunciation. They correspond to fractions of distances between some key facial features and are defined in terms of distances between feature points. The fractional units used are chosen to allow enough precision. Annex C contains the list of the FAPs and the list of the FDP feature points. For each FAP the list contains the name, a short description, definition of the measurement units, whether the parameter is unidirectional (can have only positive values) or bi-directional, definition of the direction of movement for positive values, group number (for coding of selected groups), FDP subgroup number (Annex C) and quantization step size. FAPs act on FDP feature points in the indicated subgroups. The measurement units are shown in Table 6-1, where the notation 3.1.y represents the y coordinate of the feature point 3.1; also refer to Figure 6-10.

Table 6-1 Facial Animation Parameter Units

Description		**FAPU Value**
IRISD0 = 3.1.y – 3.3.y = 3.2.y – 3.4.y	Iris diameter (by definition it is equal to the distance between upper ad lower eyelid) in neutral face	IRISD = IRISD0 / 1024
ES0 = 3.5.x – 3.6.x	Eye separation	ES = ES0 / 1024
ENS0 = 3.5.y – 9.15.y	Eye - nose separation	ENS = ENS0 / 1024
MNS0 = 9.15.y – 2.2.y	Mouth - nose separation	MNS = MNS0 / 1024
MW0 = 8.3.x – 8.4.x	Mouth width	MW = MW0 / 1024
AU	Angle Unit	10^{-5} rad

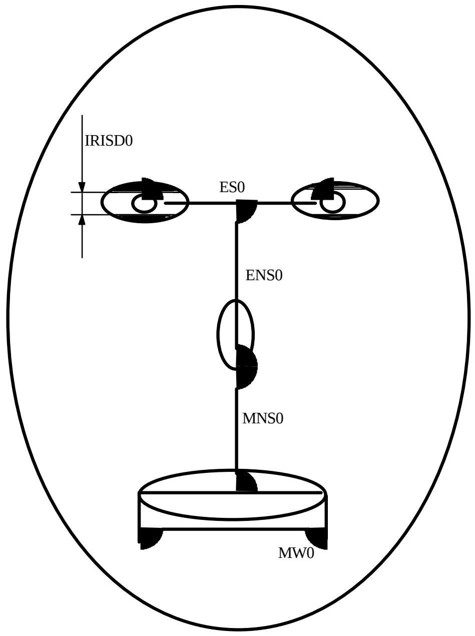


Figure 6-10 The Facial Animation Parameter Units

6.1.5.4 Description of a neutral face

At the beginning of a sequence, the face is supposed to be in a neutral position. Zero values of the FAPs correspond to a neutral face. All FAPs are expressed as displacements from the positions defined in the neutral face. The neutral face is defined as follows:

- the coordinate system is right-handed; head axes are parallel to the world axes
- gaze is in direction of Z axis
- all face muscles are relaxed
- eyelids are tangent to the iris
- the pupil is one third of IRISD0
- lips are in contact; the line of the lips is horizontal and at the same height of lip corners
- the mouth is closed and the upper teeth touch the lower ones
- the tongue is flat, horizontal with the tip of tongue touching the boundary between upper and lower teeth (feature point 6.1 touching 9.11 in Annex C)

6.1.5.5 Facial definition parameter set

The FDPs are used to customize the proprietary face model of the decoder to a particular face or to download a face model along with the information about how to animate it. The definition and description of FDP fields is given in Annex C. The FDPs are normally transmitted once per session, followed by a stream of compressed FAPs. However, if the decoder does not receive the FDPs, the use of FAPUs ensures that it can still interpret the FAP stream. This insures minimal operation in broadcast or teleconferencing applications. The FDP set is specified in BIFS syntax (see ISO/IEC 14496-1). The FDP node defines the face model to be used at the receiver. Two options are supported:

- calibration information is downloaded so that the proprietary face of the receiver can be configured using facial feature points and optionally a 3D mesh or texture.
- a face model is downloaded with the animation definition of the Facial Animation Parameters. This face model replace the proprietary face model in the receiver.

6.2 Visual bitstream syntax

6.2.1 Start codes

Start codes are specific bit patterns that do not otherwise occur in the video stream.

Each start code consists of a start code prefix followed by a start code value. The start code prefix is a string of twenty three bits with the value zero followed by a single bit with the value one. The start code prefix is thus the bit string ‘0000 0000 0000 0000 0000 0001’.

The start code value is an eight bit integer which identifies the type of start code. Many types of start code have just one start code value. However video_object_start_code and video_object_layer_start_code are represented by many start code values.

All start codes shall be byte aligned. This shall be achieved by first inserting a bit with the value zero and then, if necessary, inserting bits with the value one before the start code prefix such that the first bit of the start code prefix is the first (most significant) bit of a byte. For stuffing of 1 to 8 bits, the codewords are as follows in Table 6-2. Nevertheless, these stuffing bits shall not be present if and only if the previous code is a start code.

Table 6-2-- Stuffing codewords

Bits to be stuffed	Stuffing Codeword
1	0
2	01
3	011
4	0111
5	01111
6	011111
7	0111111
8	01111111

Table 6-3 defines the start code values for all start codes used in the visual bitstream.

Table 6-3 — Start code values

name	**start code value (hexadecimal)**
video_object_start_code	00 through 1F
video_object_layer_start_code	20 through 2F
reserved	30 through AF
visual_object_sequence__start_code	B0
visual_object_sequence_end_code	B1
user_data_start_code	B2
group_of_VOP_start_code	B3
video_session_error_code	B4
visual_object_start_code	B5
VOP_start_code	B6
reserved	B7-B9
face_object_start_code	BA
face_object_plane_start_code	BB
mesh_object_start_code	BC
mesh_object_plane_start_code	BD
still_texture_object_start_code	BE
texture_spatial_layer_start_code	BF
texture_snr_layer_start_code	C0
reserved	C0-C5
System start codes (see note)	C6 through FF
NOTE - System start codes are defined in Part 1 of this specification	

The use of the start codes is defined in the following syntax description with the exception of the video_session_error_code. The video_session_error_code has been allocated for use by a media interface to indicate where uncorrectable errors have been detected.

6.2.2 Visual Object Sequence and Visual Object

VisualObjectSequence() {	**No. of bits**	**Mnemonic**
visual_object_sequence_start_code	32	bslbf
while (nextbits()== user_data_start_code){		
user_data()		
}		
do {		
VisualObject()		
}while(nextbits()==visual_object_sequence_start_code)		
visual_object_sequence_end_code	32	bslbf
}		

VisualObject() {	**No. of bits**	**Mnemonic**
visual_object_start_code	32	bslbf
profile_and_level_indication	8	uimsbf
is_visual_object_identifier	1	uimsbf
if (is_visual_object_identifier) {		
visual_object_verid	4	uimsbf
visual_object_priority	3	uimsbf
}		
visual_object_type	4	uimsbf
next_start_code()		
while (nextbits()== user_data_start_code){		
user_data()		
}		
if (visual_object_type == “video ID” || visual_object_type == “still texture ID“) {		
video_signal_type()		
}		
if (visual_object_type == “video ID”) {		
VideoObject()		
}		
else if (visual_object_type == “still texture ID”) {		
StillTextureObject()		
}		
else if (visual_object_type == “mesh ID”) {		
MeshObject()		
}		
else if (visual_object_type == “face ID”) {		
FaceObject()		
}		
next_start_code()		
}		

video_signal_type()	1	bslbf
video_signal_type	1	bslbf
if(video_signal_type) {		
video_format	3	uimsbf
video_range	1	bslbf
colour_description	1	bslbf
if (colour_description) {		
colour_primaries	8	uimsbf
transfer_characteristics	8	uimsbf
matrix_coefficients	8	uimsbf
}		

}		

6.2.2.1 User data

user_data() {	**No. of bits**	**Mnemonic**
user_data_start_code	32	bslbf
while(nextbits() != ‘0000 0000 0000 0000 0000 0001’) {		
user_data	8	uimsbf
}		
next_start_code()		
}		

6.2.3 Video Object

VideoObject() {	**No. of bits**	**Mnemonic**
video_object_start_code /* 5 least significant bits specify video_object_id value */	32	bslbf
while (nextbits()== user_data_start_code){		
user_data()		
}		
do{		
VideoObjectLayer()		
} while (next_bits() == video_object_layer_start_code)		
}		

6.2.4 Video Object Layer

VideoObjectLayer() {	**No. of bits**	**Mnemonic**
if(nextbits() == video_object_layer_start_code) {		
short_video_header = 0		
video_object_layer_start_code	32	bslbf
is_object_layer_identifier	1	uimsbf
if (is_object_layer_identifier) {		
video_object_layer_verid	4	uimsbf
video_object_layer_priority	3	uimsbf
}		
vol_control_parameters	1	bslbf
if (vol_control_parameters)		
aspect_ratio_info	4	uimsbf
VOP_rate_code	4	uimsbf
bit_rate	30	uimsbf
vbv_buffer_size	18	uimsbf
chroma_format	2	uimsbf
low_delay	1	uimsbf
}		
video_object_layer_shape	2	uimsbf
VOP_time_increment_resolution	15	uimsbf
fixed_VOP_rate	1	bslbf
if (video_object_layer_shape != “binary only”) {		
if (video_object_layer_shape == “rectangular”) {		
marker_bit	1	bslbf
video_object_layer_width	13	uimsbf
marker_bit	1	bslbf
video_object_layer_height	13	uimsbf
}		
obmc_disable	1	bslbf
sprite_enable	1	bslbf
if (sprite_enable) {		
sprite_width	13	uimsbf
marker_bit	1	bslbf
sprite_height	13	uimsbf
marker_bit	1	bslbf
sprite_left_coordinate	13	simsbf
marker_bit	1	bslbf
sprite_top_coordinate	13	simsbf
marker_bit	1	bslbf
no_of_sprite_warping_points	6	uimsbf
sprite_warping_accuracy	2	uimsbf
sprite_brightness_change	1	bslbf
low_latency_sprite_enable	1	bslbf

}		
not_8_bit	1	bslbf
if (not_8_ bit) {		
quant_precision	4	uimsbf
bits_per_pixel	4	uimsbf
if (video_object_layer_shape==”grayscale”) {		
no_gray_quant_update	1	bslbf
}		
}		
quant_type	1	bslbf
if (quant_type) {		
load_intra_quant_mat	1	bslbf
if (load_intra_quant_mat)		
intra_quant_mat	8*[2-64]	uimsbf
load_nonintra_quant_mat	1	bslbf
if (load_nonintra_quant_mat)		
nonintra_quant_mat	8*[2-64]	uimsbf
if(video_object_layer_shape==”grayscale”) {		
load_intra_quant_mat_grayscale	1	bslbf
if(load_intra_quant_mat_grayscale)		
intra_quant_mat_grayscale	8*[2-64]	uimsbf
load_nonintra_quant_mat_grayscale	1	bslbf
if(load_nonintra_quant_mat_grayscale)		
nonintra_quant_mat_grayscale	8*[2-64]	uimsbf
}		
}		
complexity_estimation_disable	1	bslbf
if (!complexity_estimation_disable)		
define_VOP_complexity_estimation_header()		
error_resilient_disable	1	bslbf
if (!error_resilient_disable) {		
data_partitioned	1	bslbf
if(data_partitioned)		
reversible_vlc	1	bslbf
}		
scalability	1	bslbf
if (scalability) {		
ref_layer_id	4	uimsbf
ref_layer_sampling_direc	1	bslbf
hor_sampling_factor_n	5	uimsbf
hor_sampling_factor_m	5	uimsbf
vert_sampling_factor_n	5	uimsbf
vert_sampling_factor_m	5	uimsbf
enhancement_type	1	bslbf

}		
}		
random_accessible_vol	1	bslbf
else		
error_resilient_disable	1	bslbf
next_start_code()		
while (nextbits()== user_data_start_code){		
user_data()		
}		
if (sprite_enable && !low_latency_sprite_enable)		
VideoObjectPlane()		
do {		
if (next_bits() == group_of_VOP_start_code)		
Group_of_VideoObjectPlane()		
VideoObjectPlane()		
} while ((next_bits() == group_of_VOP_start_code) || (next_bits() == VOP_start_code))		
} else {		
short_video_header = 1		
do {		
short_video_start_marker	22	bslbf
video_plane_with_short_header()		
if(nextbits() == short_video_end_marker) {		
short_video _end_marker	22	uimsbf
while(!byte_aligned())		
zero_bit	1	bslbf
}		
} while(next_bits() == short_video_start_marker)		
}		
}		

define_VOP_complexity_estimation_header() {	**No. of bits**	**Mnemonic**
estimation_method	2	uimsbf
if (estimation_method ==’00’){		
shape_complexity_estimation_disable	1	
if (shape_complexity_estimation_disable) {		bslbf
opaque	1	bslbf
transparent	1	bslbf
intra_cae	1	bslbf
inter_cae	1	bslbf
no_update	1	bslbf
upsampling	1	bslbf
}		
texture_complexity_estimation_set_1_disable	1	bslbf
if (!texture_complexity_estimation_set_1_disable) {		
intra_blocks	1	bslbf
inter_blocks	1	bslbf
inter4v_blocks	1	bslbf
not_coded_blocks	1	bslbf
}		
texture_complexity_estimation_set_2_disable	1	bslbf
if (!texture_complexity_ estimation_set_2_disable) {		
dct_coefs	1	bslbf
dct_lines	1	bslbf
vlc_symbols	1	bslbf
vlc_bits	1	bslbf
}		
motion_compensation_complexity_disable	1	bslbf
If (!motion_compensation_complexity_disable) {		
apm	1	bslbf
npm	1	bslbf
interpolate_mc_q	1	bslbf
forw_back_mc_q	1	bslbf
halfpel2	1	bslbf
halfpel4	1	bslbf
}		
}		
}		

6.2.5 Group of Video Object Plane

Group_of_VideoObjectPlane() {	**No. of bits**	**Mnemonic**
group_VOP_start_code	32	bslbf
time_code	18	
closed_gov	1	bslbf
broken_link	1	bslbf
next_start_code()		
while (nextbits()== user_data_start_code){		
user_data()		
}		
}		

6.2.6 Video Object Plane and Video Plane with Short Header

VideoObjectPlane() {	**No. of bits**	**Mnemonic**
VOP_start_code	32	bslbf
VOP_coding_type	2	uimsbf
do {		
modulo_time_base	1	bslbf
} while (modulo_time_base != ‘0’)		
marker_bit	1	bslbf
VOP_time_increment	1-15	uimsbf
marker_bit	1	bslbf
VOP_coded	1	bslbf
if (VOP_coded == ’0’) {		
next_start_code()		
return()		
}		
if ((video_object_layer_shape != “binary only”) && (VOP_coding_type == “P”))		
VOP_rounding_type	1	bslbf
if (video_object_layer_shape != “rectangular”) {		
if(!(sprite_enable && VOP_coding_type == “I”)) {		
VOP_width	13	uimsbf
marker_bit	1	bslbf
VOP_height	13	uimsbf
marker_bit	1	bslbf
VOP_horizontal_mc_spatial_ref	13	simsbf
marker_bit	1	bslbf
VOP_vertical_mc_spatial_ref	13	simsbf
}		
if ((video_object_layer_shape != “ binary only”) && scalability && enhancement_type)		
background_composition	1	bslbf
change_conv_ratio_disable	1	bslbf
VOP_constant_alpha	1	bslbf
if (VOP_constant_alpha)		
VOP_constant_alpha_value	8	bslbf
}		
if (!complexity_estimation_disable)		
read_VOP_complexity_estimation_header()		
if (video_object_layer_shape != “binary only”) {		
intra_dc_vlc_thr	3	uimsbf
interlaced	1	bslbf
if (interlaced) {		
top_field_first	1	bslbf
alternate_scan	1	bslbf

}		
}		
if (sprite_enable && VOP_coding_type == “S”) {		
if (no_sprite_points > 0)		
sprite_trajectory()		
if (sprite_brightness_change)		
brightness_change_factor()		
if (sprite_transmit_mode != “stop” && low_latency_sprite_enable) {		
do {		
sprite_transmit_mode	2	uimsbf
if ((sprite_transmit_mode == “piece”) || (sprite_transmit_mode == “update”))		
decode_sprite_piece()		
} while (sprite_transmit_mode != “stop” && sprite_transmit_mode != “pause”)		
}		
next_start_code()		
return()		
}		
if (video_object_layer_shape != “binary only”) {		
VOP_quant	3-9	uimsbf
if(video_object_layer_shape==”grayscale”)		
VOP_alpha_quant	6	uimsbf
if (VOP_coding_type != “I”)		
VOP_fcode_forward	3	uimsbf
if (VOP_coding_type == “B”)		
VOP_fcode_backward	3	uimsbf
if (!scalability) {		
if (!error_resilient_disable) {		
if (video_object_layer_shape != “rectangular” && VOP_coding_type != “I”)		
VOP_shape_coding_type	1	bslbf
motion_shape_texture()		
while (nextbits_bytealigned() == resync_marker) {		
video_packet_header()		
motion_shape_texture()		
}		
}		
else{		
do {		
motion_shape_texture()		
} while (nextbits_bytealigned() != ‘0000 0000 0000 0000 0000 000’)		

}		
}		
else {		
if (enhancement_type) {		
load_backward_shape	1	bslbf
if (load_backward_shape) {		
backward_shape_width	13	uimsbf
backward_shape_ height	13	uimsbf
backward_shape_horizontal_mc_spatial_ref	13	simsbf
marker_bit	1	bslbf
backward_shape_vertical_mc_spatial_ref	13	simsbf
backward_shape()		
load_forward_shape	1	bslbf
if (load_forward_shape) {		
forward_shape_width	13	uimsbf
forward_shape_height	13	uimsbf
forward_shape_horizontal_mc_spatial_ref	13	simsbf
marker_bit	1	bslbf
forward_shape_vertical_mc_spatial_ref	13	simsbf
forward_shape()		
}		
}		
}		
ref_select_code	2	uimsbf
motion_shape_texture()		
}		
}		
else {		
if (!error_resilient_disable) {		
motion_shape_texture()		
while (nextbits_bytealigned() == resync_marker) {		
video_packet_header()		
motion_shape_texture()		
}		
} else		
motion_shape_texture()		
}		
next_start_code()		
}		

read_VOP_complexity_estimation_header() {	No. of bits	Mnemonic
if (estimation_method==’00’){		
if (VOP_prediction_type==‘00’){		
if (opaque) **dcecs_opaque**	8	uimsbf
if (transparent) **dcecs_transparent**	8	uimsbf
if (intra_cae) **dcecs_intra_cae**	8	uimsbf
if (inter_cae) **dcecs_inter_cae**	8	uimsbf
if (no_update) **dcecs_no_update**	8	uimsbf
if (upsampling) **dcecs_upsampling**	8	uimsbf
if (intra_blocks) **dcecs_intra_blocks**	8	uimsbf
if (not_coded_blocks) **dcecs_not_coded_blocks**	8	uimsbf
if (dct_coefs) **dcecs_dct_coefs**	8	uimsbf
if (dct_lines) **dcecs_dct_lines**	8	uimsbf
if (vlc_symbols) **dcecs_vlc_symbols**	8	uimsbf
if (vlc_bits) **dcecs_vlc_bits**	4	uimsbf
}		
if (VOP_prediction_type==‘01’){		
if (opaque) **dcecs_opaque**	8	uimsbf
if (transparent) **dcecs_transparent**	8	uimsbf
if (intra_cae) **dcecs_intra_cae**	8	uimsbf
if (inter_cae) **dcecs_inter_cae**	8	uimsbf
if (no_update) **dcecs_no_update**	8	uimsbf
if (upsampling) **dcecs_upsampling**	8	uimsbf
if (intra) **dcecs_intra_blocks**	8	uimsbf
if (not_coded) **dcecs_not_coded_blocks**	8	uimsbf
if (dct_coefs) **dcecs_dct_coefs**	8	uimsbf
if (dct_lines) **dcecs_dct_lines**	8	uimsbf
if (vlc_symbols) **dcecs_vlc_symbols**	8	uimsbf
if (vlc_bits) **dcecs_vlc_bits**	4	uimsbf
if (inter_blocks) **dcecs_inter_blocks**	8	uimsbf
if (inter4v_blocks) **dcecs_inter4v_blocks**	8	uimsbf
if (apm) **dcecs_apm**	8	uimsbf
if (npm) **dcecs_npm**	8	uimsbf
if (forw_back_mc_q) **dcecs_forw_back_mc_q**	8	uimsbf
if (halfpel2) **dcecs_halfpel2**	8	uimsbf
if (halfpel4) **dcecs_halfpel4**	8	uimsbf
}		
if (VOP_prediction_type==‘10’){		
if (opaque) **dcecs_opaque**	8	uimsbf
if (transparent) **dcecs_transparent**	8	uimsbf
if (intra_cae) **dcecs_intra_cae**	8	uimsbf
if (inter_cae) **dcecs_inter_cae**	8	uimsbf
if (no_update) **dcecs_no_update**	8	uimsbf

if (upsampling) **dcecs_upsampling**	8	uimsbf
if (intra_blocks) **dcecs_intra_blocks**	8	uimsbf
if (not_coded_blocks) **dcecs_not_coded_blocks**	8	uimsbf
if (dct_coefs) **dcecs_dct_coefs**	8	uimsbf
if (dct_lines) **dcecs_dct_lines**	8	uimsbf
if (vlc_symbols) **dcecs_vlc_symbols**	8	uimsbf
if (vlc_bits) **dcecs_vlc_bits**	4	uimsbf
if (inter_blocks) **dcecs_inter_blocks**	8	uimsbf
if (inter4v_blocks) **dcecs_inter4v_blocks**	8	uimsbf
if (apm) **dcecs_apm**	8	uimsbf
if (npm) **dcecs_npm**	8	uimsbf
if (forw_back_mc_q) **dcecs_forw_back_mc_q**	8	uimsbf
if (halfpel2) **dcecs_halfpel2**	8	uimsbf
if (halfpel4) **dcecs_halfpel4**	8	uimsbf
if (interpolate_mc_q) **dcecs_interpolate_mc_q**	8	uimsbf
}		
if (VOP_prediction_type==‘11’){		
if (intra_blocks) **dcecs_intra_blocks**	8	uimsbf
if (not_coded_blocks) **dcecs_not_coded_blocks**	8	uimsbf
if (dct_coefs) **dcecs_dct_coefs**	8	uimsbf
if (dct_lines) **dcecs_dct_lines**	8	uimsbf
if (vlc_symbols) **dcecs_vlc_symbols**	8	uimsbf
if (vlc_bits) **dcecs_vlc_bits**	4	uimsbf
if (inter_blocks) **dcecs_inter_blocks**	8	uimsbf
if (inter4v_blocks) **dcecs_inter4v_blocks**	8	uimsbf
if (apm) **dcecs_apm**	8	uimsbf
if (npm) **dcecs_npm**	8	uimsbf
if (forw_back_q) **dcecs_forw_back_q**	8	uimsbf
if (halfpel2) **dcecs_halfpel2**	8	uimsbf
if (halfpel4) **dcecs_halfpel4**	8	uimsbf
if (interpolate_mc_q) **dcecs_interpolate_mc_q**	8	uimsbf
}		
}		
}		

video_plane_with_short_header() {	**No. of bits**	**Mnemonic**
temporal_reference	8	uimsbf
marker_bit	1	bslbf
zero_bit	1	bslbf
split_screen_indicator	1	bslbf
document_camera_indicator	1	bslbf
full_picture_freeze_release	1	bslbf
source_format	3	bslbf
picture_coding_type	1	bslbf
four_reserved_zero_bits	4	bslbf
vop_quant	5	uimsbf
zero_bit	1	bslbf
do{		
pei	1	bslbf
if (pei == “1”)		
psupp	8	bslbf
} while (pei == “1”)		
gob_number = 0		
for(i=0; i<num_gobs_in_vop; i++)		
gob_layer()		
}		

gob_layer() {	**No. of bits**	**Mnemonic**
gob_header_empty = 1		
if(gob_number != 0) {		
if (next_bits == gob_resync_marker) {		
gob_header_empty = 0		
gob_resync_marker	17	bslbf
gob_number	5	uimsbf
gob_frame_id	2	bslbf
quant_scale	5	uimsbf
} else		
gob_number++		
}		
for(i=0; i<num_macroblocks_in_gob; i++)		
macroblock()		
if(next_bits != gob_resync_marker && nextbits_byte_aligned == gob_resync_marker)		
while(!byte_aligned())		
zero_bit	1	bslbf
}		

video_packet_header() {	**No. of bits**	**Mnemonic**
next_resync_marker()		
resync_marker	17-23	uimsbf
macroblock_number	1-14	vlclbf
if (video_object_layer_shape != “binary only”)		
quant_scale	5	uimsbf
header_extension_code	1	uimsbf
if (header_extension_code) {		
do {		
modulo_time_base	1	bslbf
} while (modulo_time_base != ‘0’)		
marker_bit	1	bslbf
VOP_time_increment	1-15	bslbf
marker_bit	1	uimsbf
VOP_coding_type	2	uimsbf
intra_dc_vlc_thr	3	uimsbf
if (video_object_layer_shape != “binary only”) {		
if (VOP_coding_type != “I”)		
VOP_fcode_forward	3	uimsbf
if (VOP_coding_type == “B”)		
VOP_fcode_backward	3	uimsbf
}		
}		
}		

6.2.6.1 Motion Shape Texture

motion_shape_texture() {	**No. of bits**	**Mnemonic**
if (data_partitioning && video_object_layer_shape != “binary only”)		
data_partitioning_motion _shape_texture()		
else		
combined_motion_shape_texture()		
}		

combined_motion_shape_texture() {	**No. of bits**	**Mnemonic**
do{		
macroblock()		
} while (nextbits_bytealigned() != resync_marker && nextbits_bytealigned() != ‘000 0000 0000 0000 0000 0000’)		
}		

data_partitioning_motion_shape_texture() {	**No. of bits**	**Mnemonic**
if (VOP_coding_type == "I") {		
data_partitioning_I_VOP()		
} else if (VOP_coding_type == "P") {		
data_partitioning_P_VOP()		
} else if (VOP_coding_type == "B") {		
combined_motion_shape_texture()		
}		

Note: Data partitioning is not supported in B-VOPs.

data_partitioned_I_VOP() {	**No. of bits**	**Mnemonic**
do{		
if (video_object_layer_shape != “rectangular”){		
bab_type	1-3	
if (bab_type >= 4) {		
if (!change_conv_rate_disable) **conv_ratio**	1-2	
scan_type	1	
binary_arithmetic_code()		
}		
}		
if (!transparent_mb()) {		
mcbpc	1-9	vlclbf
if (mb_type == 4)		
dquant	2	bslbf
if (use_intra_dc_vlc) {		
for (j = 0; j < 4; j++) {		
if (!transparent_block(j)) {		
dct_dc_size_luminance	2-11	vlclbf
if (dct_dc_size_luminance > 0)		
dct_dc_differential	1-12	vlclbf
if (dct_dc_size_luminance > 8)		
marker_bit	1	bslbf
}		
}		
for (j = 0; j < 2; j++) {		
dct_dc_size_chrominance	2-12	vlclbf
if (dct_dc_size_chrominance > 0)		
dct_dc_differential	1-12	vlclbf
if (dct_dc_size_chrominance > 8)		
marker_bit	1	bslbf
}		
}		
}		
} while (nextbits() != dc_marker)		
dc_marker /* 110 1011 0000 0000 0001 */	19	bslbf
for (i = 0; i < mb_in_video_packet; i++) {		
if (!transparent_mb()) {		
ac_pred_flag	1	bslbf
cbpy	1-6	vlclbf
}		
}		
for (i = 0; i < mb_in_video_packet; i++) {		
if (!transparent_mb()) {		

for (j = 0; j < block_count; j++)		
block(j)		
}		
}		
}		

data_partitioned_P_VOP() {	**No. of bits**	**Mnemonic**
do{		
if (video_object_layer_shape != “rectangular”){		
bab_type	1-7	vlclbf
if ((bab_type == 1) || (bab_type == 6)) {		
mvds_x	1-18	vlclbf
mvds_y	1-18	vlclbf
}		
if (bab_type >= 4) {		
if (!change_conv_rate_disable) **conv_ratio**	1-2	vlclbf
scan_type	1	bslbf
binary_arithmetic_code()		
}		
}		
if (!transparent_mb()) {		
not_coded	1	bslbf
if (!not_coded) {		
mcbpc	1-9	vlclbf
if (derived_mb_type < 3)		
motion_coding(“forward”, derived_mb_type)		
}		
}		
} while (nextbits() != motion_marker)		
motion_marker **/*** 1 1111 0000 0000 0001 */	17	bslbf
for (i = 0; i < mb_in_video_packet; i++) {		
if (!transparent_mb()) {		
if (!not_coded){		
if (derived_mb_type >= 3)		
ac_pred_flag	1	bslbf
cbpy	1-6	vlclbf
if (derived_mb_type == 1 || derived_mb_type == 4)		
dquant	2	bslbf
if (derived_mb_type >= 3 && use_intra_dc_vlc) {		
for (j = 0; j < 4; j++) {		
if (!transparent_block(j)) {		
dct_dc_size_luminance	2-11	vlclbf
if (dct_dc_size_luminance > 0)		
dct_dc_differential	1-12	vlclbf
if (dct_dc_size_chrominance > 8)		
marker_bit	1	bslbf
}		
for (j = 0; j < 2; j++) {		
dc_size_chrominance	2-11	vlclbf

if (dct_dc_size_chrominance > 0)		
dct_dc_differential	1-12	vlclbf
if (dct_dc_size_chrominance > 8)		
marker_bit	1	bslbf
}		
} derived		
} !not_coded		
}!trans		
}for		
for (i = 0; i < mb_in_video_packet; i++) {		
if (!transparent_mb()) {		
if (! not_coded) {		
for (j = 0; j < block_count; j++)		
block(j)		
}		
}		
}		
}		

motion_coding(mode, type_of_mb) {	**No. of bits**	**Mnemonic**
motion_vector(mode)		
if (type_of_mb == 2) {		
for (i = 0; i < 3; i++)		
motion_vector(mode)		
}		
}		

6.2.6.2 Sprite coding

decode_sprite_piece() {	**No. of bits**	**Mnemonic**
piece_quant	5	bslbf
piece_width	9	bslbf
piece_height	9	bslbf
marker_bit	1	bslbf
piece_xoffset	9	bslbf
piece_yoffset	9	bslbf
sprite_shape_texture()		
}		

sprite_shape_texture() {	**No. of bits**	**Mnemonic**
if (sprite_transmit_mode == “piece”) {		
for (i=0; i < piece_height; i++) {		
for (j=0; j < piece_width; j++) {		
if (!send_mb()) {		
macroblock()		
}		
}		
}		
}		
if (sprite_transmit_mode == “update”) {		
for (i=0; i < piece_height; i++) {		
for (j=0; j < piece_width; j++) {		
macroblock()		
}		
}		
}		
}		

sprite_trajectory() {	**No. of bits**	**Mnemonic**
for (i=0; i < no_of_sprite_warping_points; i++) {		
warping_mv_code(du[i])		
warping_mv_code(dv[i])		
}		
}		

warping_mv_code(d) {	**No. of bits**	**Mnemonic**
dmv_length	2-9	uimsbf
dmv_code	0-11	uimsbf
}		

brightness_change_factor() {	**No. of bits**	**Mnemonic**
brightness_change_factor_size	1-4	uimsbf
brightness_change_factor_code	5-10	uimsbf
}		

6.2.7 Macroblock

macroblock() {	**No. of bits**	**Mnemonic**
if (VOP_coding_type != “B”) {		

if (video_object_layer_shape != “rectangular”
   && !(sprite_enable && low_latency_sprite_enable
   && sprite_transmit_mode == “update”))		

mb_binary_shape_coding()		
if (video_object_layer_shape != “binary only”) {		
if (!transparent_mb()) {		

if (VOP_coding_type != “I” && !(sprite_enable
     && sprite_transmit_mode == “piece”))		

not_coded	1	bslbf
if (!not_coded || VOP_coding_type == “I”) {		
mcbpc	1-9	vlclbf

if ((!short_video_header &&
      derived_mb_type == 3 ||
      derived_mb_type == 4))		

ac_pred_flag	1	bslbf
if (derived_mb_type != “stuffing”)		
cbpy	1-6	vlclbf
else		
return()		

if (derived_mb_type == 1 ||
      derived_mb_type == 4)		

dquant	2	uimsbf
if (interlaced)		
interlaced_information()		

if (!(ref_select_code==‘11’ && scalability)
      && VOP_coding_type != “S”) {		

if (derived_mb_type == 0 ||
       derived_mb_type == 1) {		

motion_vector(“forward”))		
if (field_prediction)		
motion_vector(“forward”)		
}		
if (derived_mb_type == 2) {		
for (j=0; j < 4; j++)		
if (!transparent_block(j))		
motion_vector(“forward”))		
}		
}		
for (i = 0; i < block_count; i++)		
block(i)		
}		

}		
}		
}		
else if (co_located_not_coded != 1 || (ref_select_code == ’11’ || enhancement_type == 1) && scalability)) {		
if (video_object_layer_shape != “rectangular”)		
mb_binary_shape_coding()		
if (video_object_layer_shape != “binary only”) {		
if (!transparent_mb()) {		
modb	1-2	vlclbf
if (modb) {		
if (modb > 0)		
mb_type	1-4	vlclbf
if (modb == 2)		
cbpb	6	uimsbf
if (ref_select_code != ‘00’ || !scalability) {		
if (mb_type != “1” && cbpb!=0)		
dquant	2	uimsbf
if (field_prediction)		
interlaced_information()		
if (mb_type == ‘01’ || mb_type == ‘0001’) {		
motion_vector(“forward”)		
if (interlaced)		
motion_vector(“forward”)		
}		
if (mb_type == ‘01’ || mb_type == ‘001’) {		
motion_vector(“backward”)		
if (field_prediction)		
motion_vector(“backward”)		
}		
if (mb_type == “1”)		
motion_vector(“direct”)		
}		
if (ref_select_code == ‘00’ && scalability && cbpb !=0) {		
dquant	2	uimsbf
if (mb_type == ‘01’ || mb_type == ‘1’)		
motion_vector(“forward”)		
}		
for (i = 0; i < block_count; i++)		
block(i)		
}		
}		

}		
}		
if(video_object_layer_shape==’grayscale’&& !transparent_mb()) {		
if(VOP_coding_type==”I” || (VOP_coding_type==”P” && !not_coded && mb_type==”I”)) {		
CODA_I	1	bslbf
if(CODA_I==”coded”) {		
ac_pred_flag_alpha	1	bslbf
CBPA	2—6	vlclbf
for(i=0;i<alpha_block_count;i++)		
alpha_block(i)		
}		
} else { /* P or B macroblock */		
if(VOP_coding_type==”P” || co_located_not_coded!=1) {		
CODA_PB	1—2	vlclbf
if(CODA_PB==”coded”) {		
CBPA	2—6	vlclbf
for(i=0;i<alpha_block_count;i++)		
alpha_block(i)		
}		
}		
}		
}		
}		

6.2.7.1 MB Binary Shape Coding

mb_binary_shape_coding() {	**No. of bits**	**Mnemonic**
bab_type	1-7	vlclbf
if ((VOP_coding_type == ‘P’) || (VOP_coding_type == ‘B’)) {		
if ((bab_type==1) || (bab_type == 6)) {		
mvds_x	1-18	vlclbf
mvds_y	1-18	vlclbf
}		
}		
if (bab_type >=4) {		
if (!change_conv_ratio_disable)		
conv_ratio	1-2	vlcbf
scan_type	1	bslbf
binary_arithmetic_code()		
}		
}		

backward_shape () {	**No. of bits**	**Mnemonic**
for(i=0; i<backward_shape_height/16; i++)		
for(j=0; j<backward_shape_width/16; j++) {		
bab_type	1-3	vlclbf
if (bab_type >=4) {		
if (!change_conv_ratio_disable)		
conv_ratio	1-2	vlcbf
scan_type	1	bslbf
binary_arithmetic_code()		
}		
}		
}		

forward_shape () {	**No. of bits**	**Mnemonic**
for(i=0; i<forward_shape_height/16; i++)		
for(j=0; j<forward_shape_width/16; j++) {		
bab_type	1-3	vlclbf
if (bab_type >=4) {		
if (!change_conv_ratio_disable)		
conv_ratio	1-2	vlcbf
scan_type	1	bslbf
binary_arithmetic_code()		
}		
}		
}		

6.2.7.2 Motion vector

motion_vector (mode) {	**No. of bits**	**Mnemonic**
if (mode == „direct“) {		
horizontal_mv_data	1-13	vlclbf
vertical_mv_data	1-13	vlclbf
}		
else if (mode == „forward“) {		
horizontal_mv_data	1-13	vlclbf
if ((VOP_fcode_forward != 1)&&(horizontal_mv_data != 0))		
horizontal_mv_residual	1-6	uimsbf
vertical_mv_data	1-13	vlclbf
if ((VOP_fcode_forward != 1)&&(vertical_mv_data != 0))		
vertical_mv_residual	1-6	uimsbf
}		
else if (mode == „backward“) {		
horizontal_mv_data	1-13	vlclbf
if ((VOP_fcode_backward != 1)&&(horizontal_mv_data != 0))		
horizontal_mv_residual	1-6	uimsbf
vertical_mv_data	1-13	vlclbf
if ((VOP_fcode_backward != 1)&&(vertical_mv_data != 0))		
vertical_mv_residual	1-6	uimsbf
}		
}		

6.2.7.3 Interlaced Information

interlaced_information ( ) {	**No. of bits**	**Mnemonic**
if ((derived_mbtype == 3) || (derived_mbtype == 4) || (cbp != 0))		
dct_type	1	bslbf
if (((VOP_coding_type == "P") && ((derived_mbtype == 0) || (derived_mbtype == 1))) || ((VOP_coding_type == "B") && (mb_type != "1"))) {		
field_prediction	1	bslbf
if (field_prediction) {		
if (VOP_coding_type == "P" || (VOP_coding_type == "B" && mb_type != "001")) {		
forward_top_field_reference	1	bslbf
forward_bottom_field_reference	1	bslbf
}		
if ((VOP_coding_type == "B") && (mb_type != "0001")) {		
backward_top_field_reference	1	bslbf
backward_bottom_field_reference	1	bslbf
}		
}		
}		
}		

6.2.8 Block

The detailed syntax for the terms "First DCT coefficient", "Subsequent DCT coefficient" and "End of Block" is fully described in the section 7.

block(i) {	**No. of bits**	**Mnemonic**
if (pattern_code[i]) {		
if((derived_mb_type==3 || derived_mb_type==4) &&use_intra_dc_vlc && ! data_partitioning && short_video_header==0) {		
if (i<4) {		
dct_dc_size_luminance	2-11	vlclbf
if(dct_dc_size_luminance != 0)		
dct_dc_differential	1-11	uimsbf
if (dct_dc_size_luminance > 8)		
marker_bit	1	bslbf
} else {		
dct_dc_size_chrominance	2-12	vlclbf
if(dct_dc_size_chrominance !=0)		
dct_dc_differential	1-11	uimsbf
if (dct_dc_size_luminance > 8)		
marker_bit	1	bslbf
}		
} else {		
First DCT coefficient	2-24	vlclbf
}		
while (nextbits() != lastcoef)		
Subsequent DCT coefficients	3-24	vlclbf
}		
}		

6.2.8.1 Alpha Block

The syntax for DCT coefficient decoding is the same as for Block in Section 6.2.8.

alpha_block(i) {		
if(pattern_coded[i]) {		
if(VOP_coding_type=="I" || (VOP_coding_type=="P" && !not_coded && mb_type=="I")) {		
dct_dc_size_alpha	2-12	vlclbf
if(dct_dc_size_alpha != 0)		
dct_dc_differential	1-11	uimsbf
} else {		
First DCT coefficient	2-24	
}		
while(nextbits() != lastcoef)		
Subsequent DCT coefficients	3-24	vlclbf
}		
}		

6.2.9 Still Texture Object

StillTextureObject() {	**No. of bits**	**Mnemonic**
still_texture_object_start_code	32	bslbf
texture_object_id	16	uimsbf
marker_bit	1	bslbf
wavelet_filter_type	1	uimsbf
wavelet_download	1	uimsbf
if (wavelet_download == "1"){		
download_wavelet_filters( )		
}		
wavelet_stuffing quantization_type	2	uimsbf
scan_direction	1	bslbf
start_code_enable	1	bslbf
wavelet_stuffing	3	uimsbf
texture_object_layer_shape	2	uimsbf
wavelet_stuffing	3	uimsbf
if(texture_object_layer_shape == "00"){		
texture_object_layer_width	15	uimsbf
marker_bit	1	bslbf
texture_object_layer_height	15	uimsbf
marker_bit	1	bslbf
}		
else {		
horizontal_ref	15	imsbf
marker_bit	1	bslbf
vertical_ref	15	imsbf
marker_bit	1	bslbf
object_width	15	uimsbf

marker_bit	1	bslbf
object_height	15	uimsbf
marker_bit	1	bslbf
shape_object_decoding ( )		
}		
for (color = “y”, “u”, “v”){		
wavelet_dc_decode()		
}		
if(quantization_type == 1){		
TextureLayerSQ ( )		
}		
else if (quantization_type == 2){		
spatial_scalability_levels	5	uimsbf
if (start_code_enable == 1) {		
do {		
TextureSpatialLayerMQ ( )		
} while (nextbits() == texture_spatial_layer_start_code)		
} else {		
for (i =0; i<spatial_scalability_levels; i++)		
TextureSpatialLayerMQNSC ( )		
}		
}		
else if (quantization_type == 3){		
TextureSNRLayerMQNSC ( )		
for (color = “y”, “u”, “v”)		
do{		
quant_byte	8	uimsbf
} while(quant_byte >>7)		
max_bitplanes	5	uimsbf
if (start_code_enable == 1) {		
if (scan_direction == 0) {		
do {		
TextureSNRLayerBQ ( )		
} while (nextbits() == texture_snr_layer_start_code)		
} else {		
do {		
TextureSpatialLayerBQ ( )		
} while (nextbits() == texture_spatial_layer_start_code)		
}		
} else {		
if (scan_direction == 0) {		

for (i =0; i<max_bitplane; i++)		
TextureSNRLayerBQNSC ()		
} else {		
for (i =0; i<wavelet_decomposition_levels; i++)		
TextureSpatialLayerBQNSC ( )		
}		
}		
}		

6.2.9.1 TextureLayerSQ

TextureLayerSQ() {	**No. of bits**	**Mnemonic**
For (color = “y”, “u”, “v”){		
do{		
quant_byte	8	uimsbf
} while(quant_byte >>7)		
do{		
root_max_alphabet_byte	8	uimsbf
} while (root_max_alphabet_byte >>7)		
marker_bit	1	bslbf
do{		
valz_max_alphabet_byte	8	uimsbf
} while (valz_max_alphabet_byte >>7)		
do{		
valnz_max_alphabet_byte	8	uimsbf
} while (valnz_max_alphabet_byte >>7)		
}		
if (scan_direction == 0) {		
for (i = 0; i<tree_blocks; i++)		
for (color = “y”, “u”, “v”)		
arith_decode_highbands()		
} else {		
if (start_code_enable) {		
do {		
TextureSpatialLayerSQ()		
} while (nextbits() == texture_spatial_layer_start_code)		
} else {		
for (i = 0; i< wavelet_decomposition_levels; i++)		
TextureSpatialLayerSQNSC()		
}		
}		
}		

6.2.9.2 TextureSpatialLayerSQ

TextureSpatialLayerSQ() {	**No. of bits**	**Mnemonic**
texture_spatial_layer_start_code	32	bslbf
texture_spatial_layer_id	5	uimsbf
TextureSpatialLayerSQNSC()		
next_start_code ( )		
}		

6.2.9.3 TextureSpatialLayerSQNSC

TextureSpatialLayerSQNSC() {	**No. of bits**	**Mnemonic**
for (color = “y”, “u”, “v”) {		
if (texture_spatial_layer_id != 0 || color != “u”, “v”)		
arith_decode_highbands()		
}		
}		

6.2.9.4 TextureSpatialLayerMQ

TextureSpatialLayerMQ() {	**No. of bits**	**Mnemonic**
texture_spatial_layer_start_code	32	bslbf
texture_spatial_layer_id	5	uimsbf
snr_scalability_levels	5	uimsbf
do {		
TextureSNRLayerMQ( )		
} while (nextbits() == texture_snr_layer_start_code)		
next_start_code ( )		
}		

6.2.9.5 TextureSpatialLayerMQNSC

TextureSpatialLayerMQNSC() {	**No. of bits**	**Mnemonic**
snr_scalability_levels	5	uimsbf
for (i =0; i<snr_scalability_levels; i++)		
TextureSNRLayerMQNSC ( )		
}		

6.2.9.6 TextureSNRLayerMQ

TextureSNRLayerMQ(){		
texture_snr_layer_start_code	32	bslbf
texture_snr_layer_id	5	uimsbf
TextureSNRLayerMQNSC()		
next_start_code ( )		
}		

6.2.9.7 TextureSNRLayerMQNSC

TextureSNRLayerMQNSC(){		
For (color = “y”, “u”, “v”){		
do{		
quant_byte	8	uimsbf
} while(quant_byte >>7)		
do{		
root_max_alphabet_byte	8	uimsbf
} while (root_max_alphabet_byte >>7)		
marker_bit	1	bslbf
do{		
valz_max_alphabet_byte	8	uimsbf
} while (valz_max_alphabet_byte >>7)		
do{		
valnz_max_alphabet_byte	8	uimsbf
} while (valnz_max_alphabet_byte >>7)		
}		
if (scan_direction == 0) {		
for (i = 0; i<tree_blocks; i++)		
for (color = “y”, “u”, “v”)		
arith_decode_highbands()		
} else {		
for (i = 0; i< spatial_layers; i++) {		
for (color = “y”, “u”, “v”) {		
if (wavelet_decomposition_layer_id != 0 || color != “u”, “v”)		
arith_decode_highbands()		
}		
}		
}		
}		

6.2.9.8 TextureSpatialLayerBQ

TextureSpatialLayerBQ() {	**No. of bits**	**Mnemonic**
texture_spatial_layer_start_code	32	bslbf
texture_spatial_layer_id	5	uimsbf
for (i=0; i<max_bitplanes; i++)		
TextureBitPlaneBQNSC()		
next_start_code ( )		
}		

6.2.9.9 TextureSpatialLayerBQNSC

TextureSpatialLayerBQNSC() {	**No. of bits**	**Mnemonic**
for (i=0; i<max_bitplanes; i++)		
TextureBitPlaneBQNSC()		
}		

6.2.9.10 TextureBitPlaneBQNSC

TextureBitPlaneBQNSC() {	**No. of bits**	**Mnemonic**
for (color = “y”, “u”, “v”)		
if (wavelet_decomposition_layer_id != 0 || color != “u”, “v”) {		
snr_all_zero	1	bslbf
if(snr_all_zero == 0)		
arith_decode_highbands_bilevel()		
}		
}		
}		

6.2.9.11 TextureSNRLayerBQ

TextureSNRLayerBQ() {	**No. of bits**	**Mnemonic**
texture_snr_layer_start_code	32	bslbf
texture_snr_layer_id	5	uimsbf
TextureSNRLayerBQNSC()		
next_start_code ( )		
}		

6.2.9.12 TextureSNRLayerBQNSC

TextureSNRLayerBQNSC() {	**No. of bits**	**Mnemonic**
for (color = “y”, “u”, “v”)		
snr_all_zero[color]	1	bslbf
for (i=0; i<wavelet_decomposition_levels; i++) {		
for (color = “y”, “u”, “v”) {		
if(snr_all_zero[color] == 0)		
arith_decode_highbands_bilevel()		
}		
}		
}		

6.2.9.13 DownloadWaveletFilters

download_wavelet_filters( ){	**No. of bits**	**Mnemonic**
lowpass_filter_length	4	uimsbf
highpass_filter_length	4	uimsbf
do{		
if (wavelet_filter_type == 0) {		
filter_tap_integer	16	imsbf
marker_bit	1	bslbf
} else {		
filter_tap_float_high	16	uimsbf
marker_bit	1	bslbf
filter_tap_float_low	16	uimsbf
marker_bit	1	bslbf
}		
} while (lowpass_filter_length--)		
do{		
if (wavelet_filter_type == 0){		
filter_tap_integer	16	imsbf
marker_bit	1	bslbf
} else {		
filter_tap_float_high	16	uimsbf
marker_bit	1	bslbf
filter_tap_float_low	16	uimsbf
marker_bit	1	bslbf
}		
} while (highpass_filter_length--)		
if (wavelet_filter_type == 0) {		
integer_scale	16	uimsbf
marker_bit		
}		
}		

6.2.9.14 Wavelet dc decode

wavelet_dc_decode() {	No. of bits	Mnemonic
mean	8	uimsbf
do{		
quant_dc_byte	8	uimsbf
} while(quant_dc_byte >>7)		
do{		
band_offset_byte	8	uimsbf
} while (band_offset_byte >>7)		
do{		
band_max_byte	8	uimsbf
} while (band_max_byte >>7)		
arith_decode_dc()		
}		

6.2.9.15 Wavelet higher bands decode

wavelet_ higher_bands_decode() {	No. of bits	Mnemonic
do{		
root_max_alphabet_byte	8	uimsbf
} while (root_max_alphabet_byte >>7)		
marker_bit	1	bslbf
do{		
valz_max_alphabet_byte	8	uimsbf
} while (valz_max_alphabet_byte >>7)		
do{		
valnz_max_alphabet_byte	8	uimsbf
} while (valnz_max_alphabet_byte >>7)		
arith_decode_highbands()		
}		

wavelet_ higher_bands_decode_bilevel() {	No. of bits	Mnemonic
arith_decode_highbands_bilevel()		
}		

6.2.9.16 Shape Object Decoding

shape_object_decoding() {	**No. of bits**	**Mnemonic**
change_conv_ratio_disable	1	bslbf
STO_constant_alpha	1	bslbf
if (STO_constant_alpha)		
STO_constant_alpha_value	8	bslbf
for (i=0; i<((object_width*object_height)/(16*16)); i++) {		
bab_type	1-2	vlclbf
if (bab_type ==4) {		
if (!change_conv_ratio_disable)		
conv_ratio	1-2	vlcbf
scan_type	1	bslbf
binary_arithmetic_decode()		
}		
}		
}		

6.2.10 Mesh Object

MeshObject() {	**No. of bits**	**Mnemonic**
mesh_object_start_code	32	bslbf
do{		
MeshObjectPlane()		
} while (nextbits() == **mesh_object_plane_start_code** || nextbits() != ‘0000 0000 0000 0000 0000 0001’)		
}		

6.2.10.1 Mesh Object Plane

MeshObjectPlane() {	**No. of bits**	**Mnemonic**
MeshObjectPlaneHeader()		
MeshObjectPlaneData()		
}		

MeshObjectPlaneHeader() {	**No. of bits**	**Mnemonic**
if (nextbits() == ‘0000 0000 0000 0000 0000 0001’)		
mesh_object_plane_start_code	32	bslbf
is_intra	1	bslbf
mesh_mask	1	bslbf
Temporal_header()		
}		

MeshObjectPlaneData() {	**No. of bits**	**Mnemonic**
if (mesh_mask == 1) {		
if (is_intra == 1)		
mesh_geometry()		
else		
mesh_motion()		
}		
next_start_code()		
}		

6.2.10.2 Mesh geometry

mesh_geometry() {	**No. of bits**	**Mnemonic**
mesh_type _code	2	bslbf
if (mesh_type_code == ‘00’) {		
nr_of_mesh_nodes_hor	10	uimsbf
nr_of_mesh_nodes_vert	10	uimsbf
marker_bit	1	uimsbf
mesh_rect_size_hor	8	uimsbf
mesh_rect_size_vert	8	uimsbf
triangle_split_code	2	bslbf
}		
else if (mesh_type_code == ‘01’) {		
nr_of_mesh_nodes	16	uimsbf
marker_bit	1	uimsbf
nr_of_boundary_nodes	10	uimsbf
marker_bit	1	uimsbf
node0_x	10	uimsbf
node0_y	10	uimsbf
marker_bit	1	uimsbf
for (n=1; n < nr_of_mesh_nodes; n++) {		
delta_x_len_vlc	2-9	vlcbf
if (delta_x_len_vlc)		
delta_x	1-11	vlcbf
delta_y_len_vlc	2-9	vlcbf
if (delta_y_len_vlc)		
delta_y	1-11	vlcbf
}		
}		
}		

6.2.10.3 Mesh motion

mesh_motion() {	**No. of bits**	**Mnemonic**
motion_range_code	3	bslbf
for (n=0; n <nr_of_mesh_nodes; n++) {		
node_motion_vector_flag	1	bslbf
if (node_motion_vector_flag == ‘0’) {		
delta_mv_x_vlc	1-13	vlcbf
if ((motion_range_code != 1) && (delta_mv_x_vlc != 0))		
delta_mv_x_res	1-6	vlcbf
delta_mv_y_vlc	1-13	vlcbf
if ((motion_range_code != 1) && (delta_mv_y_vlc != 0))		
delta_mv_y_res	1-6	vlcbf
}		
}		
}		

6.2.11 Face Object

fba_object() {	**No. of bits**	**Mnemonic**
face_object_start_code	32	bslbf
do {		
fba_object_plane()		
} while(!((nextbits_bytealigned() == ‘000 0000 0000 0000 0000 0000’) && (nextbits_bytealigned() != face_object_plane_start_code)))		
}		

6.2.11.1 Face Object Plane

fba_object_plane() {	No. of bits	Mnemonic
fba_object_plane_header()		
fba_object_plane_data()		
}		

fba_object_plane_header() {	No. of bits	Mnemonic
if (nextbits_bytealigned() == ‘000 0000 0000 0000 0000 0000’) {		

next_start_code()		
fba_object_plane_start_code	32	bslbf
}		
is_intra	1	bslbf
fba_object_mask	2	bslbf
temporal_header()		
}		

fba_object_plane_data() {		
if(fba_object_mask &'01') {		
if(is_intra) {		
fap_quant	5	uimsbf
for (group_number = 1; group_number <= 10; group_number++) {		
marker_bit	1	uimsbf
fap_mask_type	2	bslbf
if(fap_mask_type == '01'|| fap_mask_type == '10')		
fap_group_mask[group_number]	2-16	vlcbf
}		
fba_object_coding_type	2	bslbf
if(fba_object_coding_type == 0) {		
is_i_new_max	1	bslbf
is_i_new_min	1	bslbf
is_p_new_max	1	bslbf
is_p_new_min	1	bslbf
decode_new_minmax()		
decode_ifap()		
}		
if(fba_object_coding_type == 1)		
decode_i_segment()		
}		
else {		
if(fba_object_coding_type == 0)		
decode_pfap()		
if(fba_object_coding_type == 1)		
decode_p_segment()		
}		
}		
}		

temporal_header() {		
if (is_intra) {		
is_frame_rate	1	bslbf
if(is_frame_rate)		
decode_frame_rate()		
is_time_code	1	bslbf
if (is_time_code)		
time_code	18	bsIbf
}		
skip_frames	1	bslbf
if(skip_frames)		
decode_skip_frames()		
}		

6.2.11.2 Decode frame rate and skip frames

decode_frame_rate(){	**No. of bits**	**Mnemonic**
frame_rate	8	uimsbf
seconds	4	uimsbf
frequency_offset	1	uimsbf
}		

decode_skip_frames(){	**No. of bits**	**Mnemonic**
do{		
number_of_frames_to_skip	4	uimsbf
} while (number_of_frames_to_skip = “1111”)		
}		

6.2.11.3 Decode new minmax

decode_new_minmax() {	**No. of bits**	**Mnemonic**
if (is_i_new_max) {		
for (group_number = 2, j=0, group_number <= 10, group_number++)		
for (i=0; i < NFAP[group_number]; i++, j++) {		
if (!(i & 0x3))		
marker_bit	1	uimsbf
if (fap_group_mask[group_number] & (1 <<i))		
i_new_max[j]	5	uimsbf
}		
if (is_i_new_min) {		
for (group_number = 2, j=0, group_number <= 10, group_number++)		
for (i=0; i < NFAP[group_number]; i++, j++) {		
if (!(i & 0x3))		
marker_bit	1	uimsbf
if (fap_group_mask[group_number] & (1 <<i))		
i_new_min[j]	5	uimsbf
}		
if (is_p_new_max) {		
for (group_number = 2, j=0, group_number <= 10, group_number++)		
for (i=0; i < NFAP[group_number]; i++, j++) {		
if (!(i & 0x3))		
marker_bit	1	uimsbf
if (fap_group_mask[group_number] & (1 <<i))		
p_new_max[j]	5	uimsbf
}		
if (is_p_new_min) {		
for (group_number = 2, j=0, group_number <= 10, group_number++)		
for (i=0; i < NFAP[group_number]; i++, j++) {		
if (!(i & 0x3))		
marker_bit	1	uimsbf
if (fap_group_mask[group_number] & (1 <<i))		
p_new_min[j]	5	uimsbf
}		
}		
}		

6.2.11.4 Decode ifap

decode_ifap(){	No. of bits	Mnemonic
for (group_number = 1, j=0; group_number <= 10; group_number++) {		
if (group_number == 1) {		
if(fap_group_mask[1] & 0x1)		
decode_viseme()		
if(fap_group_mask[1] & 0x2)		
decode_expression()		
} else {		
for (i= 0; i<NFAP[group_number]; i++, j++) {		
if(fap_group_mask[group_number] & (1 << i)) {		
aa_decode(ifap_Q[j],ifap_cum_freq[j])		
}		
}		
}		
}		
}		

6.2.11.5 Decode pfap

decode_pfap(){	No. of bits	Mnemonic
for (group_number = 1, j=0; group_number <= 10; group_number++) {		
if (group_number == 1) {		
if(fap_group_mask[1] & 0x1)		
decode_viseme()		
if(fap_group_mask[1] & 0x2)		
decode_expression()		
} else {		
for (I= 0; i<NFAP[group_number]; i++, j++) {		
if(fap_group_mask[group_number] & (1 << i)) {		
aa_decode(pfap_diff[j], pfap_cum_freq[j])		
}		
}		
}		
}		
}		

6.2.11.6 Decode viseme and expression

decode_viseme() {	**No. of bits**	**Mnemonic**
aa_decode(viseme_select1Q, viseme_select1_cum_freq)		vlclbf
aa_decode(viseme_select2Q, viseme_select2_cum_freq)		vlclbf
aa_decode(viseme_blendQ, viseme_blend_cum_freq)		vlclbf
viseme_def	1	bslbf
}		

decode_expression() {	**No. of bits**	**Mnemonic**
aa_decode(expression_select1Q, expression_select1_cum_freq)		vlclbf
aa_decode(expression_intensity1Q, expression_intensity1_cum_freq)		vlclbf
aa_decode(expression_select2Q, expression_select2_cum_freq)		vlclbf
aa_decode(expression_intensity2Q, expression_intensity2_cum_freq)		vlclbf
aa_decode(expression_blendQ, expression_blend_cum_freq)		vlclbf
init_face	1	bslbf
expression_def	1	bslbf
}		

6.2.11.7 Face Object Plane Group

face_object_plane_group() {	**No. of bits**	**Mnemonic**
face_object_plane_start_code	32	bslbf
is_intra	1	bslbf
if (is_intra) {		
face_paramset_mask	2	bslbf
is_frame_rate	1	bslbf
if(is_frame_rate)		
decode_frame_rate()		
is_time_code	1	bslbf
if(is_time_code)		
time_code	18	
skip_frames	1	bslbf
if(skip_frames)		
decode_skip_frames()		
if(face_paramset_mask ==’01’) {		
fap_quant_index	5	uimsbf
for (group_number = 1 to 10) {		
marker_bit	1	uimsbf
fap_mask_type	2	bslbf
if(fap_mask_type == ‘01’|| fap_mask_type == ‘10’)		
fap_group_mask[group_number]	2-16	vlcbf
}		
decode_i_segment()		
} else {		
face_object_group_prediction()		
}		
next_start_code()		
}		

6.2.11.8 Face Object Group Prediction

face_object_group_prediction() {	**No. of bits**	**Mnemonic**
skip_frames	1	bslbf
if(skip_frames)		
decode_skip_frames()		
if(face_paramset_mask ==’01’) {		
decode_p_segment()		
}		
}		

6.2.11.9 Decode i_segment

decode_i_segment(){	**No. of bits**	**Mnemonic**
for (group_number= 1, j=0; group_number<= 10; group_number++) {		
if (group_number == 1) {		
if(fap_group_mask[1] & 0x1)		
decode_i_viseme_segment()		
if(fap_group_mask[1] & 0x2)		
decode_i_expression_segment()		
} else {		
for(i=0; i<NFAP[group_number]; i++, j++) {		
if(fap_group_mask[group_number] & (1 << i)) {		
decode_i_dc(dc_Q[j])		
decode_ac(ac_Q[j])		
}		
}		
}		
}		
}		

6.2.11.10 Decode p_segment

decode_p_segment(){	**No. of bits**	**Mnemonic**
for (group_number = 1, j=0; group_number <= 10; group_number++) {		
if (group_number == 1) {		
if(fap_group_mask[1] & 0x1)		
decode_p_viseme_segment()		
if(fap_group_mask[1] & 0x2)		
decode_p_expression_segment()		
} else {		
for (i=0; i<NFAP[group_number]; i++, j++) {		
If(fap_group_mask[group_number] & (1 << i)) {		
decode_p_dc(dc_Q[j])		
decode_ac(ac_Q[j])		
}		
}		
}		
}		
}		

6.2.11.11 Decode viseme and expression

decode_i_viseme_segment(){	**No. of bits**	**Mnemonic**
viseme_segment_select1Q[0]	4	uimsbf
viseme_segment_select2Q[0]	4	uimsbf
viseme_segment_blendQ[0]	6	uimsbf
viseme_segment_def[0]	1	bslbf
for (k=1; k<16, k++) {		
viseme_segment_select1Q_diff[k]		vlclbf
viseme_segment_select2Q_diff[k]		vlclbf
viseme_segment_blendQ_diff[k]		vlclbf
viseme_segment_def[k]	1	bslbf
}		
}		

decode_p _viseme_segment(){	**No. of bits**	**Mnemonic**
for (k=0; k<16, k++) {		
viseme_segment_select1Q_diff[k]		vlclbf
viseme_segment_select2Q_diff[k]		vlclbf
viseme_segment_blendQ_diff[k]		vlclbf
viseme_segment_def[k]	1	bslbf
}		
}		

decode_i_expression_segment(){	**No. of bits**	**Mnemonic**
expression_segment_select1Q[0]	4	uimsbf
expression_segment_select2Q[0]	4	uimsbf
expression_segment_intensity1Q[0]	6	uimsbf
expression_segment_intensity2Q[0]	6	uimsbf
expression_segment_init_face[0]	1	bslbf
expression_segment_def[0]	1	bslbf
for (k=1; k<16, k++) {		
expression_segment_select1Q_diff[k]		vlclbf
expression_segment_select2Q_diff[k]		vlclbf
expression_segment_intensity1Q_diff[k]		vlclbf
expression_segment_intensity2Q_diff[k]		vlclbf
expression_segment_init_face[k]	1	bslbf
expression_segment_def[k]	1	bslbf
}		
}		

decode_p _expression_segment(){	**No. of bits**	**Mnemonic**
for (k=0; k<16, k++) {		
expression_segment_select1Q_diff[k]		vlclbf
expression_segment_select2Q_diff[k]		vlclbf
expression_segment_intensity1Q_diff[k]		vlclbf
expression_segment_intensity2Q_diff[k]		vlclbf
expression_segment_init_face[k]	1	bslbf
expression_segment_def[k]	1	bslbf
}		
}		

decode_i_dc(dc_Q) {	**No. of bits**	**Mnemonic**
dc_Q	16	simsbf
if(dc_Q == -256*128)		
dc_Q	31	simsbf
}		

decode_p_dc(dc_Q_diff) {	**No. of bits**	**Mnemonic**
dc_Q_diff		vlclbf
dc_Q_diff = dc_Q_diff- 256		
if(dc_Q_diff == -256)		
dc_Q_diff	16	simsbf
if(dc_Q == 0-256*128)		
dc_Q_diff	32	simsbf
}		

decode_ac(ac_Q[i]) {	No. of bits	Mnemonic
this = 0		
next = 0		
while(next < 15) {		
count_of_runs		vlclbf
if (count_of_runs == 15)		
next = 16		
else {		
next = this+1+count_of_runs		
for (n=this+1; n<next; n++)		
ac_Q[i][n] = 0		
ac_Q[i][next]		vlclbf
if(ac_Q[i][next] == 256)		
decode_i_dc(ac_Q[i][next])		
else		
ac_Q[i][next] = ac_Q[i][next]-256		
this = next		
}		
}		
}		

6.3 Visual bitstream semantics

6.3.1 Semantic rules for higher syntactic structures

This clause details the rules that govern the way in which the higher level syntactic elements may be combined together to produce a legal bitstream. Subsequent clauses detail the semantic meaning of all fields in the video bitstream.

6.3.2 Visual Object Sequence and Visual Object

visual_object_sequence_start_code -- The visual_session_start_code is the bit string '000001B0' in hexadecimal. It initiates a visual session.

visual_object_sequence_end_code -- The visual_session_end_code is the bit string '000001B1' in hexadecimal. It terminates a visual session.

visual_object_start_code -- The visual_object_start_code is the bit string '000001B5' in hexadecimal. It initiates a visual object.

profile_and_level_indication – This is an 8-bit integer used to signal the profile and level identification. The meaning of the bits is given in Annex G.

visual_object_type -- The visual_object_type is a 4-bit code given in Table 6-4 which identifies the type of the visual object.

Table 6-4 Meaning of visual object type

code	**visual object type**
0000	reserved
0001	video ID
0010	still texture ID
0011	mesh ID
0100	face ID
0101	reserved
:	:
:	:
1111	reserved

is_visual_object_identifier – This is a 1-bit code which when set to '1' indicates that version identification and priority is specified for the visual object. When set to '0', no version identification or priority needs to be specified.

visual_object_verid – This is a 4-bit code which identifies the version number of the visual object. It takes values between 1 and 15, a zero value is disallowed.

visual_object_priority – This is a 3-bit code which specifies the priority of the visual object. It takes values between 1 and 7, with 1 representing the highest priority and 7, the lowest priority. The value of zero is reserved.

video_signal_type - A flag which if set to '1' indicates the presence of video_signal_type information.

video_format -- This is a three bit integer indicating the representation of the pictures before being coded in accordance with this specification. Its meaning is defined in Table 6-6. If the video_signal_type() is not present in the bitstream then the video format may be assumed to be "Unspecified video format".

Table 6-6. Meaning of video_format

video_format	Meaning
000	Component
001	PAL
010	NTSC
011	SECAM
100	MAC
101	Unspecified video format
110	Reserved
111	Reserved

colour_description -- A flag which if set to '1' indicates the presence of colour_primaries, transfer_characteristics and matrix_coefficients in the bitstream.

colour_primaries -- This 8-bit integer describes the chromaticity coordinates of the source primaries, and is defined in Table 6-7.

Table 6-7. Colour Primaries

Value	Primaries		
0	(forbidden)		
1	Recommendation ITU-R BT.709		
	primary	x	y
	green	0,300	0,600
	blue	0,150	0,060
	red	0,640	0,330
2	Unspecified Video Image characteristics are unknown.		
3	Reserved		
4	Recommendation ITU-R BT.470-2 System M		
	primary	x	y
	green	0,21	0,71
	blue	0,14	0,08
	red	0,67	0,33
5	Recommendation ITU-R BT.470-2 System B, G		
	primary	x	y
	green	0,29	0,60
	blue	0,15	0,06
	red	0,64	0,33

6	SMPTE 170M primary x y green 0,310 0,595 blue 0,155 0,070 red 0,630 0,340
7	SMPTE 240M (1987) primary x y green 0,310 0,595 blue 0,155 0,070 red 0,630 0,340
8	Generic film (colour filters using Illuminant C) primary x y green 0,243 0,692 (Wratten 58) blue 0,145 0,049 (Wratten 47) red 0,681 0,319 (Wratten 25)
9-255	Reserved

In the case that video_signal_type() is not present in the bitstream or colour_description is zero the chromaticity is assumed to be that corresponding to colour_primaries having the value 5.

transfer_characteristics -- This 8-bit integer describes the opto-electronic transfer characteristic of the source picture, and is defined in Table 6-8.

Table 6-8. Transfer Characteristics

Value	Transfer Characteristic
0	(forbidden)
1	Recommendation ITU-R BT.709 $V = 1{,}099\ L_C^{0{,}45} - 0{,}099$ for $1 \geq L_C \geq 0{,}018$ $V = 4{,}500\ L_C$ for $0{,}018 > L_C \geq 0$
2	Unspecified Video Image characteristics are unknown.
3	reserved
4	Recommendation ITU-R BT.470-2 System M Assumed display gamma 2,2
5	Recommendation ITU-R BT.470-2 System B, G Assumed display gamma 2,8
6	SMPTE 170M $V = 1{,}099\ L_C^{0{,}45} - 0{,}099$ for $1 \geq L_C \geq 0{,}018$ $V = 4{,}500\ L_C$ for $0{,}018 > L_C \geq 0$
7	SMPTE 240M (1987) $V = 1{,}1115\ L_C^{0{,}45} - 0{,}1115$ for $L_C \geq 0{,}0228$ $V = 4{,}0\ L_C$ for $0{,}0228 > L_C$
8	Linear transfer characteristics i.e. $V = L_C$
9	Logarithmic transfer characteristic (100:1 range) $V = 1.0\text{-}\mathrm{Log}_{10}(Lc)/2$ for $1 = L_C = 0.01$ $V = 0.0$ for $0.01 > L_C$
10	Logarithmic transfer characteristic (316.22777:1 range) $V = 1.0\text{-}\mathrm{Log}_{10}(Lc)/2.5$ for $1 = L_C = 0.0031622777$ $V = 0.0$ for $0.0031622777 > L_C$

11-255	reserved

In the case that video_signal_type() is not present in the bitstream or colour_description is zero the transfer characteristics are assumed to be those corresponding to transfer_characteristics having the value 5.

matrix_coefficients -- This 8-bit integer describes the matrix coefficients used in deriving luminance and chrominance signals from the green, blue, and red primaries, and is defined in Table 6-9.

video_range -- This one-bit flag indicates the black level and range of the luminance and chrominance signals.

In this table:

E'_Y is analogue with values between 0 and 1

E'_{PB} and E'_{PR} are analogue between the values -0,5 and 0,5

E'_R, E'_G and E'_B are analogue with values between 0 and 1

White is defined as $E'_y=1$, $E'_{PB}=0$, $E'_{PR}=0$; $E'_R = E'_G = E'_B = 1$.

Y, Cb and Cr are related to E'_Y, E'_{PB} and E'_{PR} by the following formulae:

if **video_range=0:**

$Y = (219 * 2^{n-8} * E'_Y) + 2^{n-4}$.

$Cb = (224 * 2^{n-8} * E'_{PB}) + 2^{n-1}$

$Cr = (224 * 2^{n-8} * E'_{PR}) + 2^{n-1}$

if **video_range=1:**

$Y = (2^n * E'_Y)$

$Cb = (2^n * E'_{PB}) + 2^{n-1}$

$Cr = (2^n * E'_{PR}) + 2^{n-1}$

for n bit video.

For example, for 8 bit video,

video_range=0 gives a range of Y from 16 to 235, Cb and Cr from -112 to +112;

video_range=1 gives a range of Y from 0 to 255, Cb and Cr from -128 to +127.

Table 6-9. Matrix Coefficients

Value	Matrix
0	(forbidden)
1	Recommendation ITU-R BT.709 $E¢_Y = 0,7154\ E¢_G + 0,0721\ E¢_B + 0,2125\ E¢_R$ $E¢_{PB} = -0,386\ E¢_G + 0,500\ E¢_B - 0,115\ E¢_R$ $E¢_{PR} = -0,454\ E¢_G - 0,046\ E¢_B + 0,500\ E¢_R$
2	Unspecified Video Image characteristics are unknown.
3	reserved
4	FCC $E¢_Y = 0,59\ E¢_G + 0,11\ E¢_B + 0,30\ E¢_R$ $E¢_{PB} = -0,331\ E¢_G + 0,500\ E¢_B - 0,169\ E¢_R$ $E¢_{PR} = -0,421\ E¢_G - 0,079\ E¢_B + 0,500\ E¢_R$

5	Recommendation ITU-R BT.470-2 System B, G $E¢_{Y} = 0{,}587\ E¢_{G} + 0{,}114\ E¢_{B} + 0{,}299\ E¢_{R}$ $E¢_{PB} = -0{,}331\ E¢_{G} + 0{,}500\ E¢_{B} -0{,}169\ E¢_{R}$ $E¢_{PR} = -0{,}419\ E¢_{G} - 0{,}081\ E¢_{B} + 0{,}500\ E¢_{R}$
6	SMPTE 170M $E¢_{Y} = 0{,}587\ E¢_{G} + 0{,}114\ E¢_{B} + 0{,}299\ E¢_{R}$ $E¢_{PB} = -0{,}331\ E¢_{G} + 0{,}500\ E¢_{B} -0{,}169\ E¢_{R}$ $E¢_{PR} = -0{,}419\ E¢_{G} - 0{,}081\ E¢_{B} + 0{,}500\ E¢_{R}$
7	SMPTE 240M (1987) $E¢_{Y} = 0{,}701\ E¢_{G} + 0{,}087\ E¢_{B} + 0{,}212\ E¢_{R}$ $E¢_{PB} = -0{,}384\ E¢_{G} + 0{,}500\ E¢_{B} -0{,}116\ E¢_{R}$ $E¢_{PR} = -0{,}445\ E¢_{G} - 0{,}055\ E¢_{B} + 0{,}500\ E¢_{R}$
8-255	reserved

In the case that video_signal_type() is not present in the bitstream or colour_description is zero the matrix coefficients are assumed to be those corresponding to matrix_coefficients having the value 5.

In the case that video_signal_type() is not present in the bitstream, video_range is assumed to have the value 0 (a range of Y from 16 to 235 for 8-bit video).

6.3.2.1 User data

user_data_start_code -- The user_data_start_code is the bit string ‘000001B2’ in hexadecimal. It identifies the beginning of user data. The user data continues until receipt of another start code.

user_data -- This is an 8 bit integer, an arbitrary number of which may follow one another. User data is defined by users for their specific applications. In the series of consecutive user_data bytes there shall not be a string of 23 or more consecutive zero bits.

6.3.3 Video Object

video_object_start_code -- The video_object_start_code is a string of 32 bits. The first 27 bits are ‘0000 0000 0000 0000 0000 0001 000‘ in binary and the last 5-bits represent one of the values in the range of ‘00000’ to ‘11111’ in binary. The video_object_start_code marks a new video object.

video_object_id -- This is given by the last 5-bits of the video_object_start_code. The video_object_id uniquely identifies a video object.

6.3.4 Video Object Layer

video_object_layer_start_code -- The video_object_layer_start_code is a string of 32 bits. The first 28 bits are ‘0000 0000 0000 0000 0000 0001 0010‘ in binary and the last 4-bits represent one of the values in the range of ‘0000’ to ‘1111’ in binary. The video_object_layer_start_code marks a new video object layer.

short_video_header – The short_video_header bit is a flag which is set when an abbreviated header format is used for video content. This indicates video data which begins with a short_video_start_marker rather than a longer start code such as visual_object_ start_code. The short header format is included herein to provide forward compatibility with video codecs designed using the earlier video coding specification ITU-T Recommendation H.263. All decoders which support video objects shall support both header formats (short_video_header equal to 0 or 1) for the subset of video tools that is expressible in either form.

video_plane_with_short_header() – This is a syntax layer encapsulating a video plane which has only the limited set of capabilities available using the short header format.

short_video_start_marker – This is a 22-bit start marker containing the value '0000 0000 0000 0000 1000 00'. It is used to mark the location of a video plane having the short header format. short_video_start_marker shall be byte aligned by the insertion of zero to seven zero-valued bits as necessary to achieve byte alignment prior to short_video_start_marker.

short_video_end_marker – This is a 22-bit end of sequence marker containing the value '0000 0000 0000 0000 1111 11'. It is used to mark the end of a sequence of video_plane_with_short_header(). short_video_end_marker may (and should) be byte aligned by the insertion of zero to seven zero-valued bits to achieve byte alignment prior to short_video_end_marker.

zero_bit – This is a single bit having the value zero ('0').

random_accessible_vol -- This flag may be set to "1" to indicate that every VOP in this VOL is individually decodable. If all of the VOPs in this VOL are intra-coded VOPs and some more conditions are satisfied then random_accessible_vol may be set to "1". random_accessible_vol may be omitted from the bitstream (by setting random_access_flag to "0") in which case it shall be assumed to have the value zero. The flag random_accessible_vol is not used by the decoding process. random_accessible_vol is intended to aid random access or editing capability. This shall be set to "0" if any of the VOPs in the VOL are non-intra coded or certain other conditions are not fulfilled.

is_visual_object_identifier – This is a 1-bit code which when set to '1' indicates that version identification and priority is specified for the visual object layer. When set to '0', no version identification or priority needs to be specified.

video_object_layer_verid – This is a 4-bit code which identifies the version number of the visual object layer. It takes values between 1 and 15, a zero value is disallowed. If both visual_object_verid and video_object_layer_verid exist, the semantics of visual_object_layer_verid supersedes the other.

video_object_layer_priority – This is a 3-bit code which specifies the priority of the video object layer. It takes values between 1 and 7, with 1 representing the highest priority and 7, the lowest priority. The value of zero is reserved.

vol_control_parameters – This a one-bit flag which when set to '1' indicates presence of following vol control parameters: aspect_ratio_info, VOP_rate_code ,bit_rate, vbv_buffer_size , chroma_format and low_delay.

aspect_ratio_info -- This is a four-bit integer which defines the value of aspect ratio.

VOP_rate_code -- This is a four-bit integer which defines the value of VOP rate.

bit_rate -- This is a 30-bit integer which specifies the bitrate of the bitstream measured in units of 400 bits/second, rounded upwards. The value zero is forbidden.

vbv_buffer_size -- The vbv_buffer_size is a 18-bit integer.

chroma_format - This is a two bit integer indicating the chrominance format as defined in the Table 6-5.

Table 6-5 Meaning of chroma_format

chroma_format	Meaning
00	reserved
01	4:2:0
10	reserved
11	reserved

low_delay - This flag, when set to 1, indicates that the sequence does not contain any B-VOPs, that the VOP reordering delay is not present.When set to 0, it indicates that the sequence may contain B-VOPs, and that the VOP reordering delay is present. This flag is not used during the decoding process and therefore can be ignored by decoders, but it is necessary to define and verify the compliance of low-delay bitstreams.

video_object_layer_id -- This is given by the last 4-bits of the video_object_layer_start_code. The video_object_layer_id uniquely identifies a video object layer.

video_object_layer_shape -- This is a 2-bit integer defined in Table 6-6. It identifies the shape type of a video object layer.

Table 6-6 Video Object Layer shape type

shape format	Meaning
00	rectangular
01	binary
10	binary only
11	grayscale

VOP_time_increment_resolution -- -- This is a 15-bit unsigned integer that indicates the number of evenly spaced subintervals, call ticks, within one modulo time. One modulo time represents the fixed interval of one second. The value zero is forbidden.

fixed_VOP_rate -- This is a one-bit flag which when set to ‘1’ indicates that all VOPs are coded with a fixed frame rate.

video_object_layer_width -- The video_object_layer_width is a 13-bit unsigned integer representing the width of the displayable part of the luminance component in pixel units. The width of the encoded luminance component of VOPs in macroblocks is (video_object_layer_width+15)/16. The displayable part is left-aligned in the encoded VOPs.

video_object_layer_height -- The video_object_layer_height is a 13-bit unsigned integer representing the height of the displayable part of the luminance component in pixel units. The height of the encoded luminance component of VOPs in macroblocks is (video_object_layer_height+15)/16. The displayable part is top-aligned in the encoded VOPs.

obmc_disable -- This is a one-bit flag which when set to ‘1’ disables overlapped block motion compensation.

sprite_enable -- This is a one-bit flag which when set to '1' indicates the presence of sprites.

sprite_width – This is a 13-bit unsigned integer which identifies the horizontal dimension of the sprite.

sprite_height -- This is a 13-bit unsigned integer which identifies the vertical dimension of the sprite.

sprite_left_coordinate – This is a 13-bit signed integer which defines the left-edge of the sprite.

sprite_top_coordinate – This is a 13-bit signed integer which defines the top edge of the sprite.

no_of_sprite_warping_points – This is a 6-bit unsigned integer which represents the number of points used in sprite warping. When its value is 0 and when sprite_enable is set to '1', warping is identity (stationary sprite) and no coordinates need to be coded. When its value is 4, a perspective transform is used. When its value is 1,2 or 3, an affine transform is used. Further, the case of value 1 is separated as a special case from that of values 2 or 3. Table 6-7 shows the various choices.

Table 6-7 Number of point and implied warping function

Number of points	**warping function**
0	Stationary
1	Translation
2,3	Affine
4	Perspective

sprite_warping_accuracy – This is a 2-bit code which indicates the quantization accuracy of motion vectors used in the warping process for sprites. Table 6-8 shows the meaning of various codewords

Table 6-8 Meaning of sprite warping accuracy codewords

code	**sprite_warping_accuracy**
00	½ pixel
01	¼ pixel
10	1/8 pixel
11	1/16 pixel

sprite_brightness_change – This is a one-bit flag which when set to '1' indicates a change in brightness during sprite warping, alternatively, a value of '0' means no change in brightness.

low_latency_sprite_enable -- This is a one-bit flag which when set to "1" indicates the presence of low_latency sprite, alternatively, a value of "0" means basic sprite.

quant_type -- This is a one-bit flag which when set to '1' that the first inverse quantisation method and when set to '0' indicates that the second inverse quantisation method is used for inverse quantisation of the DCT coefficients. Both inverse quantisation methods are described in section 7.3.4. For the first inverse quantization method, two matrices are used, one for intra blocks the other for non-intra blocks.

In MPEG-style quantization, two matrices are used, one for intra blocks the other for non-intra blocks.

The default matrix for intra blocks is:

8	17	18	19	21	23	25	27
17	18	19	21	23	25	27	28
20	21	22	23	24	26	28	30
21	22	23	24	26	28	30	32
22	23	24	26	28	30	32	35
23	24	26	28	30	32	35	38
25	26	28	30	32	35	38	41
27	28	30	32	35	38	41	45

The default matrix for non-intra blocks is:

16	17	18	19	20	21	22	23
17	18	19	20	21	22	23	24
18	19	20	21	22	23	24	25
19	20	21	22	23	24	26	27
20	21	22	23	25	26	27	28
21	22	23	24	26	27	28	30
22	23	24	26	27	28	30	31
23	24	25	27	28	30	31	33

no_gray_quant_update – This is a on-bit flag which is set to ‘1’ when a fixed quantiser is used for the decoding of grayscale alpha data. When this flag is set to ‘0’, the grayscale alpha quantiser is updated on every macroblock by generating it anew from the luminance quantiser value, but with an appropriate scale factor applied. See the description in Section 7.4.4.2.

load_intra_quant_mat_grayscale – This is a one-bit flag which is set to ‘1’ when intra_quant_mat_grayscale follows. If it is set to ‘0’ then there is no change in the quantisation matrix values that shall be used.

intra_quant_mat_grayscale – This is a list of 2 to 64 eight-bit unsigned integers defining the grayscale intra alpha quantisation matrix to be used. The semantics and the default quantisation matrix are identical to those of intra_quant_mat.

load_nonintra_quant_mat_grayscale – This is a one-bit flag which is set to ‘1’ when nonintra_quant_mat_grayscale follows. If it is set to ‘0’ then there is no change in the quantisation matrix values that shall be used.

nonintra_quant_mat_grayscale – This is a list of 2 to 64 eight-bit unsigned integers defining the grayscale nonintra alpha quantisation matrix to be used. The semantics and the default quantisation matrix are identical to those of nonintra_quant_mat.

load_intra_quant_mat -- This is a one-bit flag which is set to ‘1’ when intra_quant_mat follows. If it is set to ‘0’ then there is no change in the values that shall be used.

intra_quant_mat -- This is a list of 2 to 64 eight-bit unsigned integers. The new values are in zigzag scan order and replace the previous values. A value of 0 indicates that no more values are transmitted and the remaining, non-transmitted values are set equal to the last non-zero value. The first value shall always be 8 and is not used in the decoding process.

load_nonintra_quant_mat -- This is a one-bit flag which is set to ‘1’ when nonintra_quant_mat follows. If it is set to ‘0’ then there is no change in the values that shall be used.

nonintra_quant_mat -- This is a list of 2 to 64 eight-bit unsigned integers. The new values are in zigzag scan order and replace the previous values. A value of 0 indicates that no more values are transmitted and the remaining, non-transmitted values are set equal to the last non-zero value. The first value shall not be 0.

complexity_estimation_disable – This is a one-bit flag which disables complexity estimation header in each VOP.

estimation_method -- Setting of the of the estimation method,it is „00“ for Version 1.

shape_complexity_estimation_disable -- Flag to disable setting of shape parameters.

texture_complexity_estimation_set_1_disable -- Flag to disable texture parameter set 1.

intra_blocks -- Flag enabling transmission of the number of luminance and chrominance Intra or Intra+Q coded blocks in % of the total number of blocks (bounding box).

inter_blocks -- Flag enabling transmission of the number of luminance and chrominance Inter and Inter+Q coded blocks in % of the total number of blocks (bounding box).

inter4v_blocks -- Flag enabling transmission of the number of luminance and chrominance **Inter4V** coded blocks in % of the total number of blocks (bounding box).

not_coded_blocks -- Flag enabling transmission of the number of luminance and chrominance Non Coded blocks in % of the total number of blocks (bounding box).

texture_complexity_estimation_set_2_disable -- Flag to disable texture parameter set 2.

dct_coefs -- Flag enabling transmission of the number of DCT coefficients % of the maximum number of coefficients (coded blocks).

dct_lines -- Flag enabling transmission of the number of DCT8x1 in % of the maximum number of DCT8x1 (coded blocks).

vlc_symbols -- Flag enabling transmission of the average number of VLC symbols for macroblock.

vlc_bits -- Flag enabling transmission of the average number of bits for each symbol.

motion_compensation_complexity_disable -- Flag to disable motion compensation parameter set.

apm (Advanced Prediction Mode) **--** Flag enabling transmission of the number of luminance block predicted using APM in % of the total number of blocks for VOP (bounding box).

npm (Normal Prediction Mode) **--** Flag enabling transmission of the number of luminance and chrominance blocks predicted using NPM in % of the total number of luminance and chrominance for VOP (bounding box).

interpolate_mc_q -- Flag enabling transmission of the number of luminance and chrominance interpolated blocks in % of the total number of blocks for VOP (bounding box).

forw_back_mc_q -- Flag enabling transmission of the number of luminance and chrominance predicted blocks in % of the total number of blocks for VOP (bounding box).

halfpel2 -- Flag enabling transmission of the number of luminance and chrominance block predicted by a half-pel vector on one dimension (horizontal or vertical) in % of the total number of blocks (bounding box).

halfpel4 -- Flag enabling transmission of the number of luminance and chrominance block predicted by a half-pel vector on two dimensions (horizontal and vertical) in % of the total number of blocks (bounding box).

opaque -- Flag enabling transmission of the number of luminance and chrominance blocks coded using opaque coding mode in % of the total number of blocks (bounding box).

transparent -- Flag enabling transmission of the number of luminance and chrominance blocks coded using transparent mode in % of the total number of blocks (bounding box).

intra_cae -- Flag enabling transmission of the number of luminance and chrominance blocks coded using IntraCAE coding mode in % of the total number of blocks (bounding box).

inter_cae -- Flag enabling transmission of the number of luminance and chrominance blocks coded using InterCAE coding mode in % of the total number of blocks (bounding box).

no_update -- Flag enabling transmission of the number of luminance and chrominance blocks coded using no update coding mode in % of the total number of blocks (bounding box).

upsampling -- Flag enabling transmission of the number of luminance and chrominance blocks which need upsampling from 4-4- to 8-8 block dimensions in % of the total number of blocks (bounding box).

data_partitioned -- This is a one-bit flag which when set to '1' indicates that the macroblock data is rearranged differently, specifically, motion vector data is separated from the texture data (i.e., DCT coefficients).

reversible_vlc -- This is a one-bit flag which when set to '1' indicates that the reversible variable length tables (Table 11-22, Table 11-23 and Table 11-24) should be used when decoding DCT coefficients. These tables can only be used when data_partition flag is enabled.

scalability -- This is a one-bit flag which when set to '1' indicates that the current layer uses scalable coding. If the current layer is used as base-layer then this flag is set to '0'.

ref_layer_id -- This is a 4-bit unsigned integer with value between 0 and 15. It indicates the layer to be used as reference for prediction(s) in the case of scalability.

ref_layer_sampling_direc -- This is a one-bit flag which when set to ‘1’ indicates that the resolution of the reference layer (specified by reference_layer_id) is higher than the resolution of the layer being coded. If it is set to ‘0’ then the reference layer has the same or lower resolution then the resolution of the layer being coded.

hor_sampling_factor_n -- This is a 5-bit unsigned integer which forms the numerator of the ratio used in horizontal spatial resampling in scalability. The value of zero is forbidden.

hor_sampling_factor_m -- This is a 5-bit unsigned integer which forms the denominator of the ratio used in horizontal spatial resampling in scalability. The value of zero is forbidden.

vert_sampling_factor_n -- This is a 5-bit unsigned integer which forms the numerator of the ratio used in vertical spatial resampling in scalability. The value of zero is forbidden.

vert_sampling_factor_m -- This is a 5-bit unsigned integer which forms the denominator of the ratio used in vertical spatial resampling in scalability. The value of zero is forbidden.

enhancement_type -- This is a 1-bit flag which is set to ‘1’ when the current layer enhances the partial region of the reference layer. If it is set to ‘0’ then the current layer enhances the entire region of the reference layer. The default value of this flag is ‘0’.

random_accessible_vol -- This flag may be set to “1” to indicate that every VOP in this VOL is individually decodable. If all of the VOPs in this VOL are intra-coded VOPs and some more conditions are satisfied then random_accessible_vol may be set to “1”. random_accessible_vol may be omitted from the bitstream (by setting random_access_flag to “0”) in which case it shall be assumed to have the value zero. The flag random_accessible_vol is not used by the decoding process. random_accessible_vol is intended to aid random access or editing capability. This shall be set to “0” if any of the VOPs in the VOL are non-intra coded or certain other conditions are not fulfilled.

not_8_bit -- This one bit flag is set when the video data precision is not 8 bits per pixel.

quant_precision -- This field specifies the number of bits used to represent quantiser parameters. Values between 3 and 9 are allowed. When not_8_bit is zero, and therefore quant_precision is not transmitted, it takes a default value of 5.

bits_per_pixel -- This field specifies the video data precision in bits per pixel. It may take different values for different video object layers within a single video object. A value of 12 in this field would indicate 12 bits per pixel. This field may take values between 4 and 12. When not_8_bit is zero and bits_per_pixel is not present, the video data precision is always 8 bits per pixel, which is equivalent to specifying a value of 8 in this field.

6.3.5 Group of Video Object Plane

group_VOP_start_code -- The group_start_code is the unique code of length of 32bit. It identifies the beginning of a GOV header.

time_code -- This is a 18-bit integer containing the following: time_code_hours, time_code_minutes, marker_bit and time_code_seconds as shown in Table 6-9. The parameters correspond to those defined in the IEC standard publication 461 for “time and control codes for video tape recorders”. The time code refers to the first plane (in display order) after the GOV header.

Table 6-9 Meaning of time_code

time_code	**range of value**	**No. of bits**	**Mnemonic**
time_code_hours	0 - 23	5	uimsbf
time_code_minutes	0 - 59	6	uimsbf
marker_bit	1	1	bslbf
time_code_seconds	0 - 59	6	uimsbf

closed_gov -- This is a one-bit flag which indicates the nature of the predictions used in the first consecutive B-VOPs (if any) immediately following the first coded I-VOP after the GOV header .The closed_gov is set to '1' to indicate that these B-VOPs have been encoded using only backward prediction or intra coding. This bit is provided for use during any editing which occurs after encoding. If the previous pictures have been removed by editing, broken_link may be set to '1' so that a decoder may avoid displaying these B-VOPs following the first I-VOP following the group of plane header. However if the closed_gov bit is set to '1', then the editor may choose not to set the broken_link bit as these B-VOPs can be correctly decoded.

broken_link -- This is a one-bit flag which shall be set to '0' during encoding. It is set to '1' to indicate that the first consecutive B-VOPs (if any) immediately following the first coded I-VOP following the group of plane header may not be correctly decoded because the reference frame which is used for prediction is not available (because of the action of editing). A decoder may use this flag to avoid displaying frames that cannot be correctly decoded.

6.3.6 Video Object Plane and Video Plane with Short Header

VOP_start_code -- This is the bit string '000001B6' in hexadecimal. It marks the start of a video object plane.

VOP_coding_type -- The VOP_coding_type identifies whether a VOP is an intra-coded VOP (I), predictive-coded VOP (P), bidirectionally predictive-coded VOP (B) or sprite coded VOP (S). The meaning of VOP_coding_type is defined in Table 6-10.

Table 6-10 Meaning of VOP_coding_type

VOP_coding_type	**coding method**
00	intra-coded (I)
01	predictive-coded (P)
10	bidirectionally-predictive-coded (B)
11	sprite (S)

modulo_time_base -- This velue represents the local time base in one second resolution units (1000 milliseconds). It consists of a number of consecutive '1' followed by a '0'. Each '1' represents a duration of one second that have elapsed. For I- and P-VOPs, the number of '1's indicate the number of seconds elapsed since the synchronization point marked by the modulo_time_base of the previously decoded I- or P-VOP in decoding order. For B-VOP, the number of '1's indicate the number of seconds elapsed since the synchronization point marked by the modulo_time_base of the previously decoded I- or P-VOP in display order.

VOP_time_increment – This value represents the absolute VOP_time_increment from the synchronization point marked by the modulo_time_base measured in the number of clock ticks. It can take a value in the range of [0,VOP_time_increment_resolution). The number of bits representing the value is calculated as the minimum number of unsigned integer bits required to represent the above range. The local time base in the units of seconds is recovered by dividing this value by the VOP_time_increment_resolution.

VOP_coded -- This is a 1-bit flag which when set to '0' indicates that no subsequent data exists for the VOP. In this case, the following decoding rule applies: For an arbitrarily shaped VO (i.e. when the shape type of the VO is either 'binary' or 'binary only'), the alpha plane of the reconstructed VOP shall be completely transparent. For a rectangular VO (i.e. when the shape type of the VO is 'rectangular'), the corresponding rectangular alpha plane of the VOP, having the same size as its luminance component, shall be completely transparent. If there is no alpha plane being used in the decoding and composition process of a rectangular VO, the reconstructed VOP is filled with the respective content of the immediately preceeding VOP for which VOP_coded!=0.

VOP_rounding_type -- This is a one-bit flag which signals the value of the parameter rounding_control used for pixel value interpolation in motion compensation for P-VOPs. When this flag is set to '0', the value of rounding_control is 0, and when this flag is set to '1', the value of rounding_control is 1. When VOP_rounding_type is not present in the VOP header, the value of rounding_control is 0.

The encoder should control VOP_rounding_type so that each P-VOP have a different value for this flag from its reference VOP for motion compensation. VOP_rounding_type can have an arbitrary value if the reference picture is an I-VOP.

sprite_transmit_mode – This is a 2-bit code which signals the transmission mode of the sprite object. At video object layer initialization, the code is set to "piece" mode. When all object and quality update pieces are sent for the entire video object layer, the code is set to the "stop"mode. When an object piece is sent, the code is set to "piece" mode. When an update piece is being sent, the code is set to the "update" mode. When all sprite object pieces andquality update pieces for the current VOP are sent, the code is set to "pause" mode. Table 6-11 shows the different sprite transmit modes.

Table 6-11 Meaning of sprite transmit modes

code	**sprite_transmit_mode**
00	stop
01	piece
10	update
11	pause

VOP_width -- This is a 13-bit unsigned integer which specifies the horizontal size, in pixel units, of the rectangle that includes the VOP. The width of the encoded luminance component of VOP in macroblocks is (VOP_width+15)/16. The rectangle part is left-aligned in the encoded VOP. A zero value is forbidden.

VOP_height -- This is a 13-bit unsigned integer which specifies the vertical size, in pixel units, of the rectangle that includes the VOP. The height of the encoded luminance component of VOP in macroblocks is (VOP_height+15)/16. The rectangle part is top-aligned in the encoded VOP. A zero value is forbidden.

VOP_horizontal_mc_spatial_ref -- This is a 13-bit signed integer which specifies, in pixel units, the horizontal position of the top left of the rectangle defined by horizontal size of VOP_width. This is used for decoding and for picture composition.

marker_bit -- This is one-bit that shall be set to 1. This bit prevents emulation of start codes.

VOP_shape_coding_type – This is a 1 bit flag which specifies whether inter shape decoding is to be carried out for the current P VOP. If **VOP_shape_coding_type** is equal to '0', intra shape decoding is carried out, otherwise inter shape decoding is carried out.

VOP_vertical_mc_spatial_ref -- This is a 13-bit signed integer which specifies, in pixel units, the vertical position of the top left of the rectangle defined by vertical size of VOP_width. This is used for decoding and for picture composition.

background_composition -- This flag only occurs when scalability flag has a value of “1. This_flag is used in conjunction with enhancement_type flag. If enhancement_type is “1” and this flag is “1”, background composition specified in section 8.1 is performed. If enhancement type is “1” and this flag is “0”, any method can be used to make a background for the enhancement layer.

change_conv_ratio_disable – This is a 1-bit flag which when set to ‘1’ indicates that conv_ratio is not sent at the macroblock layer and is assumed to be 1 for all the macroblocks of the VOP. When set to ‘0’, the conv_ratio is coded at macroblock layer.

intra_dc_vlc_thr -- This is a 3-bit code allows a mechanism to switch between two VLC’s for coding of Intra DC coefficients as per Table 6-12.

Table 6-12 Meaning of intra_dc_vlc_thr

index	**meaning of intra_dc_vlc_thr**	**code**
0	Use Intra DC VLC for entire VOP	000
1	Switch to Intra AC VLC at running Qp >=13	001
2	Switch to Intra AC VLC at running Qp >=15	010
3	Switch to Intra AC VLC at running Qp >=17	011
4	Switch to Intra AC VLC at running Qp >=19	100
5	Switch to Intra AC VLC at running Qp >=21	101
6	Switch to Intra AC VLC at running Qp >=23	110
7	Use Intra AC VLC for entire VOP	111

Where running Qp is defined as Qp value used for immediately previous coded macroblock.

interlaced -- This is a 1-bit flag which being set to “1” indicates that the VOP may contain interlaced video. When this flag is set to “0”, the VOP is of non-interlaced (or progressive) format.

top_field_first -- This is a 1-bit flag which when set to “1” indicates that the top field (i.e., the field containing the top line) of reconstructed VOP is the first field to be displayed (output by the decoding process). When top_field_first is set to “0” it indicates that the bottom field of the reconstructed VOP is the first field to be displayed.

alternate_vertical_scan_flag -- This is a 1-bit flag which when set to “1” indicates the use of alternate vertical scan for interlaced VOPs.

VOP_quant -- This is an unsigned integer which specifies the absolute value of quant to be used for dequantizing the next macroblock. The length of this field is specified by the value of the parameter quant_precision. The default length is 5-bits which carries the binary representation of quantizer values from 1 to 31 in steps of 1.

VOP_alpha_quant – This is a an unsigned integer which specifies the absolute value of the initial alpha plane quantiser to be used for dequantising macroblock grayscale alpha data. The alpha plane quantiser cannot be less than 1.

VOP_constant_alpha – This bit is used to indicate the presence of VOP_constant_alpha_value. When this is set to one, VOP_constant_alpha_value is included in the bitstream.

VOP_constant_alpha_value – This is an unsigned integer which indicates the scale factor to be applied as a post processing phase of binary or grayscale shape decoding. See Section 7.4.4.2.

VOP_fcode_forward -- This is a 3-bit unsigned integer taking values from 1 to 7; the value of zero is forbidden. It is used in decoding of motion vectors.

VOP_fcode_backward -- This is a 3-bit unsigned integer taking values from 1 to 7; the value of zero is forbidden. It is used in decoding of motion vectors.

VOP_shape_coding_type -- This is a 1-bit flag which when set to '0' indicates the shape coding is INTRA. When inter_prediction_shape is set to '1' indicates the shape coding is INTER.

resync_marker -- This is a binary string of at least 16 zero's followed by a one'0 0000 0000 0000 0001'. For an I-VOP, the resync marker is 16 zeros followed by a one. The length of this resync marker is dependent on the value of VOP_fcode_forward, for a P-VOP, and the larger value of either VOP_fcode_forward and VOP_fcode_backward. The relationship between the length of the resync_marker and appropriate fcode is given by 16 + fcode. The resync_marker is (15+fcode) zeros followed by a one. A resync marker shall only be located immediately before a macroblock and aligned with a byte

Coded data for the top-left macroblock of the bounding box of a VOP shall immediately follow the VOP header, followed by the remaining macroblocks in the bounding box in the conventional left-to-right, top-to-bottom scan order. Video packets shall also be transmitted following the conventional left-to-right, top-to-bottom macroblock scan order. The last MB of one video packet is guaranteed to immediately precede the first MB of the following video packet in the MB scan order.

macroblock_number -- This is a variable length code with length between 1 and 14 bits and is only present when error_resilient_disable flag is set to '0'. It identifies the macroblock number within a VOP. The number of the top-left macroblock in a VOP shall be zero. The macroblock number increases from left to right and from top to bottom. The actual length of the code depends on the total number of macroblocks in the VOP calculated according to Table 6-13, the code itself is simply a binary representation of the macroblock number.

Table 6-13 Length of macroblock_number code

length of macroblock_number code	**((VOP_width+15)/16) * ((VOP_height+15)/16)**
1	1-2
2	3-4
3	5-8
4	9-16
5	17-32
6	33-64
7	65-128
8	129-256
9	257-512
10	513-1024
11	1025-2048
12	2049-4096
13	4097-8192
14	8193-16384
15	16385-32768
16	32769-65536
17	65537-131072
18	131073-262144

quant_scale – This is an unsigned integer which specifies the absolute value of quant to be used for dequantizing the next macroblock. The length of this field is specified by the value of the parameter quant_precision. The default length is 5-bits.

motion_marker -- This is a 17-bit binary string '1 1111 0000 0000 0001'. It is only present when the data_partitioning flag is set to '1'. It is used in conjunction with the resync_marker fields, macroblock_number, quant_scale and header_extension_code, a motion_marker is inserted after the motion data (prior to the texture data). The motion_marker is unique from the motion data and enables the decoder to determine when all the motion information has been received correctly.

dc_marker -- This is a 19 bit binary string '110 1011 0000 0000 0001'. It is present when the data_partitioning flag is set to '1'. It is used for I-VOPs only, in conjunction with the resync_marker field, macroblock_number, quant_scale and header_extension_code. A dc_marker is inserted into the bitstream after the mcbpc, dquant and dc data but before the ac_pred flag and remaining texture information.

header_extension_code -- This is a 1-bit flag which when set to '1' indicates the prescence of additional fields in the header. When header_extension_code is is se to '1', modulo_time_base, VOP_time_increment and VOP_coding_type are also included in the video packet header. Furthermore, if the VOP_coding_type is equal to either a P or B VOP, the appropriate fcodes are also present.

load_backward_shape -- This is a one-bit flag which when set to ‘1’ implies that the backward shape of the previous VOP in the same layer is copied to the forward shape for the current VOP and the backward shape of the current VOP is decoded from the bitstream. When this flag is set to ‘0’, the forward shape of the previous VOP is copied to the forward_shape of the current VOP and the backward shape of the previous VOP in the same layer is copied to the backward shape of the current VOP. This flag shall be ‘1’ when (1) background_composition is ‘1’ and VOP_coded of the previous VOP in the same layer is ‘0’ or (2) background_composition is ‘1’ and the current VOP is the first VOP in the current layer.

backward_shape_width -- This is a 13-bit unsigned integer which specifies the horizontal size, in pixel units, of the rectangle that includes the backward shape. A zero value is forbidden.

backward_shape_height -- This is a 13-bit unsigned integer which specifies the vertical size, in pixel units, of the rectangle that includes the backward shape. A zero value is forbidden.

backward_shape_horizontal_mc_spatial_ref -- This is a 13-bit signed integer which specifies, in pixel units, the horizontal position of the top left of the rectangle that includes the backward shape. This is used for decoding and for picture composition.

marker_bit -- This is one-bit that shall be set to 1. This bit prevents emulation of start codes.

backward_shape_vertical_mc_spatial_ref -- This is a 13-bit signed integer which specifies, in pixel units, the vertical position of the top left of the rectangle that includes the backward shape. This is used for decoding and for picture composition.

backward_shape() **--** The decoding process of the backward shape is identical to the decoding process for the shape of I-VOP with binary only mode (video_object_layer_shape = “10”).

load_forward_shape -- This is a one-bit flag which when set to ‘1’ implies that the forward shape is decoded from the bitstream. This flag shall be ‘1’ when (1) background_composition is ‘1’ and VOP_coded of the previous VOP in the same layer is ‘0’ or (2) background_composition is ‘1’ and the current VOP is the first VOP in the current layer.

forward_shape_width -- This is a 13-bit unsigned integer which specifies the horizontal size, in pixel units, of the rectangle that includes the forward shape. A zero value is forbidden.

forward_shape_height -- This is a 13-bit unsigned integer which specifies the vertical size, in pixel units, of the rectangle that includes the forward shape. A zero value is forbidden.

forward_shape_horizontal_mc_spatial_ref -- This is a 13-bit signed integer which specifies, in pixel units, the horizontal position of the top left of the rectangle that includes the forward shape. This is used for decoding and for picture composition.

marker_bit -- This is one-bit that shall be set to 1. This bit prevents emulation of start codes.

forward_shape_vertical_mc_spatial_ref -- This is a 13-bit signed integer which specifies, in pixel units, the vertical position of the top left of the rectangle that includes the forward shape. This is used for decoding and for picture composition.

forward_shape() **--** The decoding process of the backward shape is identical to the decoding process for the shape of I-VOP with binary only mode (video_object_layer_shape = “10”).

ref_select_code -- This is a 2-bit unsigned integer which specifies prediction reference choices for P- and B-VOPs in enhancement layer with respect to decoded reference layer identified by ref_layer_id. The meaning of allowed values is specified in Table 7-14 and Table 7-15.

6.3.6.1 Definition of DCECS variable values

The semantic of all complexity estimation parameters is defined at the VO syntax level. DCECS variables represent % values. The actual % values have been converted to 8 bit words by normalization to 256. To each 8 bit word a binary 1 is added to prevent start code emulation (i.e 0% = ‘00000001’, 99.5% = ‘11111111’ and is conventionally considered equal to). The binary ‘00000000’ string is a forbidden value. The only parameter expressed in their absolute value is the dcecs_vlc_bits parameter expressed as a 4 bit word.

dcecs_intra_blocks -- 8 bit number representing the % of luminance and chrominance Intra or Intra+Q coded blocks on the total number of blocks (bounding box).

dcecs_inter_blocks -- 8 bit number representing the % of luminance and chrominance Inter and Inter+Q coded blocks on the total number of blocks (bounding box).

dcecs_inter4v_blocks -- 8 bit number representing the % of luminance and chrominance Inter4V coded blocks on the total number of blocks (bounding box).

dcecs_not_coded_blocks -- 8 bit number representing the % of luminance and chrominance Non Coded blocks on the total number of blocks (bounding box).

dcecs_dct_coefs **--** 8 bit number representing the % of the number of DCT coefficients on the maximum number of coefficients (coded blocks).

dcecs_dct_lines -- 8 bit number representing the % of the number of DCT8x1 on the maximum number of DCT8x1 (coded blocks).

dcecs_vlc_symbols -- 8 bit number representing the average number of VLC symbols for macroblock.

dcecs_vlc_bits -- 4 bit number representing the average number of bits for each symbol.

dcecs_apm (Advanced Prediction Mode) **--** 8 bit number representing the % of the number of luminance block predicted using APM on the total number of blocks for VOP (bounding box).

dcecs_npm (Normal Prediction Mode) **--** 8 bit number representing the % of luminance and chrominance blocks predicted using NPM on the total number of luminance and chrominance blocks for VOP (bounding box).

dcecs_interpolate_mc_q -- 8 bit number representing the % of luminance and chrominance interpolated blocks in % of the total number of blocks for VOP (bounding box).

dcecs_forw_back_mc_q -- 8 bit number representing the % of luminance and chrominance predicted blocks on the total number of blocks for VOP (bounding box).

dcecs_halfpel2 -- 8 bit number representing the % of luminance and chrominance blocks predicted by a half-pel vector on one dimension (horizontal or vertical) on the total number of blocks (bounding box).

dcecs_halfpel4 -- 8 bit number representing the % of luminance and chrominance blocks predicted by a half-pel vector on two dimensions (horizontal and vertical) on the total number of blocks (bounding box).

dcecs_opaque -- 8 bit number representing the % of luminance and chrominance blocks using opaque coding mode on the total number of blocks (bounding box).

dcecs_transparent -- 8 bit number representing the % of luminance and chrominance blocks using transparent coding mode on the total number of blocks (bounding box).

dcecs_intra_cae -- 8 bit number representing the % of luminance and chrominance blocks using IntraCAE coding mode on the total number of blocks (bounding box).

dcecs_inter_cae -- 8 bit number representing the % of luminance and chrominance blocks using InterCAE coding mode on the total number of blocks (bounding box).

dcecs_no_update -- 8 bit number representing the % of luminance and chrominance blocks using no update coding mode on the total number of blocks (bounding box).

dcecs_upsampling -- 8 bit number representing the % of luminance and chrominance blocks which need upsampling from 4-4- to 8-8 block dimensions on the total number of blocks (bounding box).

6.3.6.2 Video Plane with Short Header

video_plane_with_short_header() – This data structure contains a video plane using an abbreviated header format. Certain values of parameters shall have pre-defined and fixed values for any video_plane_with_short_header, due to the limited capability of signaling information in the short header format. These parameters having fixed values are shown in Table 6-14.

Table 6-14-Fixed Settings for video_plane_with_short_header()

Parameter	Value
video_object_layer_shape	“rectangular”
obmc_disable	1
quant_type	0
error_resilient_disable	1
data_partitioned	0
block_count	6
reversible_vlc	0
vop_rounding_type	0
vop_fcode_forward	1
vop_coded	1
interlaced	0
complexity_estimation_disable	1
use_intra_dc_vlc	0
scalability	0
not_8_bit	0
bits_per_pixel	8

temporal_reference – This is an 8-bit number which can have 256 possible values. It is formed by incrementing its value in the previously transmitted video_plane_with_short_header() by one plus the number of non-transmitted pictures (at 30000/1001 Hz) since the previously transmitted picture. The arithmetic is performed with only the eight LSBs.

zero_bit – This is a single bit having the value zero (0).

split_screen_indicator – This is a boolean signal that indicates that the upper and lower half of the decoded picture could be displayed side by side. This bit has no direct effect on the encoding or decoding of the video plane.

document_camera_indicator – This is a boolean signal that indicates that the video content of the vop is sourced as a representation from a document camera or graphic representation, as opposed to a view of natural video content. This bit has no direct effect on the encoding or decoding of the video plane.

full_picture_freeze_release – This is a boolean signal that indicates that resumption of display updates should be activated if the display of the video content has been frozen due to errors, packet losses, or for some other reason such as the receipt of a external signal. This bit has no direct effect on the encoding or decoding of the video plane.

source_format – This is an indication of the width and height of the rectangular video plane represented by the video_plane_with_short_header. The meaning of this field is shown in Table 6-15. Each of these source formats has the same vop time increment resolution which is equal to 30000/1001 (approximately 29.97) Hz and the same width:height pixel aspect ratio (288/3):(352/4), which equals 12:11 in relatively prime numbers and which defines a CIF picture as having a width:height picture aspect ratio of 4:3.

Table 6-15 Parameters Defined by source_format Field

source_format value	Source Format Meaning	vop_width	vop_height	num_macroblocks_in_gob	num_gobs_in_vop
000	reserved	reserved	reserved	reserved	reserved
001	sub-QCIF	128	96	8	6
010	QCIF	176	144	11	9
011	CIF	352	288	22	18
100	4CIF	704	576	88	18
101	16CIF	1408	1152	352	18
110	reserved	reserved	reserved	reserved	reserved
111	reserved	reserved	reserved	reserved	reserved

picture_coding_type – This bit indicates the vop_coding_type. When equal to zero, the vop_coding_type is "I", and when equal to one, the vop_coding_type is "P".

four_reserved_zero_bits – This is a four-bit field containing bits which are reserved for future use and equal to zero.

pei – This is a single bit which, when equal to one, indicates the presence of a byte of psupp data following the pei bit.

psupp — This is an eight bit field which is present when pei is equal to one. The pei + psupp mechanism provides for a reserved method of later allowing the definition of backward-compatible data to be added to the bitstream. Decoders shall accept and discard psupp when pei is equal to one, with no effect on the decoding of the video data. The pei and psupp combination pair may be repeated if present. The ability for an encoder to add pei and psupp to the bitstream is reserved for future use.

gob_number – This is a five-bit number which indicates the location of video data within the video plane. A group of blocks (or GOB) contains a number of macroblocks in raster scanning order within the picture. For a given gob_number, the GOB contains the num_macroblocks_per_gob macroblocks starting with macroblock_number = gob_number * num_macroblocks_per_gob. The gob_number can either be read from the bitstream or inferred from the progress of macroblock decoding as shown in the syntax description pseudo-code.

num_macroblocks_in_gob – This is the number of macroblocks in each group of blocks (GOB) unit. This parameter is derived from the source_format as shown in Table 6-15.

num_gobs_in_vop – This is the number of GOBs in the vop. This parameter is derived from the source_format as shown in Table 6-15-1.

gob_layer() – This is a layer containing a fixed number of macroblocks in the vop. Which macroblocks which belong to each gob can be determined by gob_number and num_macroblocks_in_gob.

gob_resync_marker – This is a fixed length code of 17 bits having the value '0000 0000 0000 0000 1' which may optionally be inserted at the beginning of each gob_layer(). Its purpose is to serve as a type of resynchronization marker for error recovery in the bitstream. The gob_resync_marker codes may (and should) be byte aligned by inserting zero to seven zero-valued bits in the bitstream just prior to the gob_resync_marker in order to obtain byte alignment. The gob_resync_marker shall not be present for the first GOB (for which gob_number = 0).

gob_number – This is a five-bit number which indicates which GOB is being processed in the vop. Its value may either be read following a gob_resync_marker or may be inferred from the progress of macroblock decoding. All GOBs shall appear in the bitstream of each video_plane_with_short_header(), and the GOBs shall appear in a strictly increasing order in the bitstream. In other words, if a gob_number is read from the bitstream after a gob_resync_marker, its value must be the same as the value that would have been inferred in the absence of the gob_resync_marker.

gob_frame_id – This is a two bit field which is intended to help determine whether the data following a gob_resync_marker can be used in cases for which the vop header of the video_plane_with_short_header() may have been lost. gob_frame_id shall have the same value in every GOB header of a given video_plane_with_short_header(). Moreover, if any field among the split_screen_indicator or document_camera_indicator or full_picture_freeze_release or source_format or picture_coding_type as indicated in the header of a video_plane_with_short_header() is the same as for the previous transmitted picture in the same video object, gob_frame_id shall have the same value as in that previous video_plane_with_short_header(). However, if any of these fields in the header of a certain video_plane_with_short_header() differs from that in the previous transmitted video_plane_with_short_header() of the same video object, the value for gob_frame_id in that picture shall differ from the value in the previous picture.

6.3.6.3 Shape coding

bab_type – This is a variable length code between 1 and 6 bits. It indicates the coding mode used for the bab. There are seven bab_types as depicted in Table 6-16 . The VLC tables used depend on the decoding context i.e. the bab_types of blocks already received. For I-VOPs, the context-switched VLC table of Table 11-26 is used. For P-VOPs and B-VOPs, the context switched table of Table 11-27 is used.

Table 6-16 List of bab_types and usage

bab_type	**Semantic**	**Used in**
0	MVDs==0 && No Update	P,B VOPs
1	MVDs!=0 && No Update	P,B VOPs
2	transparent	All VOP types
3	opaque	All VOP types
4	intraCAE	All VOP types

5	MVDs==0 && interCAE	P,B VOPs
6	MVDs!=0 && interCAE	P,B VOPs

The bab_type determines what other information fields will be present for the bab shape. No further shape information is present if the bab_type = 0, 2 or 3. opaque means that all pixels of the bab are part of the object. transparent means that none of the bab pixels belong to the object. IntraCAE means the intra-mode CAE decoding will be required to reconstruct the pixels of the bab. No_update means that motion compensation is used to copy the bab from the previous VOP's binary alpha map. InterCAE means the motion compensation and inter_mode CAE decoding are used to reconstruct the bab. MVDs refers to the motion vector difference for shape.

mvds_x – This is a VLC code between 1 and 18 bits. It represents the horizontal element of the motion vector difference for the bab. The motion vector difference is in full integer precision. The VLC table is shown is Table 11-28.

mvds_y -- This is a VLC code between 1 and 18 bits. It represents the vertical element of the motion vector difference for the bab. The motion vector difference is in full integer precision. If mvds_x is '1', then the VLC table of Table 11-29 , otherwise the VLC table of Table 11-29 is used.

conv_ratio – This is VLC code of length 1-2 bits. It specifies the factor used for sub-sampling the 16x16 pixel bab. The decoder must up-sample the decoded bab by this factor. The possible values for this factor are 1, 2 and 4 and the VLC table used is given in Table 11-30.

scan_type– This is a 1-bit flag where a value of '0' implies that the bab is in transposed form i.e. the BAB has been transposed prior to coding. The decoder must then transpose the bab back to its original form following decoding. If this flag is '1', then no transposition is performed.

binary_arithmetic_code() – This is a binary arithmetic decoder representing the pixel values of the bab. This code may be generated by intra cae or inter cae depending on the bab_type. Cae decoding relies on the knowledge of intra_prob[] and inter_prob[], probability tables given in Annex B.

6.3.6.4 Sprite coding

warping_mv_code(dmv) **--** The codeword for each differential motion vector consists of a VLC indicating the length of the dmv code (dmv_length) and a FLC, dmv_code-, with dmv_length bits. The codewords are listed in Table 11-32 Code table for the first trajectory point.

brightness_change_factor () **--** The codeword for brightness_change_factor consists of a variable length code denoting brightness_change_factor_size and a fix length code, brightness_change_factor, of brightness_change_factor_size bits (sign bit included). The codewords are listed in Table 11-33 The codewords are listed in Table 11-33

send_mb() **--** This function returns 1 if the current macroblock has already been sent previously and "not coded". Otherwise it returns 0.

piece_quant -- This is a 5-bit unsigned interger which indicates the quant to be used for a sprite-piece until updated by a subsequent dquant.   The piece_quant carries the binary representation of quantizer values from 1 to 31 in steps of 1.

piece_width -- This value specifies the width of the sprite piece measured in macroblock units.

piece_height -- This value specifies the height of the sprite piece measured in macroblock units.

piece_xoffset -- This value specifies the horizontal offset location, measured in macroblock units from the left edge of the sprite object, for the placement of the sprite piece into the sprite object buffer at the decoder.

piece_yoffset -- This value specifies the vertical offset location, measured in macroblock units from the top edge of the sprite object.

decode_sprite_piece () -- It decodes a selected region of the sprite object or its update. It also decodes the parameters required by the decoder to properly incorporate the pieces. All the static-sprite-object pieces will be encoded using a subset of the I-VOP syntax. And the static-sprite-update pieces use a subset of the P-VOP syntax. The sprite update is defined as the difference between the original sprite texture and the reconstructed sprite assembled from all the sprite object pieces.

sprite_shape_texture() -- For the static-sprite-object pieces, shape and texture are coded using the macroblock layer structure in I-VOPs. And the static-sprite-update pieces use the P-VOP inter-macroblock syntax -- except that there are no motion vectors and shape information included in this syntax structure. Macroblocks raster scanning is employed to encode a sprite piece; however, whenever the scan encounters a macroblock which has been part of some previously sent sprite piece, then the block is not coded and the corresponding macroblock layer is empty.

6.3.7 Macroblock related

not_coded -- This is a 1-bit flag which signals if a macroblock is coded or not. When set to'1' it indicates that a macroblock is not coded and no further data is included in the bitstream for this macroblock; decoder shall treat this macroblock as 'inter' with motion vector equal to zero and no DCT coefficient data. When set to '0' it indicates that the macroblock is coded and its data is included in the bitstream.

mcbpc -- This is a variable length code that is used to derive the macroblock type and the coded block pattern for chrominance . It is always included for coded macroblocks. Table 11-6 and Table 11-7 list all allowed codes for mcbpc in I- and P-VOPs respectively. The values of the column "MB type" in these tables are used as the variable "derived_mb_type" which is used in the respective syntax part for motion and texture decoding. In P-vops using the short video header format (i.e., when short_video_header is 1), mcbpc codes indicating macroblock type 2 shall not be used.

ac_pred_flag -- This is a 1-bit flag which when set to '1' indicates that either the first row or the first column of ac coefficients are differentially coded for intra coded macroblocks.

modb -- This is a variable length code present only in coded macroblocks of B-VOPs. It indicates whether mb_type and/or cbpb information is present for a macroblock. The codes for modb are listed in Table 11-3.

mb_type -- This variable length code is present only in coded macroblocks of B-VOPs. Further, it is present only in those macroblocks for which one motion vector is included. The codes for mb_type are shown in Table 11-4 for B-VOPs for no scalability and in Table 11-5 for B-VOPs with scalability. When mb_type is not present (i.e. modb=='0') for a macroblock in a B-VOP, the macroblock type is set to the default type. The default macroblock type for the enhancement layer of spatially scalable bitstreams (i.e. ref_select_code == '00' && scalability = '1') is "forward mc + Q". Otherwise, the default macroblock type is "direct".

cbpb -- This is a 3 to 6 bit code representing coded block pattern in B-VOPs, if indicated by modb. Each bit in the code represents a coded/no coded status of a block; the leftmost bit corresponds to the top left block in the macroblock. For each non-transparent blocks with coefficients, the corresponding bit in the code is set to '1'. When cbpb is not present (i.e. modb=='0' or '10') for a macroblock in a B-VOP, no coefficients are coded for all the non-transparent blocks in this macroblock.

cbpy -- This variable length code represents a pattern of non-transparent luminance blocks with at least one non intra DC transform coefficient, in a macroblock. Table 11-8 – 11-10 indicate the codes and the corresponding patterns they indicate for the respective cases of intra- and inter-MBs. If there is only one non transparent block in the macroblock, a single bit cbpy is used (1:coded, 0:not coded).

dquant -- This is a 2-bit code which specifies the change in the quantizer, quant, for I- and P-VOPs. Table 6-17 lists the codes and the differential values they represent. The value of quant lies in range of 1 to 31; if the value of quant after adding dquant value is less than 1 or exceeds 31, it shall be correspondingly clipped to 1 and 31.

Table 6-17 dquant codes and corresponding values

dquant code	value
00	-1
01	-2
10	1
11	2

dbquant -- This is a variable length code which specifies the change in quantizer for B-VOPs. Table 6-18 lists the codes and the differential values they represent. If the value of quant after adding dbquant value is less than 1 or exceeds 31, it shall be correspondingly clipped to 1 and 31.

Table 6-18 dbquant codes and corresponding values

dbquant code	value
10	-2
0	0
11	2

CODA_I – This is a one-bit flag which is set to "1" to indicate that all the values in the grayscale alpha macroblock are equal to 255 (AlphaOpaqueValue). When set to "0", this flag indicates that one or more 8x8 blocks are coded according to CBPA.

ac_pred_flag_alpha – This is a one-bit flag which when set to '1' indicates that either the first row or the first column of ac coefficients are to be differentially decoded for intra alpha macroblocks. It has the same effect for alpha as the corresponding luminance flag.

CODA_PB – This is a VLC indicating the coding status for P or B alpha macroblocks. The semantics are given in the table below (Table 6-19). When this VLC indicates that the alpha macroblock is all opaque, this means that all values are set to 255 (AlphaOpaqueValue).

Table 6-19: CODA_PB codes and corresponding values

CODA_PB	Meaning
1	alpha residue all zero
01	alpha macroblock all opaque
00	alpha residue coded

CBPA – This is the coded block pattern for grayscale alpha texture data. For I, P and B VOPs, this VLC is exactly the same as the INTER (P) CBPY VLC described by Tables 11-8 thru 11-10. CBPA is followed by the alpha block data which is coded in the same way as texture block data. Note that grayscale alpha blocks with alpha all equal to zero (transparent) are not included in the bitstream.

6.3.7.1 MB Binary Shape Coding

babtype – This defines the coding type of the current bab according to Table 11-26 and Table 11-27 for intra and inter mode, respectively.

mvds_x –This defines the size of the x-component of the differential motion vector for the current bab according to Table 11-28.

mvds_y -- This defines the size of the y-component of the differential motion vector for the current bab according to Table 11-28 if mvds_x!=0 and according to Table 11-29 if mvds_x==0.

conv_ratio –This defines the upsampling factor according to Table 11-30 to be applied after decoding the current shape information

scan_type –This defines according to Table 6-20 whether the current bordered to be decoded bab and the eventual bordered motion compensated bab need to be transposed

Table 6-20 scan_type

scan_type	meaning
0	transpose bab as in matrix transpose
1	do not transpose

binary_arithmetic_code() –This is a binary arithmetic decoder that defines the context dependent arithmetically to be decoded binary shape information. The meaning of the bits is defined by the arithmetic decoder according to Section 7.4.3

6.3.7.2 Motion vector

horizontal_mv_data — This is a variable length code, as defined in Table 11-9, which is used in motion vector decoding as described in section 7.5.3.

horizontal_mv_residual — This is an unsigned integer which is used in motion vector decoding as described in section 7.5.3. The number of bits in the bitstream for horizontal_mv_residual, r_size, is derived from either VOP_fcode_forward or VOP_fcode_backward as follows;

r_size = VOP_fcode_forward - 1 or r_size = VOP_fcode_backward - 1

vertical_mv_data — This is a variable length code, as defined in Table 11-9, which is used in motion vector decoding as described in section 7.5.3.

vertical_mv_residual — This is an unsigned integer which is used in motion vector decoding as described in section 7.5.3. The number of bits in the bitstream for vertical_mv_residual, r_size, is derived from either VOP_fcode_forward or VOP_fcode_backward as follows;

r_size = VOP_fcode_forward - 1 or r_size = VOP_fcode_backward - 1

6.3.7.3 Interlaced Information

dct_type – This is a 1-bit flag indicating whether the macroblock is frame DCT coded or field DCT coded. If this flag is set to “1”, the macroblock is field DCT coded; otherwise, the macroblock is frame DCT coded. This flag is only present in the bitstream if the interlaced flag is set to “1” and the macroblock is coded (coded blcok pattern is non-zero) or intra-coded. Boundary blocks are always coded in frame-based mode.

field_prediction – This is a 1-bit flag indicating whether the macroblock is field predicted or frame predicted. This flag is set to ‘1’ when the macroblock is predicted using field motion vectors. If it is set to ‘0’ then frame prediction (16x16 or 8x8) will be used. This flag is only present in the bitstream if the interlaced flag is set to “1” and the derived_mb_type is “0” or “1” in the P-VOP or an non-direct mode macroblock in the B-VOP.

forward_top_field_reference – This is a 1-bit flag which indicates the reference field for the forward motion compensation of the top field. When this flag is set to ‘0’, the top field is used as the reference field. If it is set to ‘1’ then the bottom field will be used as the reference field. This flag is only present in the bitstream if the field_prediction flag is set to “1” and the macroblock is not backward predicted.

forward_bottom_field_reference – This is a 1-bit flag which indicates the reference field for the forward motion compensation of the bottom field. When this flag is set to ‘0’, the top field is used as the reference field. If it is set to ‘1’ then the bottom field will be used as the reference field. This flag is only present in the bitstream if the field_prediction flag is set to “1” and the macroblock is not backward predicted.

backward_top_field_reference – This is a 1-bit flag which indicates the reference field for the backward motion compensation of the top field. When this flag is set to ‘0’, the top field is used as the reference field. If it is set to ‘1’ then the bottom field will be used as the reference field. This flag is only present in the bitstream if the field_prediction flag is set to “1” and the macroblock is not forward predicted.

backward_bottom_field_reference – This is a 1-bit flag which indicates the reference field for the backward motion compensation of the bottom field. When this flag is set to ‘0’, the top field is used as the reference field. If it is set to ‘1’ then the bottom field will be used as the reference field.. This flag is only present in the bitstream if the field_prediction flag is set to “1” and the macroblock is not forward predicted.

6.3.8 Block related

dct_dc_size_luminance -- This is a variable length code as defined in Table 11-12 that is used to derive the value of the differential dc coefficients of luminance values in blocks in intra macroblocks. This value categorizes the coefficients according to their size.

dct_dc_size_chrominance -- This is a variable length code as defined in Table 11-13 that is used to derive the value of the differential dc coefficients of chrominance values in blocks in intra macroblocks. This value categorizes the coefficients according to their size.

dct_dc_differential -- This is a variable length code as defined in Table 11-14 that is used to derive the value of the differential dc coefficients in blocks in intra macroblocks. After identifying the category of the dc coefficient in size from dct_dc_size_luminance or dct_dc_size_chrominance, this value denotes which actual difference in that category occurred.

6.3.8.1 Alpha block related

dct_dc_size_alpha – This is a variable length code for coding the alpha block dc coefficient. Its semantics are the same as dct_dc_size_luminance in Section 6.3.8.

6.3.9 Still texture object

still_texture_object_start_code -- The still_texture_object_start_code is a string of 32 bits. The first 24 bits are '0000 0000 0000 0000 0000 0001' and the last 8 bits are defined in Table 6-3.

texture_object_id -- This is given by 16-bits representing one of the values in the range of '0000 0000 0000 0000' to '1111 1111 1111 1111' in binary. The texture_object_layer_id uniquely identifies a texture object layer.

wavelet_filter_type -- This field indicates the arithmetic precision which is used for the wavelet decomposition as the following:

Table 6-21 Wavelet type

wavelet_filter_type	Meaning
0	integer
1	Double float

wavelet_download – This field indicates if the 2-band filter bank is specificed in the bitstream:

Table 6-22 Wavelet downloading flag

wavelet_download	meaning
0	default filters
1	specified in bitstream

The default filter banks are described in the 11.2.2.

max_bitplanes -- This field indicates the number of maximum bitplanes in bilevel_quant mode.

wavelet_decomposition_levels -- This field indicates the number of levels in the wavelet decomposition of the texture.

texture_spatial_layer_start_code -- The texture_spatial_layer_start_code is a string of 32 bits. The 32 bits are '0000 0000 0000 0000 0000 0001 1011 1111' in binary. The texture_spatial_layer_start_code marks the start of a new spatial layer.

texture_spatial_layer_id -- This is given by 5-bits representing one of the values in the range of '00000' to '11111' in binary. The texture_spatial_layer_id uniquely identifies a spatial layer.

texture_snr_layer_start_code -- The texture_snr_layer_start_code is a string of 32 bits. The 32 bits are '0000 0000 0000 0000 0000 0001 1100 0000' in binary. The texture_snr_layer_start_code marks the start of a new snr layer.

texture_snr_layer_id -- This is given by 5-bits representing one of the values in the range of '00000' to '11111' in binary. The texture_snr_layer_id uniquely identifies an SNR layer.

Note: All the start codes start at the byte boundary. Appropriate number of bits is stuffed before any start code to byte-align the bitstream.

texture_object_layer_shape -- This is a 2-bit integer defined in Table 6-23. It identifies the shape type of a texture object layer.

Table 6-23 Texture Object Layer Shape type

shape_format	**Meaning**
00	rectangular
01	binary
10	reserved
11	reserved

scan_direction -- This field indicates the scan order of AC coefficients. In single-quant and multi-quant mode, if this flag is `0', then the coefficients are scanned in the tree-depth fashion. If it is `1', then they are scanned in the subband by subband fashion. In bilevel_quant mode, if the flag is `0', then they are scanned in bitplane by bitplane fashion. Within each bitplane, they are scanned in a subband by subband fashion. If it is "1", they are scanned from the low wavelet decomposition layer to high wavelet decomposition layer. Within each wavelet decomposition layer, they are scanned from most significant bitplane down to the least significant bitplane.

wavelet_stuffing -- These 3 stuffing bits are reserved for future expansion. It is currently defined to be '111'.

spatial_layers -- This field indicates the number of spatial layers. It is equivalent to the maximum number of the wavelet decomposition layers in that scalability layer.

texture_object_layer_width -- The texture_object_layer_width is a 15-bit unsigned integer representing the width of the displayable part of the luminance component in pixel units. A zero value is forbidden.

texture_object_layer_height -- The texture_object_layer_width is a 15-bit unsigned integer representing the height of the displayable part of the luminance component in pixel units. A zero value is forbidden.

horizontal_ref -- This is a 15-bit integer which specifies, in pixel units, the horizontal position of the top left of the rectangle defined by horizontal size of object_width. This is used for decoding and for picture composition.

vertical_ref -- This is a 15-bit integer which specifies, in pixel units, the vertical position of the top left of the rectangle defined by vertical size of object_height. This is used for decoding and for picture composition.

object_width -- This is a 15-bit unsigned integer which specifies the horizontal size, in pixel units, of the rectangle that includes the object. A zero value is forbidden.

object_height -- This is a 15-bit unsigned integer which specifies the vertical size, in pixel units, of the rectangle that includes the object. A zero value is forbidden.

lowpass_filter_length – This field defines the length of the low pass filter in binary ranging from "0001" (length of 1) to "1111" (length of 15.)

highpass_filter_length – This field defines the length of the high pass filter in binary ranging from "0001" (length of 1) to "1111" (length of 15.)

filter_tap_integer – This field defines an integer filter coefficient in 16bit signed integer. The filter coefficients are decoded from the left most tap to the right most tap order.

filter_tap_float_high– This field defines the left 16 bits of a floating filter coefficient which is defined in 32-bit IEEE floating format. The filter coefficients are decoded from the left most tap to the right most tap order.

filter_tap_float_low– This field defines the right 16 bits of a floating filter coefficient which is defined in 32-bit IEEE floating format. The filter coefficients are decoded from the left most tap to the right most tap order.

integer_scale – This field defines the scaling factor of the integer wavelet, by which the output of each composition level is divided by // operation. A zero value is forbidden.

mean -- This field indicates the mean value of one color component of the texture.

quant_dc_byte -- This field indicates the quantization step size for one color component of the DC subband. A zero value is forbidden. The quantization step size parameter, quant_dc, is decoded using the function get_param( ): quant = get_param(7);

spatial_scalability_levels -- This field indicates the number of spatial scalability layers supported in the bitstream. This number can be from 1 to wavelet_decomposition_levels. For single_quant mode, the only valid value of spatial_scalability_levels is 1.

quantization_type -- This field indicates the type of quantization as shown in Table 6-24

Table 6-24 The quantization type

Quantization_type	**Code**
single quantizer	01
multi quantizer	10
bi-level quantizer	11

snr_start_code_enable -- If this flag is enabled (disable =0; enabled = 1), the start code followed by an id to be inserted in to each spatial scalability layer and/or each SNR scalability layer.

quant_byte -- This field defines one byte of the quantization step size for each scalability layer. A zero value is forbidden. The quantization step size parameter, quant, is decoded using the function get_param( ): quant = get_param(7);

snr_scalability_levels -- This field indicates the number of levels of SNR scalability supported in this spatial scalability level.

snr_all_zero -- This flag indicates whether all the coefficients in the SNR layer are zero or not. The value '0' for this flag indicates that the SNR layer contains some nonzero coefficients which are coded after this flag. The value '1' for this flag indicates that the current SNR layer only contains zero coefficients and therefore the layer is skipped.

band_offset_byte-- This field defines one byte of the absolute value of the parameter band_offset. This parameter is added to each DC band coefficient obtained by arithmetic decoding. The parameter band_offset is decoded using the function get_param( ):

```
band_offset = -get_param( 7 );
```

where function get_param() is defined as

```
           int get_param(int  nbit)
           {
          int count = 0;
          int word =0;
     int value = 0;
          int module = 1<<(nbit);

                     do{
               word= get_next_word_from_bitstream( nbit+1);
               value += (word  & (module-1) ) << (count * nbit);
               count ++;
          } while( word>> nbit);
               return value;
            }
```

The function get_next_word_from_bitstream(x) reads the next x bits from the input bitstream.

band_max_byte -- This field defines one byte of the maximum value of the DC band. The parameter band_max_value is decoded using function get_param( ):

```
band_max_value =  get_param( 7 );
```

root_max_alphabet_byte-- This field defines one byte of the maximum absolute value of the quantized coefficients of the three lowest AC bands. This parameter is decoded using the function get_param( ):

```
root_max_alphabet = get_param ( 7 );
```

valz_max_alphabet_byte-- This field defines one byte of the maximum absolute value of the quantized coefficients of the 3 highest AC bands. The parameter valz_max is decoded using the function get_param( ):

```
valz_max_alphabet = get_param ( 7 );
```

valnz_max_alphabet_byte-- This field defines one byte of the maximum absolute value of the quantized coefficients which belong to the middle AC bands (the bands between the 3 lowest and the 3 highest AC bands). The parameter valnz_max_alphabet is decoded using the function get_param( ):

```
valnz_max_alphabet = get_param ( 7 );
```

arith_decode_dc() – This is an arithmetic decoder for decoding the quantized coefficient values of DC band. This bitstream is generated by an adaptive arithmetic encoder. The arithmetic decoding relies on the initialization of a uniform probability distribution model described in 11.2.2. The decoding procedure is same as arith_decode_highbands(). The arith_decode_dc() function uses the same arithmetic decoder as described in arith_decode_highbands() but it uses different scanning, and a different probability model (*DC*).

arith_decode_highbands() -- This is an arithmetic decoder for decoding the quantized coefficient values of the higher bands (all bands except DC band). The bitstream is generated by an adaptive arithmetic encoder. The arithmetic decoding relies on the initialization of the uniform probability distribution models described in 11.2.2. This decoder uses only integer arithmetic. It also uses an adaptive probability model based on the frequency counts of the previously decoded symbols. The maximum range (or precision) specified is (2^16) - 1 (16 bits). The maximum frequency count is (2^14) - 1 (14 bits).

arith_decode_highbands_bilevel() -- This is an arithmetic decoder for decoding the quantized coefficient values of the higher bands in the bilevel_quant mode (all bands except DC band). The bitstream is generated by an adaptive arithmetic encoder. The arithmetic decoding relies on the initialization of the uniform probability distribution models described The decoding procedure is the same as arith_decode_highbands(). The arith_decode_highbands_bilevel()function uses the same arithmetic decoder as described in arith_decode_highbands(), but it uses bitplane scanning, and a different probability model as described in 11.2.2. In this mode, The maximum range (or precision) specified is (2^16) - 1 (16 bits). The maximum frequency count is 127.

6.3.9.1 **Shape Object decoding**

change_conv_ratio_disable –This specifies whether conv_ratio is encoded at the shape object decoding function. If it is set to “1” when disable.

STO_constant_alpha -- This is a 1-bit flag when set to ‘1’, the opaque alpha values of the binary mask are replaced with the alpha value specified by STO_constant_alpha_value.

STO_constant_alpha_value -- This is an 8-bit code that gives the alpha value to replace the opaque pixels in the binary alpha mask. Value ‘0’ is forbidden.

bab_type – This is a variable length code of 1-2 bits. It indicates the coding mode used for the bab. There are three bab_types as depicted in Table 6-16 . The VLC tables used depend on the decoding context i.e. the bab_types of blocks already received.

Table 6-25 List of bab_types and usage

bab_type	**Semantic**	**code**
2	transparent	10
3	opaque	0
4	intraCAE	11

The bab_type determines what other information fields will be present for the bab shape. No further shape information is present if the bab_type = 2 or 3. opaque means that all pixels of the bab are part of the object. transparent means that none of the bab pixels belong to the object. IntraCAE means the intra-mode CAE decoding will be required to reconstruct the pixels of the bab.

conv_ratio – This is VLC code of length 1-2 bits. It specifies the factor used for sub-sampling the 16x16 pixel bab. The decoder must up-sample the decoded bab by this factor. The possible values for this factor are 1, 2 and 4 and the VLC table used is given in Table 11-30.

scan_type– This is a 1-bit flag where a value of ‘0’ implies that the bab is in transposed form i.e. the bab has been transposed prior to coding. The decoder must then transpose the bab back to its original form following decoding. If this flag is ‘1’, then no transposition is performed.

binary_arithmetic_decode() – This is a binary arithmetic decoder representing the pixel values of the bab. Cae decoding relies on the knowledge of intra_prob[], probability tables given in Annex B.

6.3.10 Mesh related

mesh_object_start_code – The mesh_object_start_code is the bit string ‘000001BC’ in hexadecimal. It initiates a mesh object.

6.3.10.1 Mesh object plane

mesh_object_plane_start_code – The mesh_object_plane_start_code is the bit string ‘000001BD’ in hexadecimal. It initiates a mesh object plane.

new_mesh_flag -- This is a 1-bit flag which when set to ‘1’ indicates that a new mesh is following in the bitstream. When set to ‘0’ it indicates that the current mesh is coded with respect to the previous mesh by using node motion vectors

6.3.10.2 Mesh geometry

mesh_type_code -- This is a 2-bit integer defined in Table 6-26. It indicates the type of initial mesh geometry being encoded.

Table 6-26 Mesh type code

mesh type code	mesh geometry
00	uniform
01	Delaunay
10	reserved
11	reserved

nr_of_mesh_nodes_hor -- This is a 10-bit unsigned integer specifying the number of nodes in one row of a uniform mesh.

nr_of_mesh_nodes_vert -- This is a 10-bit unsigned integer specifying the number of nodes in one column of a uniform mesh.

mesh_rect_size_hor -- This is a 8-bit unsigned integer specifying the width of a rectangle of a uniform mesh (containing two triangles) in half pixel units.

mesh_rect_size_vert -- This is a 8-bit unsigned integer specifying the height of a rectangle of a uniform mesh (containing two triangles) in half pixel units.

triangle_split_code - This is a 2-bit integer defined in Table 6-27. It specifies how rectangles of a uniform mesh are split to form triangles.

Table 6-27 Specification of the triangulation type

triangle split code	Split
00	top-left to right bottom
01	bottom-left to top right
10	alternately top-left to bottom-right and bottom-left to top-right
11	alternately bottom-left to top-right and top-left to bottom-right

nr_of_mesh_nodes -- This is a 16-bit unsigned integer defining the total number of nodes (vertices) of a (non-uniform) Delaunay mesh. These nodes include both interior nodes as well as boundary nodes.

nr_of_boundary_nodes -- This is a 10-bit unsigned integer defining the number of nodes (vertices) on the boundary of a (non-uniform) Delaunay mesh.

node0_x -- This is a 10-bit integer specifying the x-coordinate of the first boundary node (vertex) of a mesh in half-pixel units with respect to a local coordinate system.

node0_y -- This is a 10-bit integer specifying the y-coordinate of the first boundary node (vertex) of a mesh in half-pixel units with respect to a local coordinate system.

delta_x_len_vlc - This is a variable-length code specifying the length of the delta_x code that follows. The delta_x_len_vlc and delta_x codes together specify the difference between the x-coordinates of a node (vertex) and the previously encoded node (vertex). The definition of the delta_x_len_vlc and delta_x codes are given in Table 11-32, the table for sprite motion trajectory coding.

delta_x -- This is an integer that defines the value of the difference between the x-coordinates of a node (vertex) and the previously encoded node (vertex) in half pixel units.

delta_y_len_vlc -- This is a variable-length code specifying the length of the delta_y code that follows. The delta_y_len_vlc and delta_y codes together specify the difference between the y-coordinates of a node (vertex) and the previously encoded node (vertex). The definition of the delta_y_len_vlc and delta_y codes are given in Table 11-32, the table for sprite motion trajectory coding.

delta_y -- This is an integer defines the value of the difference between the y-coordinates of a node (vertex) and the previously encoded node (vertex) in half pixel units.

6.3.10.3 Mesh motion

motion_range_code -- This is a 2-bit integer defined in Table 6-28. It specifies the dynamic range of motion vectors in half pel units.

Table 6-28 motion range code

motion range code	motion vector range
1	[-32, 31]
2	[-64, 63]
3	[-128, 127]

node_motion_vector_flag -- This is a 1 bit code specifying whether a node has a zero motion vector. When set to ‘1’ it indicates that a node has a zero motion vector, in which case the motion vector is not encoded. When set to ‘0’, it indicates the node has a nonzero motion vector and that motion vector data shall follow.

delta_mv_x_vlc -- This is a variable-length code defining (together with delta_mv_x_res) the value of the difference in the x-component of the motion vector of a node compared to the x-component of a predicting motion vector. The definition of the delta_mv_x_vlc codes are given in Table 11-11, the table for motion vector coding (MVD). The value delta_mv_x_vlc is given in half pixel units.

delta_mv_x_res -- This is an integer which is used in motion vector decoding as described in the section on video motion vector decoding, section 7.5.3. The number of bits in the bitstream for delta_mv_x_res is motion_range_code-1.

delta_mv_y_vlc -- This is a variable-length code defining (together with delta_mv_y_res) the value of the difference in the y-component of the motion vector of a node compared to the y-component of a predicting motion vector. The definition of the delta_mv_y_vlc codes are given in Table 11-11, the table for motion vector coding (MVD). The value delta_mv_y_vlc is given in half pixel units.

delta_mv_y_res -- This is an integer which is used in motion vector decoding as described in the section on video motion vector decoding, section 7.5.3. The number of bits in the bitstream for delta_mv_y_res is motion_range_code-1.

6.3.11 Face object

face_object_start_code -- The face_object_start_code is the bit string '000001BA' in hexadecimal. It initiates a face object.

Face_object_coding_type – This is a 2-bit integer indicating which coding method is used. Its meaning is described in Table 6-29.

Table 6-29 Face_object_coding_type

type value	Meaning
00	predictive coding
01	DCT (face_object_plane_group)
10	reserved
11	reserved

6.3.11.1 Face object plane

face_paramset_mask -- This is a 2-bit integer defined in Table 6-30. It indicates whether FAP data are present in the face_frame.

Table 6-30 Face parameter set mask

mask value	Meaning
00	unused
01	FAP present
10	reserved
11	reserved

face_object_plane_start_code -- The face_frame_start_code is the bit string '000001BB' in hexadecimal. It initiates a face object plane.

is_frame_rate – This is a 1-bit flag which when set to '1' indicates that frame rate information follows this bit field. When set to '0' no frame rate information follows this bit field.

is_time_code -- This is a 1-bit flag which when set to '1' indicates that time code information follows this bit field. When set to '0' no time code information follows this bit field.

time_code -- This is a 18-bit integer containing the following: time_code_hours, time_code_minutes, marker_bit and time_code_seconds as shown in Table 6-31. The parameters correspond to those defined in the IEC standard publication 461 for "time and control codes for video tape recorders". The time code refers to the first plane (in display order) after the GOV header. Table 6-31 shows the meaning of time_code.

Table 6-31 Meaning of time_code

time_code	range of value	No. of bits	Mnemonic
time_code_hours	0 - 23	5	uimsbf
time_code_minutes	0 - 59	6	uimsbf
marker_bit	1	1	bslbf
time_code_seconds	0 - 59	6	uimsbf

skip_frames – This is a 1-bit flag which when set to ‘1’ indicates that information follows this bit field that indicates the number of skipped frames. When set to ‘0’ no such information follows this bit field.

fap_mask_type -- This is a 2-bit integer. It indicates if the group mask will be present for the specified fap group, or if the complete faps will be present; its meaning is described in Table 6-32. In the case the type is ‘10’ the ‘0’ bit in the group mask indicates interpolate fap.

Table 6-32 fap mask type

mask type	Meaning
00	no mask nor fap
01	group mask
10	group mask’
11	fap

fap_group_mask[group_number] **-** This is a variable length bit entity that indicates, for a particular group_number which fap is represented in the bitstream. The value is interpreted as a mask of 1-bit fields. A 1-bit field in the mask that is set to ‘1’ indicates that the corresponding fap is present in the bitstream. When that 1-bit field is set to ‘0’ it indicates that the fap is not present in the bitstream. The number of bits used for the fap_group_mask depends on the group_number, and is given in Table 6-33.

Table 6-33 fap group mask bits

group_number	No. of bits
1	2
2	16
3	12
4	8
5	4
6	5
7	3
8	10
9	4
10	4

NFAP[group_number] **-** This indicates the number of FAPs in each FAP group. Its values are specified in the following table:

Table 6-34 NFAP definition

group_number	NFAP[group_number]
1	2
2	16
3	12
4	8
5	4
6	5
7	3
8	10
9	4
10	4

fap_quant – This is a 5-bit unsigned integer which is the quantization scale factor used to compute the FAPi table step size.

is_i_new_max – This is a 1-bit flag which when set to ‘1’ indicates that a new set of maximum range values for I frame follows these 4, 1-bit fields.

is_i_new_min – This is a 1-bit flag which when set to ‘1’ indicates that a new set of minimum range values for I frame follows these 4, 1-bit fields.

is_p_new_max – This is a 1-bit flag which when set to ‘1’ indicates that a new set of maximum range values for P frame follows these 4, 1-bit fields.

is_p_new_min – This is a 1-bit flag which when set to ‘1’ indicates that a new set of minimum range values for P frame follows these 4, 1-bit fields.

6.3.11.2 Face Object Prediction

skip_frames – This is a 1-bit flag which when set to ‘1’ indicates that information follows this bit field that indicates the number of skipped frames. When set to ‘0’ no such information follows this bit field.

6.3.11.3 Decode frame rate and frame skip

frame_rate – This is an 8 bit unsigned integer indicating the reference frame rate of the sequence.

seconds – This is a 4 bit unsigned integer indicating the fractional reference frame rate. The frame rate is computed as follows frame rate = (frame_rate + seconds/16).

frequency_offset -- This is a 1-bit flag which when set to ‘1’ indicates that the frame rate uses the NTSC frequency offset of 1000/1001. This bit would typically be set when frame_rate = 24, 30 or 60, in which case the resulting frame rate would be 23.97, 29.94 or 59.97 respectively. When set to ‘0’ no frequency offset is present. I.e. if (frequency_offset ==1) frame rate = (1000/1001) * (frame_rate + seconds/16).

number_of_frames_to_skip – This is a 4-bit unsigned integer indicating the number of frames skipped. If the number_of_frames_to skip is equal to 15 (pattern “1111”) then another 4-bit word follows allowing to skip up to 29 frames(pattern “11111110”). If the 8-bits pattern equals “11111111”, then another 4-bits word will follow and so on, and the number of frames skipped is incremented by 30. Each 4-bit pattern of ‘1111’ increments the total number of frames to skip with 15.

6.3.11.4 Decode new minmax

i_new_max[j] – This is a 5-bit unsigned integer used to scale the maximum value of the arithmetic decoder used in the I frame.

i_new_min[j] – This is a 5-bit unsigned integer used to scale the minimum value of the arithmetic decoder used in the I frame.

p_new_max[j] – This is a 5-bit unsigned integer used to scale the maximum value of the arithmetic decoder used in the P frame.

p_new_min[j] – This is a 5-bit unsigned integer used to scale the minimum value of the arithmetic decoder used in the P frame.

6.3.11.5 Decode viseme and expression

viseme_def -- This is a 1-bit flag which when set to '1' indicates that the mouth FAPs sent with the viseme FAP may be stored in the decoder to help with FAP interpolation in the future.

expression_def - This is a 1-bit flag which when set to '1' indicates that the FAPs sent with the expression FAP may be stored in the decoder to help with FAP interpolation in the future.

6.3.11.6 Face object plane group

face_object_plane_start_code – Defined in Section 6.3.11.1.

is_intra -- This is a 1-bit flag which when set to '1' indicates that the face object is coded in intra mode. When set to '0' it indicates that the face object is coded in predictive mode.

face_paramset_mask – Defined in Section 6.3.11.1.

is_frame_rate – Defined in Section 6.3.11.1.

is_time_code – Defined in Section 6.3.11.1.

time_code – Defined in Section 6.3.11.1.

skip_frames – Defined in Section 6.3.11.1

Fap_quant_index – This is a 5-bit unsigned integer used as the index to a fap_scale table for computing the quantization step size of DCT coefficients. The value of fap_scale is specified in the following list:

fap_scale[0 - 31] = { 1, 1, 2, 3, 5, 7, 8, 10, 12, 15, 18, 21, 25, 30, 35, 42,
50, 60, 72, 87, 105, 128, 156, 191, 234, 288, 355, 439, 543, 674, 836, 1039}

fap_mask_type -- Defined in Section 6.3.11.1

fap_group_mask[group_number] **-** Defined in Section 6.3.11.1

6.3.11.7 Face Object Group Prediction

skip_frames – See the definition in Section 6.3.11.1.

6.3.11.8 Decode frame rate and frame skip

frame_rate – See the definition in Section 6.3.11.3.

frequency_offset -- See the definition in Section 6.3.11.3.

number_of_frames_to_skip – See the definition in Section 6.3.11.3.

6.3.11.9 Decode viseme_segment and expression_segment

viseme_segment_select1Q[k] – This is the quantized value of viseme_select1 at frame k of a viseme FAP segment.

viseme_segment_select2Q[k] – This is the quantized value of viseme_select2 at frame k of a viseme FAP segment.

viseme_segment_blendQ[k] – This is the quantized value of viseme_blend at frame k of a viseme FAP segment.

viseme_segment_def[k] – This is a 1-bit flag which when set to '1' indicates that the mouth FAPs sent with the viseme FAP at frame k of a viseme FAP segment may be stored in the decoder to help with FAP interpolation in the future.

viseme_segment_select1Q_diff[k] -- This is the prediction error of viseme_select1 at frame k of a viseme FAP segment.

viseme_segment_select2Q_diff[k] – This is the prediction error of viseme_select2 at frame k of a viseme FAP segment.

viseme_segment_blendQ_diff[k] – This is the prediction error of viseme_blend at frame k of a viseme FAP segment.

expression_segment_select1Q[k] – This is the quantized value of expression_select1 at frame k of an expression FAP segment.

expression_segment_select2Q[k] – This is the quantized value of expression_select2 at frame k of an expression FAP segment.

expression_segment_intensity1Q[k] – This is the quantized value of expression_intensity1 at frame k of an expression FAP segment

expression_segment_intensity2Q[k] – This is the quantized value of expression_intensity2 at frame k of an expression FAP segment

expression_segment_select1Q_diff[k] -- This is the prediction error of expression_select1 at frame k of an expression FAP segment.

expression_segment_select2Q_diff[k] – This is the prediction error of expression_select2 at frame k of an expression FAP segment.

expression_segment_intensity1Q_diff[k] – This is the prediction error of expression_intensity1 at frame k of an expression FAP segment.

expression_segment_intensity2Q_diff[k] – This is the prediction error of expression_intensity2 at frame k of an expression FAP segment.

expression_segment_init_face[k] - This is a 1-bit flag which indicates the value of init_face at frame k of an expression FAP segment.

expression_segment_def[k] - This is a 1-bit flag which when set to '1' indicates that the FAPs sent with the expression FAP at frame k of a viseme FAP segment may be stored in the decoder to help with FAP interpolation in the future.

6.3.11.10 Decode i_dc, p_dc, and ac

dc_Q - This is the quantized DC component of the DCT coefficients. For an intra FAP segment, this component is coded as a signed integer of either 16 bits or 31 bits. The DCT quantization parameters of the 68 FAPs are specified in the following list:

```
DCTQP[1 - 68] = {1,    1,    7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
                 7.5,  7.5,  7.5,  15,   15,   15,   15,   5,    10,   10,
                 10,   10,   425,  425,  425,  425,  5,    5,    5,    5,
                 7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  20,   20,
                 20,   20,   10,   10,   10,   10,   255,  170,  255,  255,
                 7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
                 15,   15,   15,   15,   10,   10,   10,   10}
```

For DC coefficients, the quantization stepsize is obtained as follows:

qstep[i] = fap_scale[fap_quant_inex] * DCTQP[i] ÷ 3.0

dc_Q_diff - This is the quantized prediction error of a DC coefficient of an inter FAP segment. Its value is computed by subtracting the decoded DC coefficient of the previous FAP segment from the DC coefficient of the current FAP segment. It is coded by a variable length code if its value is within [-255, +255]. Outside this range, its value is coded by a signed integer of 16 or 32 bits.

count_of_runs - This is the run length of zeros preceding a non-zero AC coefficient.

ac_Q[i][next] - This is a quantized AC coefficients of a segment of FAPi. For AC coefficients, the quantization stepsize is three times larger than the DC quantization stepsize and is obtained as follows:

qstep[i] = fap_scale[fap_quant_inex] * DCTQP[i]

7. The visual decoding process

This clause specifies the decoding process that the decoder shall perform to recover visual data from the coded bit-stream. As shown in Figure 7-1, the visual decoding process includes several decoding processes such as shape-motion-texture decoding, still texture decoding, mesh decoding, and face decoding processes. After decoding the coded bit stream, it is then sent to the compositor to integrate various visual objects.

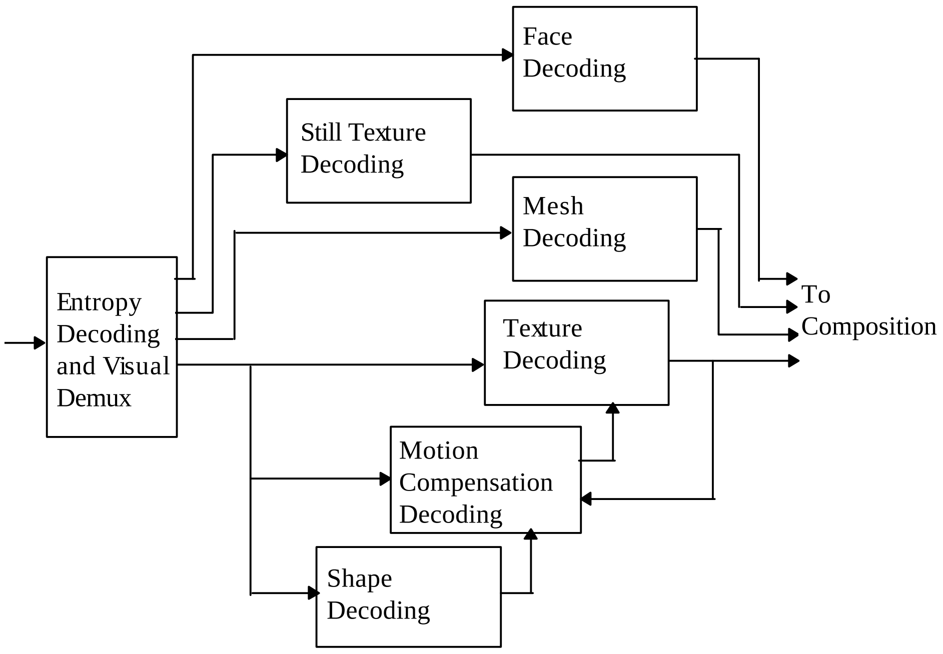


Figure 7-1 A high level view of basic visual decoding; specialized decoding such as scalable, sprite and error resilient decoding are not shown

In clauses 7.1 through 7.6 the VOP decoding process is specified in which shape, motion, texture decoding processes are the major contents. The still texture object decoding process along with view dependent object decoding are described in clauses 7.7 and 7.8, respectively. Clause 7.9 includes the mesh decoding process, and clause 7.10 features the face object decoding process. The output of the decoding process is explained at the end of clause 7.

7.1 Video decoding process

This clause specifies the decoding process that a decoder shall perform to recover VOP data from the coded video bitstream.

With the exception of the Inverse Discrete Cosine Transform (IDCT) the decoding process is defined such that all decoders shall produce numerical identical results. Any decoding process that produces identical results to the process described here, by definition, complies with this specification.

The IDCT is defined statistically such that different implementations for this function are allowed. The IDCT specification is given in Annex A.

Figure 7-2 is a diagram of the Video Decoding Process without any scalability feature. The diagram is simplified for clarity. The same decoding scheme is applied when decoding all the VOPs of a given session

Note: Throughout this specification two dimensional arrays are represented as *name*[q][p] where 'q' is the index in the vertical dimension and 'p' the index in the horizontal dimension.

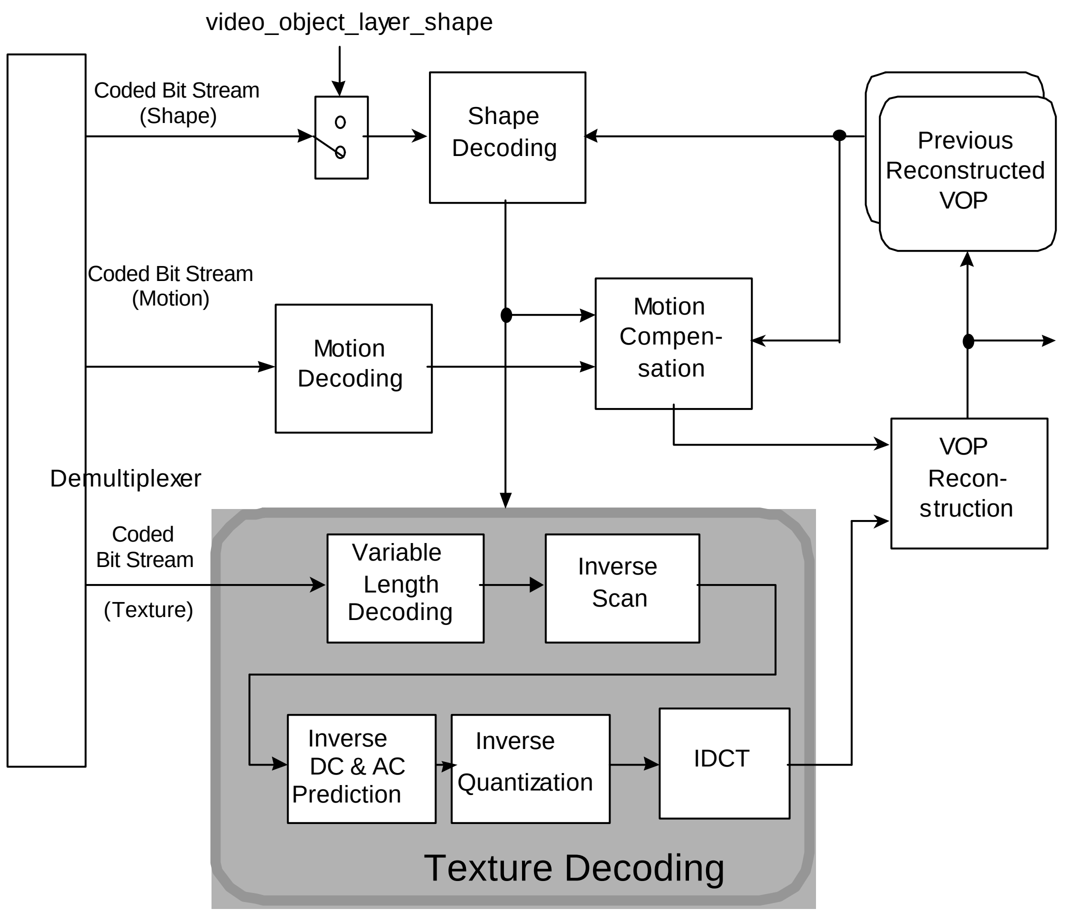


Figure 7-2 Simplified Video Decoding Process

The decoder is mainly composed of three parts : the shape decoder, motion decoder and texture decoder. The reconstructed VOP is obtained by combining the decoded shape, texture and motion information.

7.2 Higher syntactic structures

The various parameters and flags in the bitstream for VideoObjectLayer(), Group_of_VideoObjectPlane(), VideoObjectPlane(),video_plane_with_short_header(), macroblock() and block(), as well as other syntactic structures related to them shall be interpreted as discussed earlier. Many of these parameters and flags affect the decoding process. Once all the macroblocks in a given VOP have been processed, the entire VOP will have been reconstructed. In case the bitstream being decoded contains B-VOPs, reordering of VOPs may be needed as discussed in sec. 6.1.1.7.

7.3 Texture decoding

This clause describes the process used to decode the texture information of a VOP. The process of video texture is given in Figure 7-3.

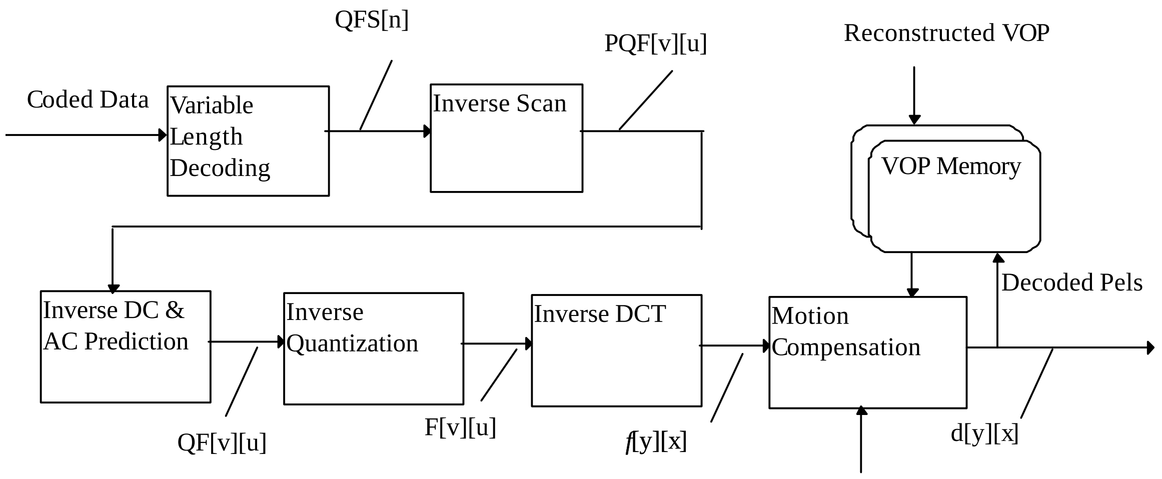


Figure 7-3 Video Texture Decoding Process

7.3.1 Variable length decoding

This section explains the decoding process. Section 7.3.1.1 specifies the process used for the DC coefficients (n=0) in an intra coded block. (n is the index of the coefficient in the appropriate zigzag scan order). Section 7.3.1.2 specifies the decoding process for all other coefficients; AC coefficients ($n \neq 0$) and DC coefficients in non-intra coded blocks.

Let cc denote the color component. It is related to the block number as specified in Table 7-1; thus cc is zero for the Y component, one and two for the first and second chrominance components respectively.

Table 7-1 Color component identification

Block Number	***cc***
	4:2:0
0	0
1	0
2	0
3	0
4	1
5	2

7.3.1.1 DC coefficients decoding in intra blocks

Differential dc coefficients in blocks in intra macroblocks are encoded as variable length code denoting dct_dc_size as defined in Table 11-12 and Table 11-13 in Annex B, and a fixed length code dc_dct_differential (Table 11-14). The dct_dc_size categorizes the c coefficients according to their "size". For each category additional bits are appended to the dct_dc_size code to uniquely identify which difference in that category actually occurred (Table 11-14). This is done by appending a fixed length code, dc_dct_differential, of dct_dc_size bits. The final value of the decoded dc coefficient is the sum of this latter differential dc value and the predicted value.

When short_video_header is 1, the dc coefficient of an intra block is not coded differentially. It is instead transmitted as a fixed length unsigned integer code of size 8 bits, unless this integer has the value 255. The values 0 and 128 shall not be used – they are reserved. If the integer value is 255, this is interpreted as a signalled value of 128.

7.3.1.2 Other coefficients

The ac coefficients are obtained by decoding the variable length codes to produce EVENTs. An EVENT is a combination of a last non-zero coefficient indication (LAST; "0": there are more nonzero coefficients in this block, "1": this is the last nonzero coefficient in this block), the number of successive zeros preceding the coded coefficient (RUN), and the non-zero value of the coded coefficient (LEVEL).

The most commonly occurring EVENTs for the luminance and chrominance components of intra blocks are decoded by referring to Table 11-15. The most commonly occurring EVENTs for the luminance and chrominance components of inter blocks are decoded be referring to Table 11-16. The last bit "s" denotes the sign of level, "0" for positive and "1" for negative. The remaining combinations of (LAST, RUN, LEVEL) are decoded as described in clause 7.3.1.3.

When short_video_header is 1, the most commonly occurring EVENTS are coded with the variable length codes given in Table 11-16 (for all coefficients other than intra DC whether in intra or inter blocks). The last bit "s" denotes the sign of level, "0" for positive and "1" for negative.

When short_video_header is 0, the variable length code table is different for intra blocks and inter blocks.

7.3.1.3 Escape code

Many possible combinations of runs and levels have no variable length code to represent them. In order to encode these statistically rare combinations an Escape Coding method is used. The escape codes of DCT coefficients are encoded in four modes. The first three of these modes are used when short_video_header is 0, and the fourth is used when short_video_header is 1. Their decoding process is specified below.

Type 1 : ESC is followed by "0", and the code following ESC + "0" is decoded as a variable length code using the standard Tcoef VLC codes given in Tables 11-15 and 11-16, but the values of LEVEL are modified following decoding to give the restored value $LEVEL^S$, as follows:

$$LEVEL^S = \text{sign}(LEVEL^+) \times [\ \text{abs}(LEVEL^+) + LMAX\ ]$$

where $LEVEL^+$ is the value after variable length decoding and LMAX is obtained from Table 11-18 and Table 11-19 as a function of the decoded values of RUN and LAST.

Type 2 : ESC is followed by “10”, and the code following ESC + “10” is decoded as a variable length code using the standard Tcoef VLC codes given in Table 11-15 and Table 11-16, but the values of RUN are modified following decoding to give the restored value RUN^S, as follows:

$$RUN^S = RUN^+ + (RMAX + 1)$$

where RUN^+ is the value after variable length decoding. RMAX is obtained from Table 11-20 and Table 11-21 as a function of the decoded values of LEVEL and LAST.

Type 3 : ESC is followed by “11”, and the code following ESC + “11” is decoded as fixed length codes. This type of escape codes are represented by 1-bit LAST, 6-bit RUN and 12-bit LEVEL. Use of this escape sequence for encoding the combinations listed in Table 11-15 and Table 11-16 prohibited. The codes for RUN and LEVEL are given in Table 11-17.

Type 4: The fourth type of escape code is used if and only if short_video_header is 1. In this case, the 15 bits following ESC are decoded as fixed length codes represented by 1-bit LAST, 6-bit RUN and 8-bit LEVEL. The values 0000 0000 and 1000 000 for LEVEL are not used (they are reserved).

7.3.1.4 Intra dc coefficient decoding for the case of switched vlc encoding

At the VOP layer, using quantizer value as the threshold, a 3 bit code (intra_dc_vlc_thr) allows switching between 2 VLCs (DC Intra VLC and AC Intra VLC) when decoding DC coefficients of Intra macroblocks, see Table 6-12.

Note: When the intra AC VLC is turned on, Intra DC coefficients are not handled separately any more, but treated the same as all other coefficients. That means that a zero Intra DC coefficient will not be coded but will simply increase the run for the following AC coefficients. The definitions of MCBPC and CBPY in Section 6.3.7 6.3.6 are changed accordingly.

7.3.2 Inverse scan

This clause specifies the way in which the one dimensional data, QFS[n] is converted into a two-dimensional array of coefficients denoted by PQF[v][u] where u and v both lie in the range of 0 to 7. Let the data at the output of the variable length decoder be denoted by QFS[n] where n is in the range of 0 to 63. Three scan patterns are defined as shown in Figure 7-4. The scan that shall be used is determined by the following method. For intra blocks, if acpred_flag=0, zigzag scan is selected for all blocks in a macroblock. Otherwise, DC prediction direction is used to select a scan on block basis. For instance, if the DC prediction refers to the horizontally adjacent block, alternate-vertical scan is selected for the current block. Otherwise (for DC prediction referring to vertically adjacent block), alternate-horizontal scan is used for the current block. For all other blocks, the 8x8 blocks of transform coefficients are scanned in the “zigzag” scanning direction.

0	1	2	3	10	11	12	13
4	5	8	9	17	16	15	14
6	7	19	18	26	27	28	29
20	21	24	25	30	31	32	33
22	23	34	35	42	43	44	45
36	37	40	41	46	47	48	49
38	39	50	51	56	57	58	59
52	53	54	55	60	61	62	63

0	4	6	20	22	36	38	52
1	5	7	21	23	37	39	53
2	8	19	24	34	40	50	54
3	9	18	25	35	41	51	55
10	17	26	30	42	46	56	60
11	16	27	31	43	47	57	61
12	15	28	32	44	48	58	62
13	14	29	33	45	49	59	63

0	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

Figure 7-4 (a) Alternate-Horizontal scan (b) Alternate-Vertical scan (c) Zigzag scan

7.3.3 Intra dc and ac prediction for intra macroblocks

This clause specifies the prediction process for decoding of coefficients. When short_video_header is "0". When short_video_header is "1", this prediction process is not performed.

7.3.3.1 DC and AC Prediction Direction

This adaptive selection of the DC and AC prediction direction is based on comparison of the horizontal and vertical DC gradients around the block to be decoded. Figure 7-5 shows the three blocks surrounding the block to be decoded. Block 'X', 'A', 'B' and 'C' respectively refer to the current block, the previous block, the above-left block, and the block immediately above, as shown.

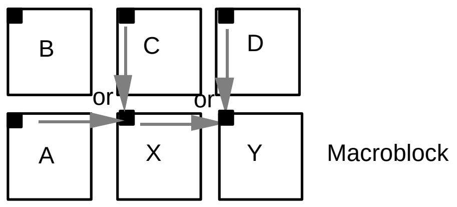


Figure 7-5 Previous neighboring blocks used in DC prediction

The inverse quantized DC values of the previous decoded blocks, F[0][0], are used to determine the direction of the DC and AC prediction as follows.

```
if ( $|F_A[0][0] – F_B[0][0]| < |F_B[0][0] – F_C[0][0]|$)
        predict from block C
else
        predict from block A
```

If any of the blocks A, B or C are outside of the VOP boundary, or the video packet boundary, or they do not belong to an intra coded macroblock, their F[0][0] values are assumed to take a value of $2^{(bits\_per\_pixel+2)}$ and are used to compute the prediction values.

7.3.3.2 Adaptive DC Coefficient Prediction

The adaptive DC prediction method involves selection of either the F[0][0] value of immediately previous block or that of the block immediately above it (in the previous row of blocks) depending on the prediction direction determined above.

```
if (predict from block C)
        $QF_X[0][0] = PQF_X[0][0] + F_C[0][0] // dc\_scaler$
else
        $QF_X[0][0] = PQF_X[0][0] + F_A[0][0] // dc\_scaler$
```

Dc_scalar is define in Table 7-2. This process is independently repeated for every block of a macroblock using appropriate immediately horizontally adjacent block 'A' and immediately vertically adjacent block 'C'.

DC predictions are performed similarly for the luminance and each of the two chrominance components.

7.3.3.3 Adaptive ac coefficient prediction

This process is used when ac_pred_flag = ‘1’, which indicates that AC prediction is performed when decoding the coefficients.

Either coefficients from the first row or the first column of a previous coded block are used to predict the co-sited coefficients of the current block. On a block basis, the best direction (from among horizontal and vertical directions) for DC coefficient prediction is also used to select the direction for AC coefficients prediction; thus, within a macroblock, for example, it becomes possible to predict each block independently from either the horizontally adjacent previous block or the vertically adjacent previous block. The AC coefficients prediction is illustrated in Figure 7-6.

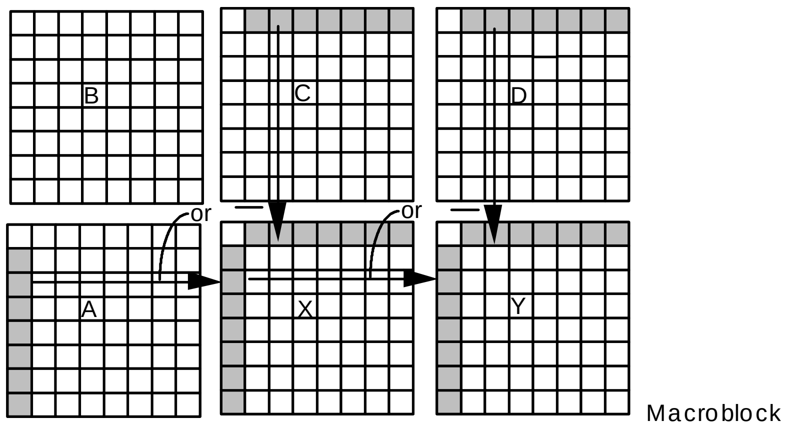


Figure 7-6 Previous neighboring blocks and coefficients used in AC prediction

To compensate for differences in the quantization of previous horizontally adjacent or vertically adjacent blocks used in AC prediction of the current block, scaling of prediction coefficients becomes necessary. Thus the prediction is modified so that the predictor is scaled by the ratio of the current quantisation stepsize and the quantisation stepsize of the predictor block. The definition is given in the equations below.

If block ‘A’ was selected as the predictor for the block for which coefficient prediction is to be performed, we calculate the horizontal AC prediction as follows.

$$QAC_{i0X'} = \frac{QAC_{i0A} \times QP_A}{QP_X}$$

$$QF_X[0][i] = PQF_X[0][i] + (QF_A[0][i] * QP_A) // QP_X$$

If block ‘C’ was selected as the predictor for the block for which coefficient prediction is to be performed, we calculate the vertical AC prediction as follows.

$$QAC_{0jX'} = \frac{QAC_{0jC} \times QP_C}{QP_X}$$

$$QF_X[j][0] = PQF_X[j][0] + (QF_C[j][0] * QP_C) // QP_X$$

If the prediction block (block 'A' or block 'C') is outside of the boundary of the VOP or video packet, then all the prediction coefficients of that block are assumed to be zero.

7.3.4 Inverse quantisation

The two-dimensional array of coefficients, $QF[v][u]$, is inverse quantised to produce the reconstructed DCT coefficients. This process is essentially a multiplication by the quantiser step size. The quantiser step size is modified by two mechanisms; a weighting matrix is used to modify the step size within a block and a scale factor is used in order that the step size can be modified at the cost of only a few bits (as compared to encoding an entire new weighting matrix).

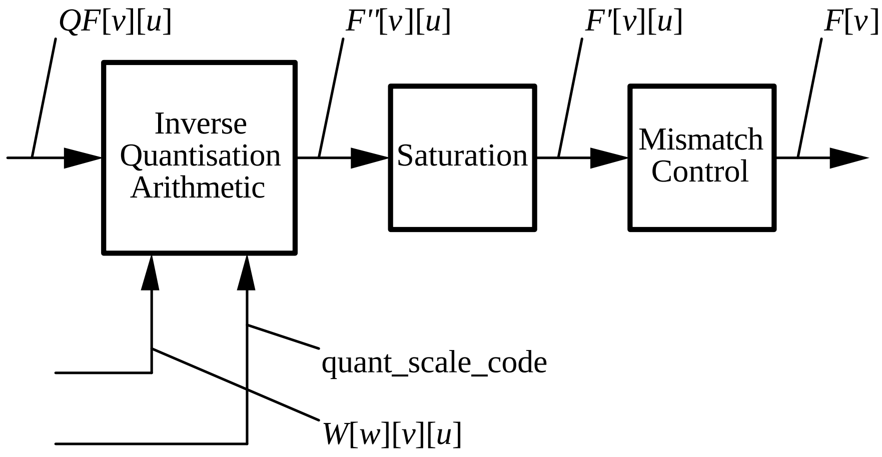


Figure 7-7 Inverse quantisation process

Figure 7-7 illustrates the overall inverse quantisation process. After the appropriate inverse quantisation arithmetic the resulting coefficients, $F''[v][u]$, are saturated to yield $F'[v][u]$ and then a mismatch control operation is performed to give the final reconstructed DCT coefficients, $F[v][u]$.

NOTE - Attention is drawn to the fact that the method of achieving mismatch control in this specification is identical to that employed by ISO/IEC 13818-2.

7.3.4.1 First inverse quantisation method

This clause specifies the first of the two inverse quantisation methods. . The method described here is used when „quant_type==1“

7.3.4.1.1 Intra dc coefficient

The DC coefficients of intra coded blocks shall be inverse quantised in a different manner to all other coefficients.

In intra blocks $F''[0][0]$ shall be obtained by multiplying $QF[0][0]$ by a constant multiplier,

The reconstructed DC values are computed as follows.

F''[0][0] = *dc_scaler* * *QF*[0][0]

When short_video_header is 1, dc_scaler is 8, otherwise dc_scaler is defined in Table 7-2.

7.3.4.1.2 Other coefficients

All coefficients other than the DC coefficient of an intra block shall be inverse quantised as specified in this clause. Two weighting matrices are used. One shall be used for intra macroblocks and the other for non-intra macroblocks. Each matrix has a default set of values which may be overwritten by down-loading a user defined matrix.

Let the weighting matrices be denoted by *W*[*w*][*v*][*u*] where *w* takes the values 0 to 1 indicating which of the matrices is being used. *W*[*0*][*v*][*u*] is for intra macroblocks, and *W*[*1*][*v*][*u*] is for non-intra macroblocks. The following equation specifies the arithmetic to reconstruct *F''*[*v*][*u*] from *QF*[*v*][*u*] (for all coefficients except intra DC coefficients).

$$F''[v][u] = ((2 \times QF[v][u] + k) \times W[w][v][u] \times quantiser\_scale) / 32$$

$$\text{where:}$$

$$k = \begin{cases} 0 & \text{intra blocks} \\ Sign(QF[v][u]) & \text{non-intra blocks} \end{cases}$$

NOTE - The above equation uses the "/" operator as defined in 4.1.

7.3.4.2 Second inverse quantisation method

This clause specifies the second of the two inverse quantisation methods. . The method described here is used when „quant_type==0". The quantization parameter *quantiser_scale* may take integer values from 1 to 31. The quantization stepsize is 2x *quantiser_scale*.

7.3.4.2.1 Dequantisation

$$|F''[v][u]| = \begin{array}{l} 0, \quad \text{if } QF[v][u] = 0 \\ 2 \times QF[v][u] \times quantiser\_scale + quantiser\_scale, \quad \text{if } QF[v][u] \neq 0,\ quantiser\_scale \text{ is odd }. \\ 2 \times QF[v][u] \times quantiser\_scale + quantiser\_scale - 1, \quad \text{if } QF[v][u] \neq 0,\ quantiser\_scale \text{ is ev} \end{array}$$

The sign of *QF[v][u]* is then added to obtain *F"[v][u]´*: *F"[v][u]*= Sign(*QF[v][u]*)x|*F"[v][u]*|
Clipping to [-2^{N_bit+3} : 2^{N_bit+3} +1] is performed before IDCT.

7.3.4.3 Optimised nonlinear inverse quantisation

Note: This section is valid for both quantization methods.

Within an Intra macroblock for which short_video_header is 0, luminance blocks are called type 1 blocks, chroma blocks are classified as type 2. When short_video_header is 1, the inverse quantization of DC intra coefficients is equivalent to using a fixed value of dc_scaler = 8, as described above in clause 7.3.4.1.1

- DC coefficients of Type 1 blocks are quantized by Nonlinear Scaler for Type 1
- DC coefficients of Type 2 blocks are quantized by Nonlinear Scaler for Type 2

Table 7-2 specifies the nonlinear dc_scaler expressed in terms of piece-wise linear characteristics.

Table 7-2 Non linear scaler for DC coefficients of DCT blocks, expressed in terms of relation with quantizer_scale

Component:Type	dc_scaler for quantiser_scale range			
	1 through 4	5 through 8	9 through 24	>= 25
Luminance: Type1	8	2x quantiser_scale	quantiser_scale +8	2 x quantiser_scale - 16
Chrominance: Type2	8	(quantiser_scale +13)/2		quantiser_scale -6

The reconstructed DC values are computed as follows.

*F"[0][0]= QF[0][0]*x *dc_scaler*

7.3.4.4 Saturation

The coefficients resulting from the Inverse Quantisation Arithmetic are saturated to lie in the range $[-2^{\text{bits\_per\_pixel} + 3}, 2^{\text{bits\_per\_pixel} + 3} - 1]$. Thus:

$$F'[v][u] = \begin{cases} 2047 & F''[v][u] > 2047 \\ F''[v][u] & -2048 \le F''[v][u] \le 2047 \\ -2048 & F''[v][u] < -2048 \end{cases}$$

7.3.4.5 Mismatch control

This mismatch control is only applicable to the first inverse quantization method. Mismatch control shall be performed by any process equivalent to the following. Firstly all of the reconstructed, saturated coefficients, *F'[v]*[*u*] in the block shall be summed. This value is then tested to determine whether it is odd or even. If the sum is even then a correction shall be made to just one coefficient; *F*[7][7]. Thus:

$$sum = \sum_{v=0}^{v<8} \sum_{u=0}^{u<8} F'[v][u]$$

$$F[v][u] = F'[v][u] \text{ for all } u,\ v \text{ except } u = v = 7$$

$$F[7][7] = \begin{cases} F'[7][7] & \text{if } sum \text{ is odd} \\ \left\{ \begin{array}{ll} F'[7][7]-1 & \text{if } F'[7][7] \text{ is odd} \\ F'[7][7]+1 & \text{if } F'[7][7] \text{ is even} \end{array} \right\} & \text{if } sum \text{ is even} \end{cases}$$

NOTE 1 It may be useful to note that the above correction for *F*[7][7] may simply be implemented by toggling the least significant bit of the twos complement representation of the coefficient. Also since only the "oddness" or "evenness" of the *sum* is of interest an exclusive OR (of just the least significant bit) may be used to calculate "*sum*".

NOTE 2 Warning. Small non-zero inputs to the IDCT may result in zero output for compliant IDCTs. If this occurs in an encoder, mismatch may occur in some pictures in a decoder that uses a different compliant IDCT. An encoder should avoid this problem and may do so by checking the output of its own IDCT. It should ensure that it never inserts any non-zero coefficients into the bitstream when the block in question reconstructs to zero through its own IDCT function. If this action is not taken by the encoder, situations can arise where large and very visible mismatches between the state of the encoder and decoder occur.

7.3.4.6 Summary of quantiser process for method 1

In summary the inverse quantisation process is any process numerically equivalent to:

```
for (v=0; v<8;v++) {
    for (u=0; u<8;u++) {
        if (QF[v][u] == 0)
            F’’[v][u] = 0;
        else if ( (u==0) && (v==0) && (macroblock_intra) ) {
            F''[v][u] = dc_scaler * QF[v][u];
        } else {
            if ( macroblock_intra ) {
                F''[v][u] = ( QF[v][u] * W[0][v][u] * quantiser_scale * 2 ) / 32;
            } else {
                F''[v][u] = ( ( ( QF[v][u] *  2 ) + Sign(QF[v][u]) ) * W[1][v][u]
                                                    * quantiser_scale ) / 32;
            }
        }
    }
}

sum = 0;
for (v=0; v<8;v++) {
    for (u=0; u<8;u++) {
        if ( F’’[v][u] > $2^{bits\_per\_pixel + 3}$ – 1 ) {
            F’[v][u] = $2^{bits\_per\_pixel + 3}$ – 1;
        } else {
            if ( F’’[v][u] < $-2^{bits\_per\_pixel + 3}$ ) {
                F’[v][u] = $-2^{bits\_per\_pixel + 3}$ ;
            } else {
                F’[v][u] = F'‘[v][u];
            }
        }
    sum = sum + F’[v][u];
    F[v][u] = F’[v][u];
    }
}

if ((sum & 1) == 0) {
    if ((F[7][7] & 1) != 0) {
        F[7][7] = F'[7][7] - 1;
    } else {
        F[7][7] = F'[7][7] + 1;
    }
}
```

7.3.5 Inverse DCT

Once the DCT coefficients, F[u][v] are reconstructed, the inverse DCT transform defined in Annex A shall be applied to obtain the inverse transformed values, $f[y][x]$.These values shall be saturated so that: $-2^{N\_bit} \leq f[y][x] \leq 2^{N\_bit} - 1$, for all x, y.

7.4 Shape decoding

Binary shape decoding is based on a block-based representation. The primary coding methods are block-based context-based binary arithmetic decoding and block-based motion compensation. The primary data structure used is denoted as the binary alpha block (bab). The bab is a square block of binary valued pixels representing the opacity/transparency for the pixels in a specified block-shaped spatial region of size 16x16 pels. In fact, each bab is co-located with each texture macroblock.

7.4.1 Higher syntactic structures

7.4.1.1 Vol decoding

If video_object_layer_shape is equal to '00' then no binary shape decoding is required. Otherwise, binary shape decoding is carried out.

7.4.1.2 VOP decoding

If video_object_layer_shape is not equal to '00' then, for each subsequent VOP, the dimensions of the bounding box of the reconstructed VOP are obtained from:

- VOP_width
- VOP_height

If these decoded dimensions are not multiples of 16, then the values of VOP_width and VOP_height are rounded up to the nearest integer, which is a multiple of 16.

Additionally, in order to facilitate motion compensation, the horizontal and spatial position of the VOP are obtained from:

- VOP_horizontal_mc_spatial_ref
- VOP_vertical_mc_spatial_ref

These spatial references may be different for each VOP but the same coordinate system must be used for all VOPs within a vol. Additionally, the decoded spatial references must have an even value.

- VOP_shape_coding_type

This flag is used in error resilient mode and enables the use of intra shape codes in P-VOPs. Finally, in the VOP class, it is necessary to decode

- change_conv_ratio_disable

This specifies whether conv_ratio is encoded at the macroblock layer.

Once the above elements have been decoded, the binary shape decoder may be applied to decode the shape of each macroblock within the bounding box.

7.4.2 Macroblock decoding

The shape information for each macroblock residing within the bounding box of the VOP is decoded into the form of a 16x16 bab.

7.4.2.1 Mode decoding

Each bab belongs to one of seven types listed in Table 7-3. The type information is given by the bab_type field which influences decoding of further shape information. For I-VOPs only three out of the seven modes are allowed as shown in Table 7-3.

Table 7-3 List of bab types

bab_type	Semantic	Used in
0	MVDs==0 && No Update	P- ,B-VOPs
1	MVDs!=0 && No Update	P- ,B-VOPs
2	Transparent	All VOP types
3	Opaque	All VOP types
4	IntraCAE	All VOP types
5	MVDs==0 && interCAE	P- ,B-VOPs
6	MVDs!=0 && interCAE	P- ,B-VOPs

7.4.2.1.1 I-VOPs

Suppose that `f(x,y)` is the bab_type of the bab located at `(x,y)`, where x is the BAB column number and y is the BAB row number. The code word for the bab_type at the position `(i,j)` is determined as follows. A context C is computed from previously decoded bab_type's.

$$C = 27*(f(i-1,j-1)-2) + 9*(f(i,j-1)-2) + 3*(f(i+1,j-1)-2) + (f(i-1,j)-2)$$

If f(x,y) references a bab outside the current VOP, bab_type is assumed to be transparent for that bab (i.e. `f(x,y)=2`). The bab_type of babs outside the current video packet is also assumed to be transparent. The VLC used to decode bab_type for the current bab is switched according to the value of the context C. This context-switched VLC table is given in Table 11-26.

7.4.2.1.2 P- and B-VOPs

The decoding of the current bab_type is dependent on the bab_type of the co-located bab in the reference VOP. The reference VOP is either a forward reference VOP or a backward reference VOP. The forward reference VOP is defined as the most recent non-empty (i.e. VOP_coded != 0) I- or P-VOP in the past, while the backward VOP is defined as the most recently decoded I- or P-VOP in the future. If the current VOP is a P-VOP, the forward reference VOP is selected as the reference VOP. If the current VOP is a B-VOP the following decision rules are applied:

1. If one of the reference VOPs is empty, the non-empty one (forward/backward) is selected as the reference VOP for the current B-VOP.

2. If both reference VOPs are non-empty, the forward reference VOP is selected if its temporal distance to the current B-VOP is not larger than that of the backward reference VOP, otherwise, the backward one is chosen.

In the special cases when closed_GOV == 1 and the forward reference VOP belongs to the previous GOV, the current B-VOP takes the backward VOP as reference.

If the sizes of the current and reference VOPs are different, some babs in the current VOP may not have a co-located equivalent in the previous VOP. Therefore the bab_type matrix of the previous VOP is manipulated to match the size of the current VOP. Two rules are defined for that purpose, namely a cut rule and a copy rule:

- ***cut rule***. If the number of lines (respectively columns) is smaller in the current VOP than in the previous one, the bottom lines (respectively rightmost columns) are eliminated from the reference VOP such that both VOP sizes match.

- ***copy rule***. If the number of lines (respectively columns) is larger in the current VOP than in the previous one, the bottom line (respectively rightmost column) is replicated as many times as needed in the reference VOP such that both VOP sizes match.

An example is shown in Figure 7-8 where both rules are applied.

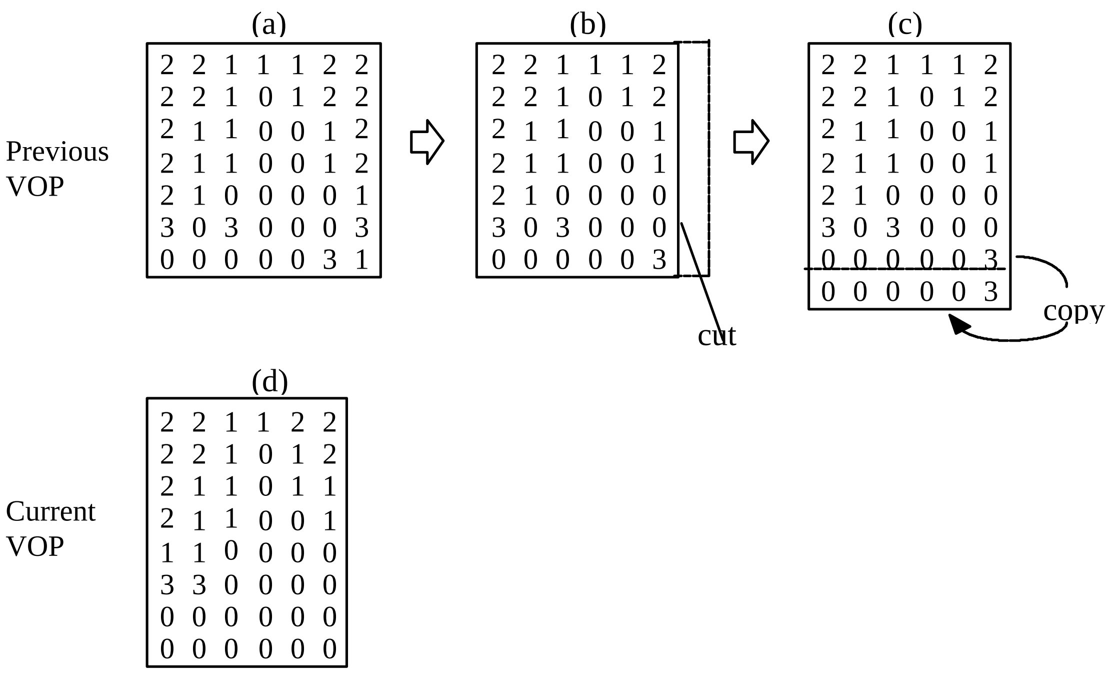


Figure 7-8 Example of size fitting between current VOP and reference VOP. The numbers represent the type of each bab.

The VLC to decode the current bab_type is switched according to the value of bab_type of the co-located bab in the reference VOP. This context-switched VLC tables for I, P and B VOPs are given in.Table 11-26 and Table 11-27. If the type of the bab is transparent, then the current bab is filled with zero (transparent) values. A similar procedure is carried out if the type is opaque, where the reconstructed bab is filled with values of 255 (opaque). For both transparent and opaque types, no further decoding of shape-related data is required for the current bab. Otherwise further decoding steps are necessary, as listed in Table 7-4. Decoding for motion compensation is described in Section 7.4.2.2, and cae decoding in Section 7.4.2.5.

Table 7-4 Decoder components applied for each type of bab

bab_type	Motion compensation	CAE decoding
0	yes	no
1	yes	no
2	no	no
3	no	no
4	no	yes
5	yes	yes

6	yes	yes

7.4.2.2 Binary alpha block motion compensation

Motion Vector of shape (MVs) is used for motion compensation (MC) of shape. The value of MVs is reconstructed as described in section **7.4.2.3**. Integer pixel motion compensation is carried out on a 16x16 block basis according to section **7.4.2.4**. Overlapped MC, half sample MC and 8x8 MC are not carried out.

If bab_type is MVDs==0 && No Update or MVDs!=0 && No Update then the motion compensated bab is taken to be the decoded bab, and no further decoding of the bab is necessary. Otherwise, cae decoding is required.

7.4.2.3 Motion vector decoding

If bab_type indicates that MVDs!=0, then mvds_x and mvds_y are VLC decoded. For decoding mvds_x, the VLC given in Table 11-28 is used. The same table is used for decoding mvds_y, unless the decoded value of mvds_x is zero. If MVDs = 0, the VLC given in Table 11-29 is used for decoding mvds_y. If bab_type indicates that MVDs==0, then both mvds_x and mvds_y are set to zero.

The integer valued shape motion vector MVs=(MVs_x,MVs_y) is determined as the sum of a predicted motion vector MVPs and MVDs = (MVDs_x,MVDs_y), where MVPs is determined as follows.

MVPs is determined by analysing certain candidate motion vectors of shape (MVs) and motion vectors of selected texture blocks (MV) around the MB corresponding to the current bab. They are located and denoted as shown in Figure 7-9 where MV1, MV2 and MV3 are rounded up to integer values towards 0. If the selected texture block is a field predicted macroblock, then MV1, MV2 or MV3 are generated by averaging the two field motion vectors and rounding toward zero. Regarding the texture MVs, the convention is that a MB possessing only 1 MV is considered the same as a MB possessing 4 MVs, where the 4 MVs are equal. By traversing MVs1, MVs2, MVs3, MV1, MV2 and MV3 in this order, MVPs is determined by taking the first encountered MV that is defined. That is, for INTER coded MBs, there will exist a defined motion vector for texture. For BABs, with bab_type = 0,1,5 or 6, there will exist a defined motion vector of shape. No valid motion vectors will exist in INTRA coded MBs and BABs. If no candidate motion vectors is defined, MVPs = (0,0).

In the case that video_object_layer_shape is “binary_only” or VOP_coding_type indicates B-VOP, MVPs is determined by considering the motion vectors of shape (MVs1, MVs2 and MVs3) only. The following sections explain the definition of MVs1, MVs2, MVs3, MV1, MV2 and MV3 of in more detail.

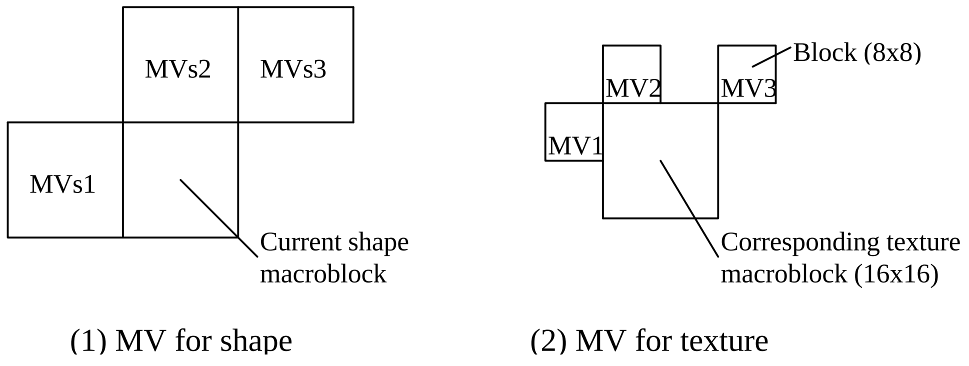


Figure 7-9 Candidates for MVPs

Defining candidate predictors from texture motion vectors:

One shape motion vector predictor MV_i (i =1,2,3) is defined for each block located around the current bab according to Figure 7-9. The definition only depends on the transparency of the reference MB. MVi is set to the corresponding block vector as long as it is in a non-transparent reference MB, otherwise, it is not defined. Note that if a reference MB is outside the current VOP or video packet, it is treated as a transparent MB.

Defining candidate predictors from shape motion vectors:

The candidate motion vector predictors MVs_i are defined by the shape motion vectors of neighbouring bab located according to Figure 7-9 (1). The MVs_i are defined according to Table 7-5.

Table 7-5 Definition of candidate shape motion vector predictors MVs1, MVs2, and MVs3 from shape motion vectors for P and B-VOPs. Note that interlaced modes are not included

Shape mode of reference MB	MVs_i for each reference shape block-***i*** (a shape block is 16x16)
MVDs == 0 or MVDs !=0 bab_type 0, 1, 5,6	The retrieved shape motion vector of the said reference MB is defined as MVs_i . Note that MVs_i is defined, and hence valid, even if the reconstructed shape block is transparent.
all_0, bab_type 2	MVs_i is undefined
all=255, bab_type 3	MVs_i is undefined
Intra, bab_type 4	MVs_i is undefined

If the reference MB is outside of the current video packet, MV_i and MVs_i are undefined.

7.4.2.4 Motion compensation

For inter mode babs (bab_type = 0,1,5 or 6), motion compensation is carried out by simple MV displacement according to the MVs.

Specifically, when bab_type is equal to 0 or 1 i.e. for the no-update modes, a displaced block of 16x16 pixels is copied from the binary alpha map of the previously decoded I or P VOP for which VOP_coded is not equal to ‘0’. When the bab_type is equal to 5 or 6 i.e. when interCAE decoding is required, then the pixels immediately bordering the displaced block (to the left, right, top and bottom) are also copied from the most recent valid reference VOP’s (as defined in Section 6.3.6**)** binary alpha map into a temporary shape block of 18x18 pixels size (see Figure 7-12). If the displaced position is outside the bounding box, then these pixels are assumed to be “transparent”.

7.4.2.5 Context based arithmetic decoding

Before decoding the binary_arithmetic_code field, border formation (see Section **7.4.2.5.2**) needs to be carried out. Then, if the scan_type field is equal to 0, the bordered to-be decoded bab and the eventual bordered motion compensated bab need to be transposed (as for matrix transposition). If change_conv_rate_disable is equal to 0, then conv_ratio is decoded to determine the size of the sub-sampled BAB, which is 16/conv_ratio by 16/conv_ratio pixels large. If change_conv_rate_disable is equal to 1, then the decoder assumes that the bab is not subsampled and thus the size is simply 16x16 pixels. Binary_arithmetic_code is then decoded by a context-based arithmetic decoder as follows. The arithmetic decoder is firstly initialised (see Section 7.4.3.3). The pixels of the sub-sampled bab are decoded in raster order. At each pixel,

1. A context number is computed based on a template, as described in Section **7.4.2.5.1**.

2. The context number is used to access the probability table (Table 11-28).

3. Using the accessed probability value, the next bits of binary_arithmetic_code are decoded by the arithmetic decoder to give the decoded pixel value.

When all pixels in sub-sampled BAB have been decoded, the arithmetic decoder is terminated (see Section 7.4.3.6).

If the scan_type field is equal to 0, the decoded bab is transposed. Then up-sampling is carried out if conv_ratio is different from 1, as described in Section **7.4.2.5.3**. Then the decoded bab is copied into the decoded shape map.

7.4.2.5.1 Context computation

For INTRA coded BABs, a 10 bit context $C = \sum_k c_k \cdot 2^k$ is built for each pixel as illustrated in Figure 7-10 (a), where c_k==0 for transparent pixels and c_k==1 for opaque pixels.

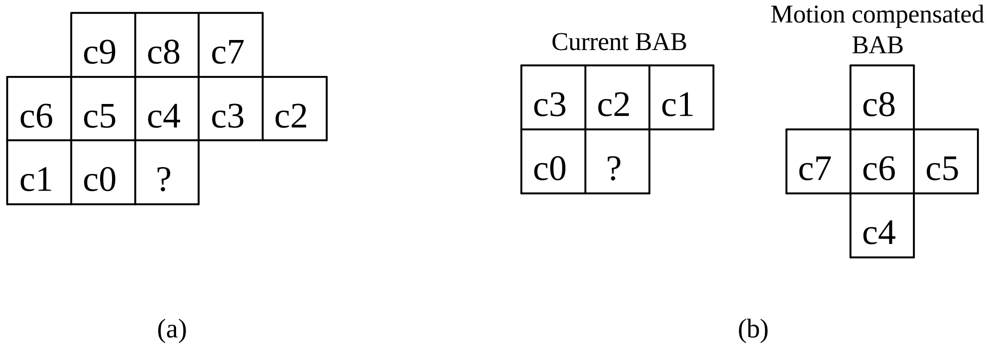


Figure 7-10 (a) The INTRA template (b) The INTER template where c6 is aligned with the pixel to be decoded. The pixel to be decoded is marked with '?'.

For INTER coded BABs, temporal redundancy is exploited by using pixels from the bordered motion compensated BAB (depicted in Figure 7-12) to make up part of the context. Specifically, a 9 bit context $C = \sum_k c_k \cdot 2^k$ is built as illustrated in Figure 7-10 (b).

There are some special cases to note.

- When building contexts, any pixels outside the bounding box of the current VOP to the left and above are assumed to be zero (transparent).
- When building contexts, any pixels outside the space of the current video packet to the left and above are assumed to be zero (transparent).
- The template may cover pixels from BABs which are unknown at decoding time. Unknown pixels are defined as area U in Figure 7-11. The values of these unknown pixels are defined by the following procedure:
    - When constructing the INTRA context, the following steps are taken in the sequence
        1. if (c7 is unknown) c7=c8,
        2. if (c3 is unknown) c3=c4,
        3. if (c2 is unknown) c2=c3.
    - When constructing the INTER context, the following conditional assignment is performed.

      if (c1 is unknown) c1=c2

7.4.2.5.2 Border formation

When decoding a BAB, pixels from neighbouring BABs can be used to make up the context. For both the INTRA and INTER cases, a 2 pixel wide border about the current BAB is used where pixels values are known, as depicted in Figure 7-11.

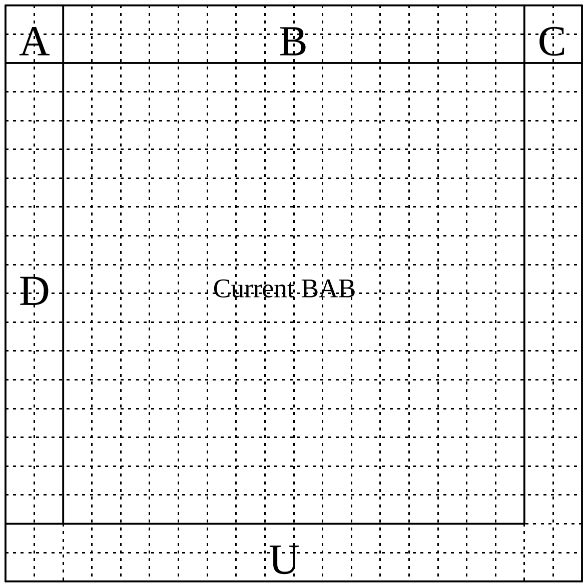


Figure 7-11 Bordered BAB. A: TOP_LEFT_BORDER. B: TOP_BORDER. C: TOP_RIGHT_BORDER. D: LEFT_BORDER. U: pixels which are unknown when decoding the current BAB.

If the value of conv_ratio is not equal to 1, a sub-sampling procedure is further applied to the BAB borders for both the current BAB and the motion compensated BAB.

The border of the current BAB is partitioned into 4:

- TOP_LEFT_BORDER, which contains pixels from the BAB located to the upper-left of the current BAB and which consists of 2 lines of 2 pixels
- TOP_BORDER, which contains pixels from the BAB located above the current BAB and which consists of 2 lines of 16 pixels
- TOP_RIGHT_BORDER, which contains pixels from the BAB located to the upper-right of the current BAB and which consists of 2 lines of 2 pixels
- LEFT_BORDER, which contains pixels from the BAB located to the left of the current BAB and which consists of 2 columns of 16 pixels

The TOP_LEFT_BORDER and TOP_RIGHT_BORDER are not sub-sampled, and kept as they are. The TOP_BORDER and LEFT_BORDER are sub-sampled such as to obtain 2 lines of 16/conv_ratio pixels and 2 columns of 16/conv_ratio pixels, respectively.

The sub-sampling procedure is performed on a line-basis for TOP_BORDER, and a column-basis for LEFT_BORDER. For each line (respectively column), the following algorithm is applied: the line (respectively column) is split into groups of conv_ratio pixels. For each group of pixels, one pixel is associated in the sub-sampled border. The value of the pixel in the sub-sampled border is OPAQUE if half or more pixels are OPAQUE in the corresponding group. Otherwise the pixel is TRANSPARENT.

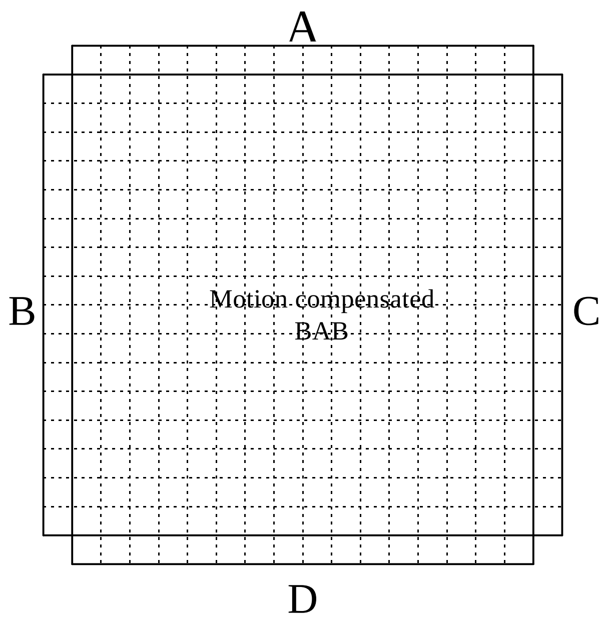


Figure 7-12 Bordered motion compensated BAB. A: TOP_BORDER. B: LEFT_BORDER. C: RIGHT_BORDER. D: BOTTOM_BORDER.

In the case of a motion compensated BAB, the border is also partitioned into 4, as shown Figure 7-12:

- TOP_BORDER, which consists of a line of 16 pixels
- LEFT_BORDER, which consists of a column of 16 pixels
- RIGHT_BORDER, which consists of a column of 16 pixels
- BOTTOM_BORDER, which consists of a line of 16 pixels

The very same sub-sampling process as described above is applied to each of these borders.

7.4.2.5.3 Upsampling

When conv_ratio is different from 1, up-sampling is carried out for the BAB. This is illustrated in Figure 7.13 where “O” in this figure is the coded pixel and “X” is the interpolated pixel. To compute the value of the interpolated pixel, a filter context from the neighboring pixels is first calculated. For the pixel value calculation, the value of “0” is used for a transparent pixel, and “1” for an opaque pixel. The values of the interpolated pixels (Pi, i=1,2,3,4, as shown in Figure 7-14) can then be determined by the following equation:

P1 : if(4*A + 2*(B+C+D) + (E+F+G+H+I+J+K+L) > Th[Cf]) then "1" else "0"

P2 : if(4*B + 2*(A+C+D) + (E+F+G+H+I+J+K+L) > Th[Cf]) then "1" else "0"

P3 : if(4*C + 2*(B+A+D) + (E+F+G+H+I+J+K+L) > Th[Cf]) then "1" else "0"

P4 : if(4*D + 2*(B+C+A) + (E+F+G+H+I+J+K+L) > Th[Cf]) then "1" else "0"

The 8-bit filter context, Cf, is calculated as follows:

$$C_f = \sum_k c_k \cdot 2^k$$

Based on the calculated Cf, the threshold value (Th[Cf]) can be obtained from the look-up table as follows:

```
Th[256] = {
  3, 6, 6, 7, 4, 7, 7, 8, 6, 7, 5, 8, 7, 8, 8, 9,
  6, 5, 5, 8, 5, 6, 8, 9, 7, 6, 8, 9, 8, 7, 9, 10,
  6, 7, 7, 8, 7, 8, 8, 9, 7, 10, 8, 9, 8, 9, 9, 10,
  7, 8, 6, 9, 6, 9, 9, 10, 8, 9, 9, 10, 11, 10, 10, 11,
  6, 9, 5, 8, 5, 6, 8, 9, 7, 10, 10, 9, 8, 7, 9, 10,
  7, 6, 8, 9, 8, 7, 7, 10, 8, 9, 9, 10, 9, 8, 10, 9,
  7, 8, 8, 9, 6, 9, 9, 10, 8, 9, 9, 10, 9, 10, 10, 9,
  8, 9, 11, 10, 7, 10, 10, 11, 9, 12, 10, 11, 10, 11, 11, 12,
  6, 7, 5, 8, 5, 6, 8, 9, 5, 6, 6, 9, 8, 9, 9, 10,
  5, 8, 8, 9, 6, 7, 9, 10, 6, 7, 9, 10, 9, 10, 10, 11,
  7, 8, 6, 9, 8, 9, 9, 10, 8, 7, 9, 10, 9, 10, 10, 11,
  8, 9, 7, 10, 9, 10, 8, 11, 9, 10, 10, 11, 10, 11, 9, 12,
  7, 8, 6, 9, 8, 9, 9, 10, 10, 9, 7, 10, 9, 10, 10, 11,
  8, 7, 7, 10, 7, 8, 8, 9, 9, 10, 10, 11, 10, 11, 11, 12,
  8, 9, 9, 10, 9, 10, 10, 9, 9, 10, 10, 11, 10, 11, 11, 12,
  9, 10, 10, 11, 10, 11, 11, 12, 10, 11, 11, 12, 11, 12, 12, 13 };
```

TOP_LEFT_BORDER, TOP_RIGHT_BORDER, sub-sampled TOP_BORDER and sub-sampled LEFT_BORDER described in the previous section are used. The other pixels outside the BAB are extended from the outermost pixels inside the BAB as shown in Figure 7-13.

In the case that conv_ratio=4, the interpolation is processed twice. The above mentioned borders of 4x4 BAB are used for the interpolation from 4x4 to 8x8, and top-border (resp. left-border) for the interpolation from 8x8 to 16x16 are up-sampled from the 4x4 BAB top-border (resp. left-border) by simple repetition.

When the BAB is on the left (and/or top) border of VOP, the borders outside VOP are set to zero value. The upsampling filter should be constrained to avoid using pixel values outside of the current video packet.

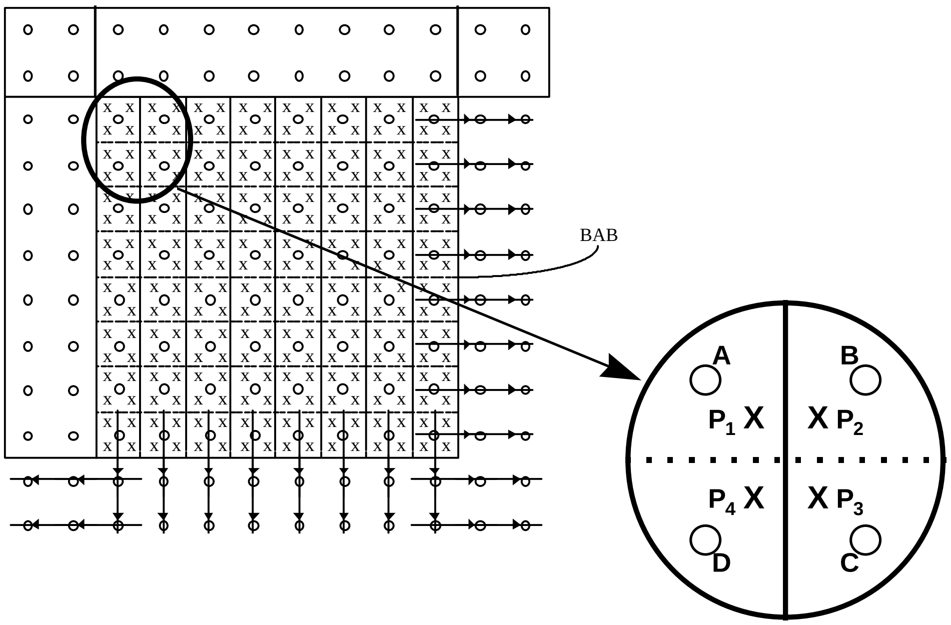


Figure 7-13 Upsampling

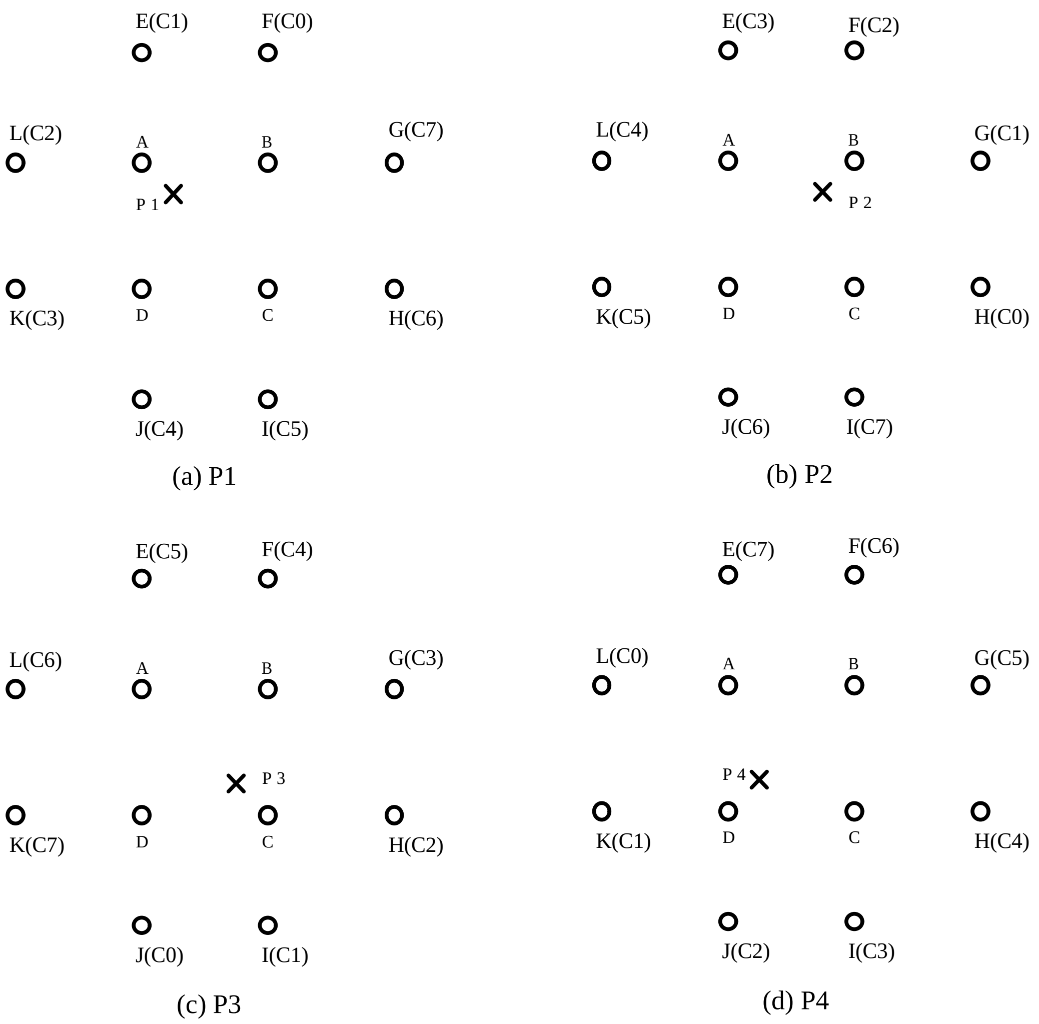


Figure 7-14 Interpolation filter and interpolation construction.

7.4.2.5.4 Down-sampling process in inter case

If bab_type is '5' or '6' (see Table 7.3), downsampling of the motion compensated bab is needed for calculating the 9 bit context in the case that conv_ratio is not 1. The motion compensated bab of size 16x16 pixels is down sampled to bab of size 16/conv_ratio by 16/conv_ratio pixels by the following rules:

- conv_ratio==2

  If the average of pixel values in 2 by 2 pixel block is equal to or greater than 127.5 the pixel value of the downsampled bab is set to 255 otherwise it is set to 0.
- conv_ratio==4

  If the average of pixel values in 4 by 4 pixel block is equal to or greater than 127.5 the pixel value of the downsampled bab is set to 255 otherwise it is set to 0.

7.4.3 Arithmetic decoding

Arithmetic decoding consists of four main steps:

- Removal of stuffed bits
- Initialization which is performed prior to the decoding of the first symbol
- Decoding of the symbol themselves. The decoding of each symbol may be followed by a re-normalization step.
- Termination which is performed after the decoding of the last symbol

7.4.3.1 Registers, symbols and constants

Several registers, symbols and constants are defined to describe the arithmetic decoder.

- HALF: 32-bit fixed point constant equal to ½(0x80000000)
- QUARTER: 32-bit fixed point constant equal to ¼ (0x40000000)
- L: 32-bit fixed point register. Contains the lower bound of the interval
- R: 32-bit fixed point register. Contains the range of the interval.
- V: 32-bit fixed point register. Contains the value of the arithmetic code. V is always larger than or equal to L and smaller than L+R.
- p0: 16-bit fixed point register. Probability of the '0' symbol.
- p1: 16-bit fixed point register. Probability of the '1' symbol.
- LPS: boolean. Value of the least probable symbol ('0' or '1').
- bit: boolean. Value of the decoded symbol.
- pLPS: 16-bit fixed point register. Probability of the LPS.
- rLPS: 32-bit fixed point register. Range corresponding to the LPS.

7.4.3.2 Bit stuffing

In order to avoid start code emulation, 1's are stuffed into the bitstream whenever there are too many successive 0's. If the first MAX_HEADING bits are 0's, then a 1 is transmitted after the MAX_HEADING-th 0. If more than MAX_MIDDLE 0's are sent successively a 1 is inserted after the MAX_MIDDLE-th 0. If the number of trailing 0's is larger than MAX_TRAILING, then a 1 is appended to the stream. The decoder shall properly skip these inserted 1's when reading data into the V register (see Section 7.4.3.3 and 7.4.3.5).

MAX_HEADING equals 3, MAX_MIDDLE equals 10, and MAX_TRAILIING equals 2.

7.4.3.3 Initialization

The lower bound L is set to 0, the rangeR to HALF-0x1 (0x7fffffff) and the first 31 bits are read in register V.

7.4.3.4 Decoding a symbol

When decoding a symbol, the probability p0 of the ‘0’ symbol is provided according to the context computed in Section **7.4.2.5.1** and using Table 11-32. p0 uses a 16-bit fixed-point number representation. Since the decoder is binary, the probability of the ‘1’ symbol is defined to be 1 minus the probability of the ‘0’ symbol, i.e. p1 = 1-p0.

The least probable symbol LPS is defined as the symbol with the lowest probability. If both probabilities are equal to ½(0x8000), the ‘0’ symbol is considered to be the least probable.

The range rLPS associated with the LPS may simply be computed as R*pLPS: The 16 most significant bits of register R are multiplied by the 16 bits of pLPS to obtain the 32 bit rLPS number.

The interval [L,L+R) is split into two intervals [L,L+R-rLPS) and [L+R-rLPS,L+R). If V is in the latter interval then the decoded symbol is equal to LPS. Otherwise the decoded symbol is the opposite of LPS. The interval [L,L+R) is then reduced to the sub-interval in which V lies.

After the new interval has been computed, the new range R might be smaller than QUARTER. If so, re-normalization is carried out, as described below.

7.4.3.5 Re-normalization

As long as R is smaller than QUARTER, re-normalization is performed.

- If the interval [L,L+R) is within [0,HALF), the interval is scaled to [2L,2L+2R). V is scaled to 2V.
- If the interval [L,L+R) is within [HALF,1) the interval is scaled to [2(L-HALF),2(L-HALF)+2R). V is scaled to 2(V-HALF).
- Otherwise the interval is scaled to [2(L-QUARTER),2(L-QUARTER)+2R). V is scaled to 2(V-QUARTER).

After each scaling, a bit is read and copied into the least significant bit of register V.

7.4.3.6 Termination

After the last symbol has been decoded, additional bits need to be “consumed”. They were introduced by the encoder to guarantee decodability.

In general 3 further bits need to be read. However, in some cases, only two bits need to be read. These cases are defined by:

- if the current interval covers entirely [QUARTER,HALF)
- if the current interval covers entirely [HALF, 3QUARTER)

After these additional bits have been read, 32 bits shall be “unread”, i.e. put the content of register V back into the bit buffer.

7.4.3.7 Software

The example software for arithmetic decoding for binary shape decoding is included in Annex B.

7.4.3.8 7.4.4.6 Method to be used when blending with greyscale alpha signal

The following explains the blending method to be applied to the video object in the compositor, which is controlled by the **composition_method** flag and the **linear_composition** flag. The **linear_composition** flag is informative only, and the decoder may ignore it and proceed as if it had the value 0. However, it is normative that the **composition_method** flag be acted upon.

The descriptions below show the processing taking place in YUV space; note that the processing can of course be implemented in RGB space to obtain equivalent results.

composition_method=0 (cross-fading)

If layer N, with an n-bit alpha signal, is overlaid over layer M to generate a new layer P, the composited Y, U, V and alpha values are:

$$\text{Pyuv} = ((2^n-1 - \text{Nalpha}) * \text{Myuv} + (\text{Nalpha} * \text{Nyuv})) / (2^n-1)$$

$$\text{Palpha} = (2^n-1)$$

composition_method=1 (Additive mixing)

If layer N, with an n-bit alpha signal, is overlaid over layer M to generate a new layer P, the composited Y, U, V and alpha values are:

$$\text{Pyuv} = \begin{cases} \text{Myuv} & \text{..... Nalpha} = 0 \\ (\text{Myuv} - \text{BLACK}) - ((\text{Myuv} - \text{BLACK}) * \text{Nalpha}) / (2^n-1) + \text{Nyuv} & \text{..... Nalpha} > 0 \end{cases}$$

(this is equivalent to Pyuv = Myuv*(1-alpha) + Nyuv, taking account of black level and the fact that the video decoder does not produce an output in areas where alpha=0)

$$\text{Palpha} = \text{Nalpha} + \text{Malpha} - (\text{Nalpha*Malpha}) / (2^n-1)$$

where

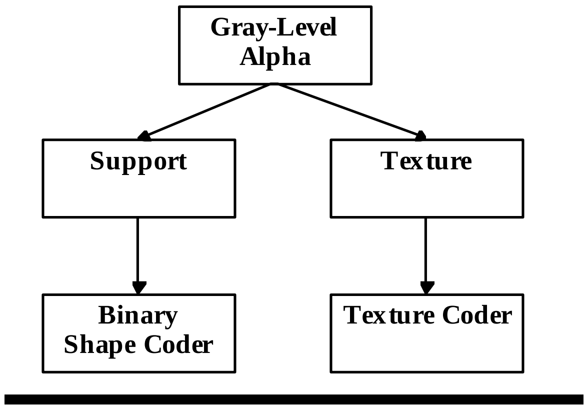
BLACK is the common black value of foreground and background objects.

NOTE The compositor must convert foreground and background objects to the same black value and signal range before composition. The black level of each video object is specified by the **video_range** bit in the **video_signal_type** field, or by the default value if the field is not present. (The RGB values of synthetic objects are specified in a range from 0 to 1, as described in ISO/IEC 14496-1).

- **linear_composition = 0:** The compositing process is carried out using the video signal in the format from which it is produced by the video decoder, that is, without converting to linear signals. Note that because video signals are usually non-linear ("gamma-corrected"), the composition will be approximate.
- **linear_composition = 1:** The compositing process is carried out using linear signals, so the output of the video decoder is converted to linear if it was originally in a non-linear form, as specified by the **video_signal_type** field. Note that the alpha signal is always linear, and therefore requires no conversion.

7.4.4 Grayscale Shape Decoding

Grayscale alpha plane decoding is achieved by the separate decoding of a support region and the values of the alpha channel. The support function is transmitted by using the binary shape as described above. The alpha values are transmitted as texture data with arbitrary shape, using almost the same coding method as is used for the luminance texture channel.



All samples which are indicated to be transparent by the binary shape data, must be set to zero in the decoded grayscale alpha plane. Within the VOP, alpha samples have the values produced by the grayscale alpha decoding process. Decoding of binary shape information is not dependent on the decoding of grayscale alpha. The alpha values are decoded into 16x16 macroblocks in the same way as the luminance channel (see sections 7.3 and 7.5). The 16x16 blocks of alpha values are referred to as alpha macroblocks hereafter. The data for each alpha macroblock is present in the bitstream immediately following the texture data for the corresponding texture macroblock. Any aspect of alpha decoding that is not covered in this document should be assumed to be the same as for the decoding of luminance.

7.4.4.1 Grayscale Alpha COD Modes

When decoding grayscale alpha macroblocks, CODA is first encountered and indicates the coding status for alpha. It is important to understand that the macroblock syntax elements for alpha are still present in the bitstream for P or B macroblocks even if the texture syntax elements indicate “not-coded” (not_coded=’1’). In this respect, the decoding of the alpha and texture data are independent. The only exception is for BVOPs when the colocated PVOP texture macroblock is skipped. In this case, no syntax is transmitted for texture or grayscale alpha, as both types of macroblock are skipped.

For macroblocks which are completely transparent (indicated by the binary shape coding), no alpha syntax elements are present and the grayscale alpha samples must all be set to zero (transparent). If CODA=”all opaque” (I, P or B macroblocks) or CODA=”not coded” (P or B macroblocks) then no more alpha data is present. Otherwise, other alpha syntax elements follow, including the coded block pattern (CBPA), followed by alpha texture data for those 8x8 blocks which are coded and non-transparent, as is the case for regular luminance macroblock texture data.

When CODA=”all opaque”, the corresponding decoded alpha macroblock is filled with a constant value of 255. This value will be called AlphaOpaqueValue.

7.4.4.2 Alpha Plane Scale Factor

For both binary and grayscale shape, the VOP header syntax element “VOP_constant_alpha” can be used to scale the alpha plane. If this bit is equal to ‘1’, then each pixel in the decoded VOP is scaled before output, using VOP_constant_alpha_value. The scaling formula is:

scaled_pixel = (original_pixel * (VOP_constant_alpha_value + 1)) / 256

Scaling is applied at the output of the decoder, such that the decoded original values, not the scaled values are used as the source for motion compensation.

7.4.4.3 Gray Scale Quantiser

When no_gray_quant_update is equal to “1”, the grayscale alpha quantiser is fixed for all macroblocks to the value indicated by VOP_alpha_quant. Otherwise, the grayscale quantiser is reset at each new macroblock to a value that depends on the current texture quantiser (after any update by dquant). The relation is:

current_alpha_quant = (current_texture_quant * VOP_alpha_quant) / VOP_quant

The resulting value of current_alpha_quant must then be clipped so that it never becomes less than 1.

7.4.4.4 Intra Macroblocks

When the texture mb_type indicates an intra macroblock in IVOPs or PVOPs, the grayscale alpha data is also decoded using intra mode.

The intra dc value is decoded in the same way as for luminance, using the same non-linear transform to convert from alpha_quant to DCScalarA. However, intra_dc_vlc_thr is not used for alpha, and therefore AC coeffiecient VLCs are never used to code the differential intra dc coefficient.

DC prediction is used in the same way as for luminance. However, when CODA_I indicates that a macroblock is all opaque, a synthetic intra dc value is created for each block in the current macroblock so that adjacent macroblocks can correctly obtain intra dc prediction values. The synthetic intra dc value is given as:

BlockIntraDC = (((AlphaOpaqueValue * 8) + (DcScalerA>>1)) / DcScalerA) * DcScalerA

AlphaOpaqueValue is described in Section 7.4.4.1.

The intra CBPA VLC makes use of the *inter* CBPY VLC table, but the intra alpha block DCT coefficients are decoded in the same manner as with luminance intra macroblocks.

7.4.4.5 Inter Macroblocks and Motion Compensation

Motion compensation is carried out for PVOPs and BVOPs, using the 8x8 or 16x16 luminance motion vectors, in the same way as for luminance data, except that regular motion compensation is used instead of OBMC. Forward, backward, bidirectional and direct mode motion compensation are used for BVOPs. Where the luminance motion vectors are not present because the texture macroblock is skipped, the exact same style of non-coded motion compensation used for luminance is applied to the alpha data (but without OBMC). Note that this does not imply that the alpha macroblock is skipped, because an error signal to update the resulting motion compensated alpha macroblock may still be present if indicated by CODA_PB. When the colocated PVOP texture macroblock is skipped for BVOPs, then the alpha macroblock is assumed to be skipped with no syntax transmitted.

CBPA and the alpha inter DCT coefficients are decoded in the same way as with luminance CBPY and inter DCT cofficients

7.5 Motion compensation decoding

In order to perform motion compensated prediction on a per VOP basis, a special padding technique, i.e. the macroblock-based repetitive padding, is applied for the reference VOP. The details of these techniques are described in the following sections.

Since a VOP may have arbitrary shape, and this shape can change from one instance to another, conventions are necessary to ensure the consistency of the motion compensation process.

The absolute (frame) coordinate system is used for referencing every VOP. At every given instance, a bounding rectangle that includes the shape of that VOP, as described in section 7.4, is defined. The left and top corner, in the absolute coordinates, of the bounding box is decoded from VOP spatial reference. Thus, the motion vector for a particular feature inside a VOP, e.g. a macroblock, refers to the displacement of the feature in absolute coordinates. No alignment of VOP bounding boxes at different time instances is performed.

In addition to the above motion compensation processing, two additional processes are supported, namely, unrestricted motion compensation and four MV motion compensation. Also, an optimal, an overlapped motion compensation can be performed. Note that in all three modes, macroblock-based padding of the arbitrarily shaped reference VOP is performed for motion compensation.

7.5.1 Padding process

The padding process defines the values of luminance and chrominance samples outside the VOP for prediction of arbitrarily shaped objects. Figure 7-15 shows a simplified diagram of this process.

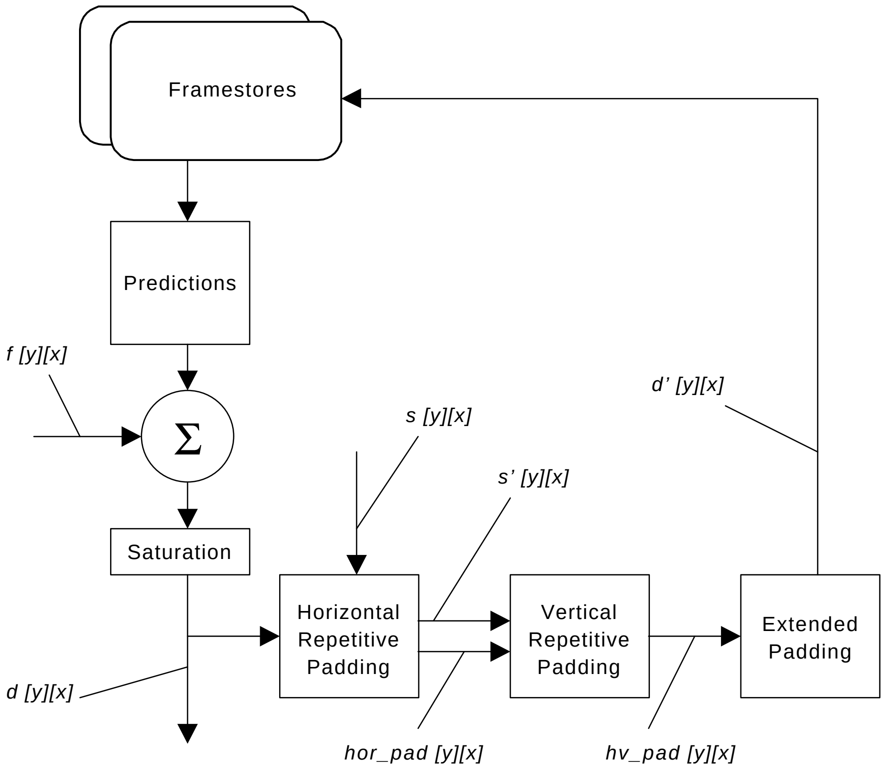


Figure 7-15 Simplified padding process

A decoded macroblock *d[y][x]* is padded by referring to the corresponding decoded shape block *s[y][x]*. The luminance component is padded per 16 x 16 samples, while the chrominance components are padded per 8 x 8 samples. A macroblock that lies on the VOP boundary (hereafter referred to as a boundary macroblock) is padded by replicating the boundary samples of the VOP towards the exterior. This process is divided into horizontal repetitive padding and vertical repetitive padding. The remaining macroblocks that are completely outside the VOP (hereafter referred to as exterior macroblocks) are filled by extended padding.

Note **-** The padding process is applied to all macroblocks inside the bounding rectangle of a VOP. The bounding rectangle of the luminance component is defined by VOP_width and VOP_height extended to multiple of 16, while that of the chrominance components is defined by (VOP_width>>1) and (VOP_height>>1) extended to multiple of 8.

7.5.1.1 Horizontal repetitive padding

Each sample at the boundary of a VOP is replicated horizontally to the left and/or right direction in order to fill the transparent region outside the VOP of a boundary macroblock. If there are two boundary sample values for filling a sample outside of a VOP, the two boundary samples are averaged (//2).

hor_pad[y][x] is generated by any process equivalent to the following example. For every line with at least one shape sample *s[y][x]* == 1(inside the VOP) :

```
for (x=0; x<N; x++) {
    if (s[y][x] == 1) { hor_pad[y][x] = d[y][x]; s'[y][x] = 1; }
    else {
        if ( s[y][x'] == 1 && s[y][x"] == 1)  {
            hor_pad[y][x] = (d[y][x']+ d[y][x"])//2;
            s'[y][x] = 1;
        } else if ( s[y][x'] == 1 ) {
            hor_pad[y][x] = d[y][x']; s'[y][x] = 1;
        } else if ( s[y][x"] == 1 ) {
            hor_pad[y][x] = d[y][x"]; s'[y][x] = 1;
        }
    }
}
```

where *x'* is the location of the nearest valid sample (*s[y][x']* == 1) at the VOP boundary to the left of the current location *x*, *x"* is the location of the nearest boundary sample to the right, and *N* is the number of samples of a line. *s'[y][x]* is initialized to 0.

7.5.1.2 Vertical repetitive padding

The remaining unfilled transparent horizontal samples (where *s'[y][x]* == 0) from Section 7.5.1.1 are padded by a similar process as the horizontal repetitive padding but in the *vertical* direction. The samples already filled in Section 7.5.1.1 are treated as if they were inside the VOP for the purpose of this vertical pass.

hv_pad[y][x] is generated by any process equivalent to the following example. For every column of *hor_pad[y][x]* :

```
for (y=0; y<M; y++) {
    if (s'[y][x] == 1)
        hv_pad[y][x] =hor_pad[y][x];
    else {
        if ( s'[y'][x] == 1 && s'[y"][x] == 1 )
            hv_pad[y][x]          =          (hor_pad[y'][x]          +
hor_pad[y"][x])//2;
        else if ( s'[y'][x] == 1 )
            hv_pad[y][x] = hor_pad[y'][x];
        else if (s'[y"][x] == 1 )
            hv_pad[y][x] = hor_pad[y"][x];
    }
}
```

where *y'* is the location of the nearest valid sample (*s'[y'][x]* == 1) above the current location *y* at the boundary of *hv_pad*, *y"* is the location of the nearest boundary sample below *y*, and *M* is the number of samples of a column.

7.5.1.3 Extended padding

Exterior macroblocks immediately next to boundary macroblocks are filled by replicating the samples at the border of the boundary macroblocks. Note that the boundary macroblocks have been completely padded in Section 7.5.1.1 and Section 7.5.1.2. If an exterior macroblock is next to more than one boundary macroblocks, one of the macroblocks is chosen, according to the following convention, for reference.

The boundary macroblocks surrounding an exterior macroblock are numbered in priority according to Figure 7-16. The exterior macroblock is then padded by replicating upwards, downwards, leftwards, or rightwards the row of samples from the horizontal or vertical border of the boundary macroblock having the largest priority number.

The remaining exterior macroblocks (not located next to any boundary macroblocks) are filled with $2^{bits\_pixel-1}$. For 8-bit luminance component and associated chrominance this implies filling with 128.

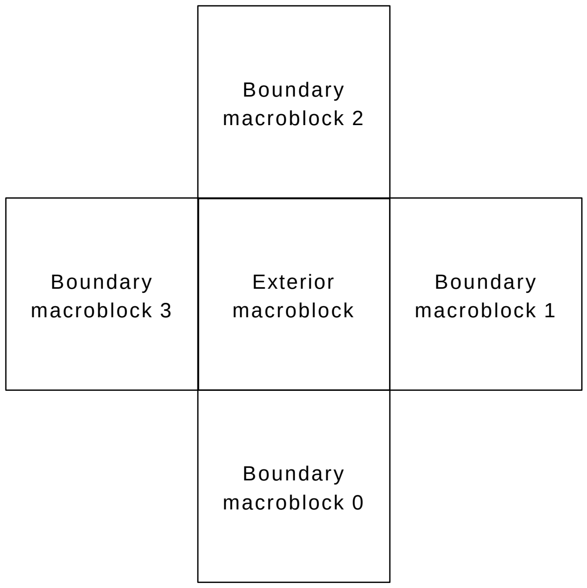


Figure 7-16 Priority of boundary macroblocks surrounding an exterior macroblock

7.5.1.4 Padding for chrominance components

Chrominance components are padded according to clauses 7.5.1.1 through 7.5.1.3 for each 8 x 8 block. The padding is performed by referring to a shape block generated by decimating the shape block of the corresponding luminance component. The similar rule is applied to interlaced video based on field to enhance subjective quality of display in 4:2:0 format. For each 2 x 2 adjacent luminance shape samples of the same fields, the corresponding chrominance shape sample is set to 1 if any of the four luminance samples are 1. Otherwise the chrominance shape sample is set to 0. For each 2 x 2 adjacent luminance shape samples, the corresponding chrominance shape sample is set to 1 if any of the four luminance shape samples are 1. Otherwise the chrominance shape sample is set to 0.

7.5.1.5 Padding of interlaced macroblocks

Macroblocks of interlaced VOP (interlaced = 1) are padded according to clauses 7.5.1.1 through 7.5.1.3. The vertical padding of the luminance component, however, is performed for each field independently. A sample outside of a VOP is therefore filled with the value of the nearest boundary sample of the same field. Completely transparent blocks are padded with 128.

7.5.1.6 Vector padding technique

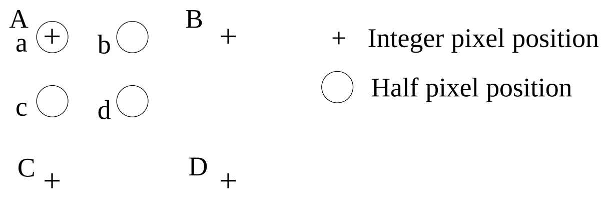
The vector padding technique is applied to generate the vectors for the transparent blocks within a non-transparent macroblock, for an INTRA-coded macroblock and for a skipped macroblock. It works in a similar way as the horizontal followed by the vertical repetitive padding, and can be simply regarded as the repetitive padding performed on a 2x2 block except that the padded values are two dimensional vectors. A macroblock has four 8x8 luminance blocks, let {MVx[i], MVy[i], i=0,1,2,3} and {Transp[i], i=0,1,2,3} be the vectors and the transparencies of the four 8x8 blocks, respectively, the vector padding is any process numerically equivalent to:

```
if (the macroblock is INTRA-coded, skipped ) {
    MVx[0] = MVx[1] = MVx[2] = MVx[3] = 0
    MVy[0] = MVy[1] = MVy[2] = MVy[3] = 0
} else {
    if(Transp[0] == TRANSPARENT) {
        MVx[0]=(Transp[1] != TRANSPARENT) ? MVx[1] :((Transp[2]!=TRANSPARENT) ?
            MVx[2]:MVx[3]));
        MVy[0]=(Transp[1] != TRANSPARENT) ? MVy[1]:((Transp[2]!=TRANSPARENT) ?
            MVy[2]:MVy[3]));
    }
if(Transp[1] == TRANSPARENT) {
    MVx[1]=(Transp[0] != TRANSPARENT) ? MVx[0] :((Transp[3]!=TRANSPARENT) ?
        MVx[3]:MVx[2]));
    MVy[1]=(Transp[0] != TRANSPARENT) ? MVy[0]:((Transp[3]!=TRANSPARENT) ?
        MVy[3]:MVy[2]));
}
if(Transp[2] == TRANSPARENT) {
    MVx[2]=(Transp[3] != TRANSPARENT) ? MVx[3] :((Transp[0]!=TRANSPARENT) ?
        MVx[0]:MVx[1]));
    MVy[2]=(Transp[3] != TRANSPARENT) ? MVy[3]:((Transp[0]!=TRANSPARENT) ?
        MVy[0]:MVy[1]));
}
if(Transp[3] == TRANSPARENT) {
    MVx[3]=(Transp[2] != TRANSPARENT) ? MVx[2] :((Transp[1]!=TRANSPARENT) ?
        MVx[1]:MVx[0]));
    MVy[3]=(Transp[2] !=TRANSPARENT) ? MVy[2]:((Transp[1]!=TRANSPARENT) ?
        MVy[1]:MVy[0]));
    }
}
```

Vector padding is only used in I- and P-VOPs, it is applied on a macroblock directly after it is decoded. The block vectors after padding are used in the P-VOP vector decoding and binary shape decoding, and in the B-VOP direct mode decoding.

7.5.2 Half sample interpolation

Pixel value interpolation for block matching when rounding is used corresponds to bilinear interpolation as depicted in Figure 7-17. The value of rounding_control is defined using the VOP_rounding_type bit in the VOP header (see clause 6.3.6). Note that the samples outside the padded region cannot be used for interpolation.



a = A,
b = (A + B + 1 - rounding_control) / 2
c = (A + C + 1 - rounding_control) / 2,
d = (A + B + C + D + 2 - rounding_control) / 4

Figure 7-17 Interpolation scheme for half sample search.

7.5.3 General motion vector decoding process

To decode a motion vector (MVx, MVy), the differential motion vector (MVDx, MVDy) is extracted from the bitstream by using the variable length decoding. Then it is added to a motion vector predictor (Px, Py) component wise to form the final motion vector. The general motion vector decoding process is any process that is equivalent to the following one. All calculations are carried out in halfpel units in the following. This process is generic in the sense that it is valid for the motion vector decoding in interlaced/progressive P- and B-VOPs except that the generation of the predictor (Px, Py) may be different.

```
r_size = VOP_fcode - 1
f = 1 << r_size
high = ( 32 * f ) -  1;
low = ( (-32) * f );
range = ( 64 * f );

if ( (f  == 1) || (horizontal_mv_data == 0) )
    MVDx = horizontal_mv_data;
else {
    MVDx = ( ( Abs(horizontal_mv_data) - 1 ) * f ) + horizontal_mv_residual + 1;
    if (horizontal_mv_data < 0)
        MVDx = - MVDx;
}

if ( (f  == 1) || (vertical_mv_data == 0) )
    MVDy = vertical_mv_data;
else {
    MVDy = ( ( Abs(vertical_mv_data) - 1 ) * f ) + vertical_mv_residual + 1;
    if (vertical_mv_data < 0)
        MVDy = - MVDy;
}


MVx = Px + MVDx;
if ( MVx < low )
    MVx = MVx + range;
if (MVx > high)
    MVx = MVx - range;

MVy = Py + MVDy;
if ( MVy < low )
    MVy = MVy + range;
if (MVy > high)
    MVy = MVy - range;
```

The parameters in the bitstream shall be such that the components of the reconstructed differential motion vector, *MVDx* and *MVDy*, shall lie in the range [*low*:*high]*. In addition the components of the reconstructed motion vector, *MVx* and *MVy*, shall also lie in the range [*low* : *high*]. The allowed range [low : high] for the motion vectors depends on the parameter VOP_fcode; it is shown in Table 7-6.

The variables *r_size*, *f*, *MVDx*, *MVDy*, *high* , *low* and *range* are temporary variables that are not used in the remainder of this specification. The parameters horizontal_mv_data, vertical_mv_data, horizontal_mv_residual and vertical_mv_residual are parameters recovered from the bitstream.

The variable *VOP_fcode* refers either to the parameter VOP_fcode_forward or to the parameter VOP_fcode_backward which have been recovered from the bitstream, depending on the respective prediction mode. In the case of P-VOP prediction only forward prediciton applies. In the case of B-VOP prediction, forward as well as backward prediction may apply.

VOP_fcode_forward or VOP_fcode_backward	motion vector range in halfsample units [low:high]
1	[-32,31]
2	[-64,63]
3	[-128,127]
4	[-256,255]
5	[-512,511]
6	[-1024,1023]
7	[-2048,2047]

Table 7-6 Range for motion vectors

If the current macroblock is a field motion compensated macroblock, then the same prediction motion vector (Px, Py) is used for both field motion vectors. Because the vertical component of a field motion vector is integral, the vertical differential motion vector encoded in the bitstream is

$$MVy = MVDy_{field} + PY / 2$$

7.5.4 Unrestricted motion compensation

Motion vectors are allowed to point outside the decoded area of a reference VOP. For an arbitrary shape VOP, the decoded area refers to the area within the bounding box, padded as described in clause Section 7.5.1. A bounding box is defined by VOP_width and VOP_height extended to multiple of 16. When a sample referenced by a motion vector stays outside the decoded VOP area, an edge sample is used. This edge sample is retrieved by limiting the motion vector to the last full pel position inside the decoded VOP area. Limitation of a motion vector is performed on a sample basis and separately for each component of the motion vector.

The coordinates of a reference sample in the reference VOP, (yref, xref) is determined as follows :

$$xref = MIN (MAX (x+dx, vhmcsr), xdim+vhmcsr-1))$$

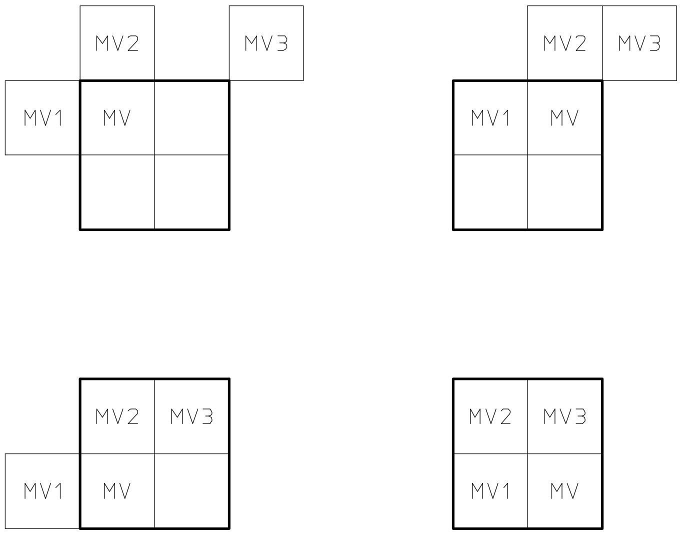
$$yref = MIN (MAX (y+dy, vvmcsr), ydim+vvmcsr-1))$$

where vhmcsr = VOP_horizontal_mc_spatial_reference, vvmcsr = VOP_vertical_mc_spatila_reference, (y, x) are the coordinates of a sample in the current VOP, (yref, xref) are the coordinates of a sample in the reference VOP, (dy, dx) is the motion vector, and (ydim, xdim) are the dimensions of the bounding box of the reference VOP. Note that for rectangular VOP, a reference VOP is defined by video_object_layer_width and video_object_layer_height. For an arbitrary shape VOP, a reference VOP of luminance is defined by VOP_width and VOP_height extended to multiple of 16, while that of chrominance is defined by (VOP_width>>1**)** and (VOP_height>>1) extended to multiple of 8.

7.5.5 Vector decoding processing and motion-compensation in progressive P-VOP

An inter-coded macroblock comprises either one motion vector for the complete macroblock or K (1< K<=4) motion vectors, one for each non-transparent 8x8 pel blocks forming the 16x16 pel macroblock, as is indicated by the MCBPC code.

For decoding a motion vector, the horizontal and vertical motion vector components are decoded differentially by using a prediction, which is formed by a median filtering of three vector candidate predictors (MV1, MV2, MV3) from the spatial neighbourhood macroblocks or blocks already decoded. The spatial position of candidate predictors for each block vector is depicted in Figure 7-18. In the case of only one motion vector present for the complete macroblock, the top-left case in Figure 7-18 is applied



.

Figure 7-18 Definition of the candidate predictors MV1, MV2 and MV3 for each of the luminance blocks in a macroblock

The following four decision rules are applied to obtain the value of the three candidate predictors:

1. If a candidate predictor MVi is in a transparent spatial neighbourhood macroblock or in a transparent block of the current macroblock it is not valid, otherwise, it is set to the corresponding block vector.
2. If one and only one candidate predictor is not valid, it is set to zero.
3. If two and only two candidate predictors are not valid, they are set to the third candidate predictor.
4. If all three candidate predictors are not valid, they are set to zero.

Note that any neighbourhood macroblock outside the current VOP or video packet is treated as transparent in the above sense. The median value of the three candidates for the same component is computed as predictor, denoted by Px and Py:

$$Px = Median(MV1x, MV2x, MV3x)$$
$$Py = Median(MV1y, MV2y, MV3y)$$

For instance, if MV1=(-2,3), MV2=(1,5) and MV3=(-1,7), then Px = -1 and Py = 5. The final motion vector is then obtained by using the general decoding process defined in the section 7.5.3.

If four vectors are used, each of the motion vectors is used for all pixels in one of the four luminance blocks in the macroblock. The numbering of the motion vectors is equivalent to the numbering of the four luminance blocks as given in Figure 6-5. Motion vector MVD_{CHR} for both chrominance blocks is derived by calculating the sum of the *K* luminance vectors, that corresponds to *K* 8x8 blocks that do not lie outside the VOP shape and dividing this sum by 2**K;* the component values of the resulting sixteenth/twelfth/eighth/fourth sample resolution vectors are modified towards the nearest half sample position as indicated below.

Table 7-7 Modification of sixteenth sample resolution chrominance vector components

sixteenth pixel position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	//16
resulting position	0	0	0	1	1	1	1	1	1	1	1	1	1	1	2	2	//2

Table 7-8 Modification of twelfth sample resolution chrominance vector components

twelfth pixel position	0	1	2	3	4	5	6	7	8	9	10	11	//12
resulting position	0	0	0	1	1	1	1	1	1	1	2	2	//2

Table 7-9 Modification of eighth sample resolution chrominance vector components

eighth pixel position	0	1	2	3	4	5	6	7	//8
resulting position	0	0	1	1	1	1	1	2	//2

Table 7-10 Modification of fourth sample resolution chrominance vector components

fourth pixel position	0	1	2	3	//4
resulting position	0	1	1	1	//2

Half sample values are found using bilinear interpolation as described in clause 7.5.2. The prediction for luminance is obtained by overlapped motion compensation as described in clause 7.5.6 if indicated by obmc_disable==0. The prediction for chrominance is obtained by applying the motion vector MVD_{CHR} to all pixels in the two chrominance blocks.

7.5.6 Overlapped motion compensation

This clause specifies the overlapped motion compensation process. This process is performed when the flag obmc_disable=0.

Each pixel in an 8*8 luminance prediction block is a weighted sum of three prediction values, divided by 8 (with rounding). In order to obtain the three prediction values, three motion vectors are used: the motion vector of the current luminance block, and two out of four "remote" vectors:

- the motion vector of the block at the left or right side of the current luminance block;
- the motion vector of the block above or below the current luminance block.

For each pixel, the remote motion vectors of the blocks at the two nearest block borders are used. This means that for the upper half of the block the motion vector corresponding to the block above the current block is used, while for the lower half of the block the motion vector corresponding to the block below the current block is used. Similarly, for the left half of the block the motion vector corresponding to the block at the left side of the current block is used, while for the right half of the block the motion vector corresponding to the block at the right side of the current block is used.

The creation of each pixel, $\overline{p}(i,j)$, in an 8*8 luminance prediction block is governed by the following equation:

$$\overline{\mathrm{p}}(\mathrm{i},\mathrm{j}) = (\mathrm{q}(\mathrm{i},\mathrm{j}) \times \mathrm{H}_0(\mathrm{i},\mathrm{j}) + \mathrm{r}(\mathrm{i},\mathrm{j}) \times \mathrm{H}_1(\mathrm{i},\mathrm{j}) + \mathrm{s}(\mathrm{i},\mathrm{j}) \times \mathrm{H}_2(\mathrm{i},\mathrm{j}) + 4)\,//8,$$

where $q(i,j)$, $r(i,j)$, and $s(i,j)$ are the pixels from the referenced picture as defined by

$$q(i,j) = p(i + MV_x^0, j + MV_y^0),$$
$$r(i,j) = p(i + MV_x^1, j + MV_y^1),$$
$$s(i,j) = p(i + MV_x^2, j + MV_y^2).$$

Here, (MV_x^0, MV_y^0) denotes the motion vector for the current block, (MV_x^1, MV_y^1) denotes the motion vector of the block either above or below, and (MV_x^2, MV_y^2) denotes the motion vector either to the left or right of the current block as defined above.

The matrices $H_0(i,j), H_1(i,j)$ and $H_2(i,j)$ are defined in Figure 7-19, Figure 7-20, and Figure 7-21, where (i,j) denotes the column and row, respectively, of the matrix.

If one of the surrounding blocks was not coded, the corresponding remote motion vector is set to zero. If one of the surrounding blocks was coded in intra mode, the corresponding remote motion vector is replaced by the motion vector for the current block. If the current block is at the border of the VOP and therefore a surrounding block is not present, the corresponding remote motion vector is replaced by the current motion vector. In addition, if the current block is at the bottom of the macroblock, the remote motion vector corresponding with an 8*8 luminance block in the macroblock below the current macroblock is replaced by the motion vector for the current block.

4	5	5	5	5	5	5	4
5	5	5	5	5	5	5	5
5	5	6	6	6	6	5	5
5	5	6	6	6	6	5	5
5	5	6	6	6	6	5	5
5	5	6	6	6	6	5	5
5	5	5	5	5	5	5	5
4	5	5	5	5	5	5	4

Figure 7-19 Weighting values, H_0, for prediction with motion vector of current luminance block

2	2	2	2	2	2	2	2
1	1	2	2	2	2	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	2	2	2	2	1	1
2	2	2	2	2	2	2	2

Figure 7-20 Weighting values, H1 , for prediction with motion vectors of the luminance blocks on top or bottom of current luminance block

2	1	1	1	1	1	1	2
2	2	1	1	1	1	2	2
2	2	1	1	1	1	2	2
2	2	1	1	1	1	2	2
2	2	1	1	1	1	2	2
2	2	1	1	1	1	2	2
2	2	1	1	1	1	2	2
2	1	1	1	1	1	1	2

Figure 7-21 Weighting values, H_2, for prediction with motion vectors of the luminance blocks to the left or right of current luminance block

7.5.7 Temporal prediction structure

1. A target P-VOP shall make reference for prediction to the most recently decoded I- or P-VOP. If the **VOP_coded** of the most recently decoded I- or P-VOP is ”0”, the target P-VOP shall make reference to a decoded I- or P-VOP which immediately precedes said most recently decoded I- or P-VOP, and whose **VOP_coded** is not zero.

2. A target B-VOP shall make reference for prediction to the most recently decoded forward and/or backward reference VOPs. The target B-VOP shall only make reference to said forward or backward reference VOPs whose **VOP_coded** is not zero. If the **VOP_coded** flags of both most recently decoded forward and backward reference VOPs are zero, the following rules applies.

   - for texture, the predictor of the target B-VOP shall be a gray macroblock of (Y, U, V) = ($2^{bits\_pixel-1}$,. $2^{bits\_pixel-1}$, $2^{bits\_pixel-1}$).
   - for binary alpha planes, the predictor shall be zero (transparent block)

Note that, binary alpha shape in B-VOP shall make reference for prediction to the most recently decoded forward reference VOP.

1. For arbitrarily shape objects, a decoded VOP whose VOP_coded is not zero but who's shape is completely transparent (shape of all 0) shall be padded by ($2^{n\text{-}bits\text{-}1}$, $2^{n\text{-}bits\text{-}1}$, $2^{n\text{-}bits\text{-}1}$) for (Y, U, V).

The temporal prediction structure is depicted in Figure 7-22.

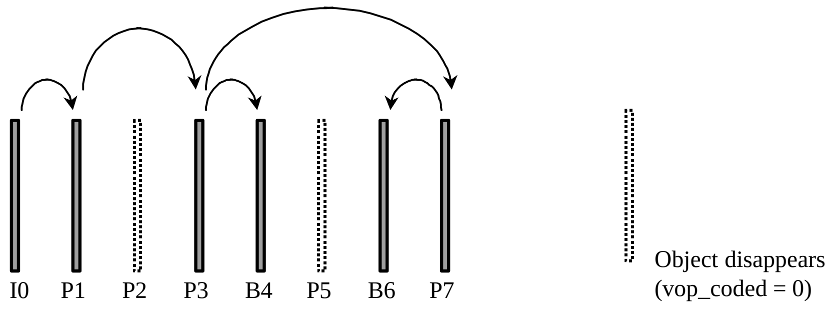


Figure 7-22 Temporal Prediction Structure.

7.5.8 Vector decoding process of non-scalable progressive B-VOPs

In B-VOPs there are three kinds of vectors, namely, 16x16 forward vector, 16x16 backward vector and the delta vector for the direct mode. The vectors are decoded with respect to the corresponding vector predictors. The basic decoding process of a differential vector is the exactly same as defined in P-VOPs except that for the delta vector of the direct mode the f_code is always one. The vector is then reconstructed by adding the decoded differential vector to the corresponding vector predictor. The vector predictor for the delta vector is always set to zero, while the forward and backward vectors have their own vector predictors, which are reset to zero only at the beginning of each macroblock row. The vector predictors are updated in the following three cases:

- after decoding a macroblock of forward mode only the forward predictor is set to the decoded forward vector
- after decoding a macroblock of backward mode only the backward predictor is set to the decoded backward vector.
- after decoding a macroblock of bi-directional mode both the forward and backward predictors are updated separately with the decoded vectors of the same type (forward/backward).

7.5.9 Motion compensation in non-scalable progressive B-VOPs

In B-VOPs the overlapped motion compensation (OBMC) is not employed. The motion-compensated prediction of B-macroblock is generated by using the decoded vectors and taking reference to the padded forward/backward reference VOPs as defined below. Arbitrarily shaped reference VOPs shall be padded accordingly.

7.5.9.1 Basic motion compensation procedure

All of the MPEG-4 motion compensation techniques are based on the formation of a prediction block, pred[i][j] of dimension (width, height), from a reference image, ref[x][y]. The coordinates of the current block (or macroblock) in the reference VOP is (x,y), the motion half-pel resolution motion vector is (dx_halfpel, dy_halfpel). The pseudo-code for this procedure is given below.

The component_width() and component_height() function give the coded VOP dimensions for the current component. For luminance, component_width() is video_object_layer_width for a rectangular VOP or VOP_width otherwise rounded up to the next multiple of 16. The luminance component_height() is defined similarly. The chrominance dimensions are one half of the corresponding luminance dimension.

```
clip_ref(ref, x, y)
{
    return(ref[MIN(MAX(x, 0), component_width(ref) - 1)]
              [MIN(MAX(y, 0), component_height(ref) - 1)]);
}

mc(pred,                       /* prediction block */
   ref,                        /* reference component */
   x, y,                       /* ref block coords for MV=(0, 0) */
   width, height,              /* reference block dimensions */
   dx_halfpel, dy_halfpel,     /* half-pel resolution motion vector */
   rounding,                   /* rounding control (0 or 1 ) */
   pred_y0,                    /* field offset in pred blk (0 or 1) */
   ref_y0,                     /* field offset in ref blk (0 or 1) */
   y_incr)                     /* vertical increment (1 or 2) */
{
    dx = dx_halfpel >> 1;
    dy = y_incr * (dy_halfpel >> y_incr);
    if (dy_halfpel & y_incr) {
        if (dx_halfpel & 1) {
            for (iy = 0; iy < height; iy += y_incr) {
                for (ix = 0; ix < width; ix++) {
                    x_ref = x + dx + ix;
                    y_ref = y + dy + iy + ref_y0;
                    pred[dx][dy + pred_y0] =
                        (clip_ref(ref, x_ref + 0, y_ref + 0) +
                         clip_ref(ref, x_ref + 1, y_ref + 0) +
                         clip_ref(ref, x_ref + 0, y_ref + y_incr) +
                         clip_ref(ref, x_ref + 1, y_ref + y_incr) +
                         2 - rounding) >> 2;
                }
            }
        } else {
            for (iy = 0; iy < height; iy += y_incr) {
                for (ix = 0; ix < width; ix++) {
                    x_ref = x + dx + ix;
                    y_ref = y + dy + iy + ref_y0;
                    pred[dx][dy + pred_y0] =
                        (clip_ref(ref, x_ref, y_ref + 0) +
                         clip_ref(ref, x_ref, y_ref + y_incr) +
                         1 - rounding) >> 1;
                }
            }
        }
    } else {
        if (dx_halfpel & 1) {
            for (iy = 0; iy < height; iy += y_incr) {
                for (ix = 0; ix < width; ix++) {
                    x_ref = x + dx + ix;
                    y_ref = y + dy + iy + ref_y0;
                    pred[dx][dy + pred_y0] =
                        (clip_ref(ref, x_ref + 0, y_ref) +
                         clip_ref(ref, x_ref + 1, y_ref) +
                         1 - rounding) >> 1;
                }
            }
        } else {
            for (iy = 0; iy < height; iy += y_incr) {
                for (ix = 0; ix < width; ix++) {
                    x_ref = x + dx + ix;
                    y_ref = y + dy + iy + ref_y0;
                    pred[dx][dy + pred_y0] =
                        clip_ref(ref, x_ref, y_ref);
                }
            }
        }
    }
```

7.5.9.2 Forward mode

Only the forward vector (MVFx,MVFy) is applied in this mode. The prediction blocks Pf_Y, Pf_U, and Pf_V are generated from the forward reference VOP, ref_Y_for for luminance component and ref_U_for and ref_V_for for chrominance components, as follows:

```
mc(Pf_Y, ref_Y_for, x, y, 16, 16, MVFx, MVFy, 0, 0, 0, 1);
mc(Pf_U, ref_U_for, x/2, y/2, 8, 8, MVFx_chro, MVFy_chro, 0, 0, 0,1);
mc(Pf_V, ref_V_for, x/2, y/2, 8, 8, MVFx_chro, MVFy_chro, 0, 0, 0,1);
```

where (MVFx_chro, MVFy_chro) is motion vector derived from the luminance motion vector by dividing each component by 2 then rounding on a basis of Table 1.4. Here (and hereafter) the function MC is defined in section 7.5.9.

7.5.9.3 Backward mode

Only the backward vector (MVBx,MVBy) is applied in this mode. The prediction blocks Pb_Y, Pb_U, and Pb_V are generated from the backward reference VOP, ref_Y_back for luminance component and ref_U_back and ref_V_back for chrominance components, as follows:

```
mc(Pb_Y, ref_Y_back, x, y, 16, 16, MVBx, MVBy, 0, 0, 0, 1);
mc(Pb_U, ref_U_back, x/2, y/2, 8, 8, MVBx_chro, MVBy_chro, 0, 0, 0,1);
mc(Pb_V, ref_V_back, x/2, y/2, 8, 8, MVBx_chro, MVBy_chro, 0, 0, 0,1);
```

where (MVBx_chro, MVBy_chro) is motion vector derived from the luminance motion vector by dividing each component by 2 then rounding on a basis of Table 1.4.

7.5.9.4 Bi-directional mode

Both the forward vector (MVFx,MVFy) and the backward vector (MVBx,MVBy) are applied in this mode. The prediction blocks Pi_Y, Pi_U, and Pi_V are generated from the forward and backward reference VOPs by doing the forward prediction, the backward prediction and then averaging both predictions pixel by pixel as follows.

```
mc(Pf_Y, ref_Y_for, x, y, 16, 16, MVFx, MVFy, 0, 0, 0, 1);
mc(Pf_U, ref_U_for, x/2, y/2, 8, 8, MVFx_chro, MVFy_chro, 0, 0, 0,1);
mc(Pf_V, ref_V_for, x/2, y/2, 8, 8, MVFx_chro, MVFy_chro, 0, 0, 0,1);
mc(Pb_Y, ref_Y_back, x, y, 16, 16, MVBx, MVBy, 0, 0, 0, 1);
mc(Pb_U, ref_U_back, x/2, y/2, 8, 8, MVBx_chro, MVBy_chro, 0, 0, 0,1);
mc(Pb_V, ref_V_back, x/2, y/2, 8, 8, MVBx_chro, MVBy_chro, 0, 0, 0,1);
Pi_Y[i][j] = (Pf_Y[i][j] + Pb_Y[i][j] + 1)>>1;        i,j=0,1,2…15;
Pi_U[i][j] = (Pf_U[i][j] + Pb_U[i][j] + 1)>>1;        i,j=0,1,2…8;
Pi_V[i][j] = (Pf_V[i][j] + Pb_V[i][j] + 1)>>1;        i,j=0,1,2…8;
```

where (MVFx_chro, MVFy_chro) and (MVBx_chro, MVBy_chro) are motion vectors derived from the forward and backward luminance motion vectors by dividing each component by 2 then rounding on a basis of Table 1.4, respectively.

7.5.9.5 Direct mode

This mode uses direct bi-directional motion compensation derived by employing I- or P-VOP macroblock motion vectors and scaling them to derive forward and backward motion vectors for macroblocks in B-VOP. This is the only mode which makes it possible to use motion vectors on 8x8 blocks. Only one delta motion vector is allowed per macroblock.

7.5.9.5.1 Formation of motion vectors for the direct mode

The direct mode utilises the motion vectors (MVs) of the co-located macroblock in the most recently decoded I- or P-VOP. The co-located macroblock is defined as the macroblock which has the same horizontal and vertical index with the current macroblock in the B-VOP. The MV vectors are the block vectors of the co-located macroblock after applying the vector padding defined in section 7.5.1.6. If the co-located macroblock is transparent and thus the MVs are not available, the direct mode is still enabled by setting MV vectors to zero vectors.

7.5.9.5.2 Calculation of vectors

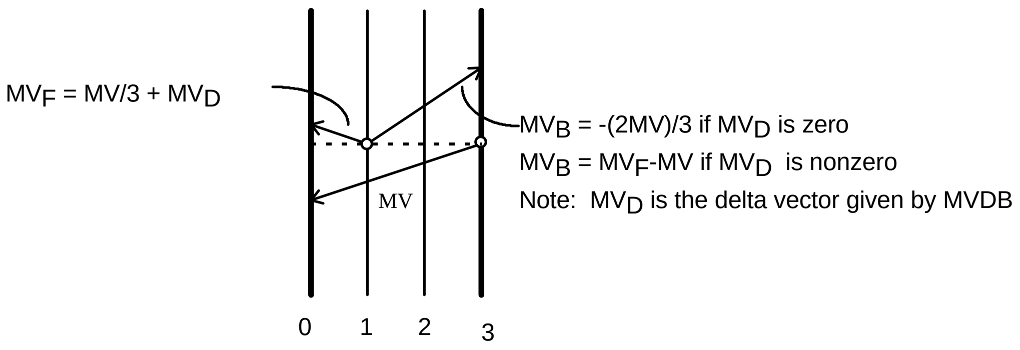

MV_F = MV/3 + MV_D
MV_B = -(2MV)/3 if MV_D is zero
MV_B = MV_F-MV if MV_D is nonzero
Note: MV_D is the delta vector given by MVDB
MV
0
1
2
3

Figure 7-23 Direct Bi-directional Prediction

Figure 7-23 shows scaling of motion vectors. The calculation of forward and backward motion vectors involves linear scaling of the collocated block in temporally next I- or P-VOP, followed by correction by a delta vector (MVDx,MVDy). The forward and the backward motion vectors are {(MVFx[i],MVFy[i]), (MVBx[i],MVBy[i]), i = 0,1,2,3} and are given in half sample units as follows.

MVFx[i] = (TRB x MVx[i]) / TRD + MVDx

MVBx[i] = (MVDx==0)? ((TRB - TRD) x MVx[i]) / TRD : MVFx[i] - MVx[i]

MVFy[i] = (TRB x MVy[i]) / TRD + MVDy

MVBy[i] = (MVDy==0)? ((TRB - TRD) x MVy[i]) / TRD : MVFy[i] - MVy[i]

i = 0,1,2,3.

where {(MVx[i],MVy[i]), i = 0,1,2,3} are the MV vectors of the co-located macroblock, TRD is the difference in temporal reference of the B-VOP and the previous reference VOP. TRD is the difference in temporal reference of the temporally next reference VOP with temporally previous reference VOP, assuming B-VOPs or skipped VOPs in between. The calculation of TRB and TRD is defined in section 7.4.12.4.1.

7.5.9.5.3 Generation of prediction blocks

Motion compensation for luminance is performed individually on 8x8 blocks to generate a macroblock. The process of generating a prediction block simply consists of using computed forward and backward motion vectors {(MVFx[i],MVFy[i]), (MVBx[i],MVBy[i]), i = 0,1,2,3} to obtain appropriate blocks from reference VOPs and averaging these blocks, same as the case of bi-directional mode except that motion compensation is performed on 8x8 blocks.

For the motion compensation of both chrominance blocks, the forward motion vector (MVFx_chro, MVFy_chro) is calculated by the sum of K forward luminance motion vectors dividing by 2K and then rounding toward the nearest half sample position as defined in Table 7-7 to Table 7-10. The backward motion vector (MVBx_chro, MVBy_chro) is derived in the same way. The rest process is the same as the chrominance motion compensation of the bi-directional mode described in section 7.5.12.3.

7.5.9.5.4 Motion compensation in skipped macroblocks

If the co-located macroblock in the most recently decoded I- or P-VOP is skipped, the current B-macroblock is treated as the forward mode with the zero motion vector (MVFx,MVFy). If the MODB equals zero the current B-macroblock is reconstructed by using the direct mode with zero delta vector.

7.6 Interlaced video decoding

This clause specifies the additional decoding process that a decoder shall perform to recover VOP data from the coded bitstream when the interlaced flag in the VOP header is set to “1”. Interlaced information (Sec. 6.3.7.2) specifies the method to decode bitstream of interlaced VOP.

7.6.1 Field DCT and DC and AC Prediction

When dct_type flag is set to ‘1’ (field DCT coding), DCT coefficients of luminance data are formed such that each 8x8 block consists of data from one field as being shown in Figure 6-7. DC and optional AC (see “ac_pred_flag”) prediction will be performed for a intra-coded macroblock. For the intra macroblocks which have dct_type flag being set to “1”, DC/AC prediction are performed to field blocks shown in Figure 7-24. After taking inverse DCT, all luminance blocks will be inverse permuted back to (frame) macroblock. Chrominance (block) data are not effected by dct_type flag.

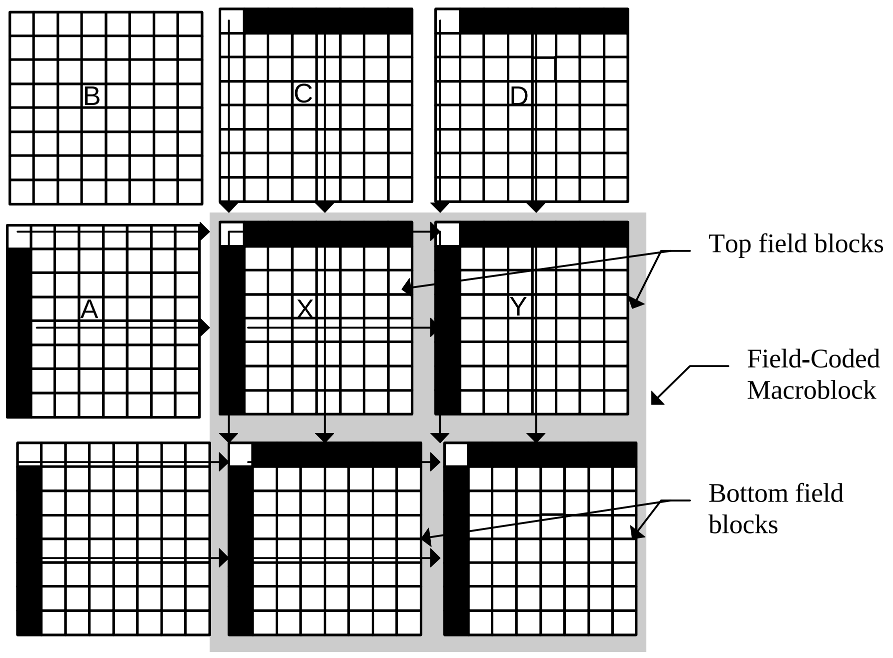


Figure 7-24 Previous neighboring blocks used in DC/AC prediction for interlaced intra blocks.

7.6.2 Motion compensation

For non-intra macroblocks in P- and B-VOPs, motion vectors are extracted syntactically following Section 6.2.7 “Macroblock”. The motion vector decoding is performed separately on the horizontal and vertical components.

7.6.2.1 Motion vector decoding in P-VOP

For each component of motion vector in P-VOPs, the median value of the candidate predictor vectors for the same component is computed and add to corresponding component of the motion vector difference obtained from the bitstream. To decode the motion vectors in a P-VOP, the decoder shall first extract the differential motion vectors ($(MVDx_{f1}, MVDy_{f1})$ and $(MVDx_{f2}, MVDy_{f2})$ for top and bottom fields of a field predicted macroblock, respectively) by a use of variable length decoding and then determine the predictor vector from three candidate vectors. These candidate predictor vectors are generated from the three motion vectors of three spatial neighborhood decoded macroblocks or blocks as follows.

CASE 1 :

If the current macroblock is a field predicted macroblock and none of the coded spatial neighborhood macroblocks is a field predicted macroblock, then candidate predictor vectors MV1, MV2, and MV3 are defined by Figure 7-25. If the candidate block *i* is not in four MV motion (8x8) mode, MV*i* represents the motion vector for the macroblock. If the candidate block *i* is in four MV motion (8x8) mode, the 8x8 block motion vector closest to the upper left block of the current MB is used. The predictors for the horizontal and vertical components are then computed by

$$P_x = Median(MV1x, MV2x, MV3x)$$
$$P_y = Median(MV1y, MV2y, MV3y).$$

For differential motion vectors both fields use the same predictor and motion vectors are recovered by

$$MVx_{f1} = MVDx_{f1} + P_x$$
$$MVy_{f1} = 2 * (MVDy_{f1} + (P_y / 2))$$
$$MVx_{f2} = MVDx_{f2} + P_x$$
$$MVy_{f2} = 2 * (MVDy_{f2} + (P_y / 2))$$

where “/” is integer division with truncation toward 0. Note that all motion vectors described above are specified as integers with one LSB representing a half-pel displacement. The vertical component of field motion vectors always even (in half-pel frame coordinates). Vertical half-pel interpolation between adjacent lines of the same field is denoted by MVy_{fi} be an odd multiple of 2 (e.g. -2,2,6,..) No vertical interpolation is needed when MVy_{fi} is an multiple of 4 (it is a full pel value).

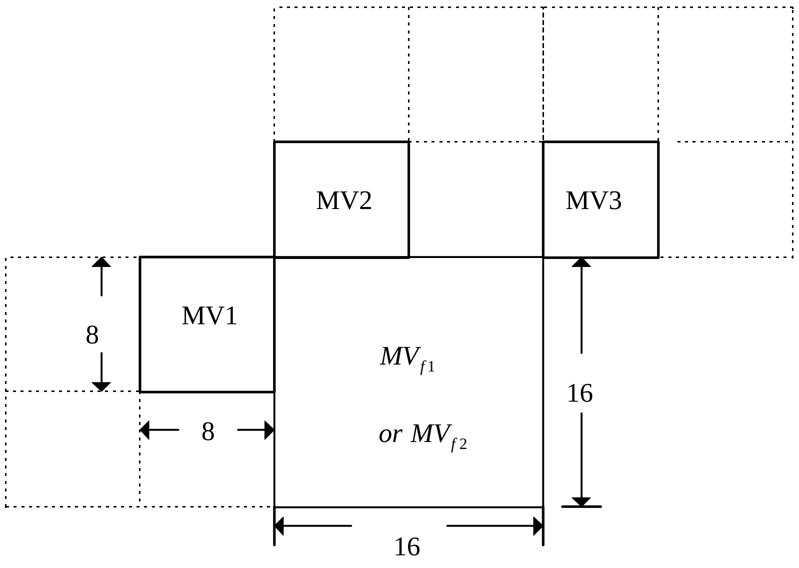


Figure 7-25 Example of motion vector prediction for field predicted macroblocks (Case1)

CASE 2 :

If the current macroblock or block is frame predicted macroblock or block and if at least one of the coded spatial neighborhood macroblocks is a field predicted macroblock, then the candidate predictor vector for each field predicted macroblock will be generated by averaging two field motion vectors such that all fractional pel offsets are mapped into the half-pel displacement. Each component (P_x or P_y) of the final predictor vector is the median value of the candidate predictor vectors for the same component. The motion vector is recovered by

$$MVx = MVDx + P_x$$
$$MVy = MVDy + P_y.$$

where

$$P_x = Median\left(MV1x, Div2Round(MVx_{f1} + MVx_{f2}), MV3x\right),$$
$$P_y = Median\left(MV1y, Div2Round(MVy_{f1} + MVy_{f2}), MV3y\right),$$

Div2Round(x) make the use of Table 7-10 as follows : *Div2Round(x)=((x>>1)&~1)+Table7_1b[x&3].*

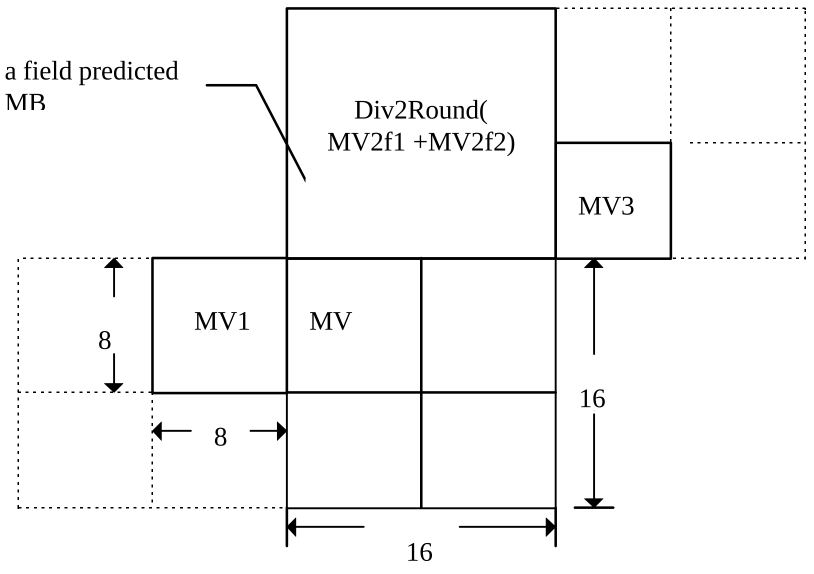


Figure 7-26 Example of motion vector prediction for field predicted macroblocks (Case 2)

CASE 3 :

Assume that the current macroblock is a field predicted macroblock and at least one of the coded spatial neighborhood macroblocks is a field predicted macroblock. If the candidate block *i* is field predicted, the candidate predictor vector MV*i* will be generated by averaging two field motion vectors such that all fractional pel offsets are mapped into the half-pel displacement as discribed in CASE 2. If the candidate block *i* is neither in four MV motion (8x8) mode nor in field prediction mode, MV*i* represents the frame motion vector for the macroblock. If the candidate block *i* is in four MV motion (8x8) mode, the 8x8 block motion vector closest to the upper left block of the current MB is used. The predictors for the horizontal and vertical components are then computed by

$$P_x = Median(MV1x, MV2x, MV3x)$$
$$P_y = Median(MV1y, MV2y, MV3y)$$

where

$$MVi\ x = Div2Round(MVx_{f1} + MVx_{f2}),$$
$$MVi\ y = Div2Round(MVy_{f1} + MVy_{f2}),$$

for some *i* in {1,2,3}.

For differential motion vectors both fields use the same predictor and motion vectors are recovered by (see both Figure 7-25 and Figure 7-26)

$$MVx_{f1} = MVDx_{f1} + P_x$$
$$MVy_{f1} = 2 * (MVDy_{f1} + (P_y / 2))$$
$$MVx_{f2} = MVDx_{f2} + P_x$$
$$MVy_{f2} = 2 * (MVDy_{f2} + (P_y / 2))$$

The motion compensated prediction macroblock is calculated calling the "field_compensate_one_reference" using the motion vectors calculated above. The top_field_ref, bottom_field_ref, and rounding type come directly from the syntax as forward_top_field_reference, forward_bottom_field_reference and VOP_rounding_type respectively. The reference VOP is defined such the the even lines (0, 2, 4, ...) are the top field and the odd lines (1, 3, 5, ...) are the bottom field.

```
field_motion_compensate_one_reference(
    luma_pred, cb_pred, cr_pred, /* Prediction component pel array */
    luma_ref, cb_ref, cr_ref,    /* Reference VOP pel arrays */
    mv_top_x, mv_top_y,           /* top field motion vector */
    mv_bot_x, mv_bot_y,          /* bottom field motion vector */
    top_field_ref,               /* top field reference */
    bottom_field_ref,            /* bottom field reference */
    x, y,                        /* current luma macroblock coords */
    rounding_type)               /* rounding type */
{
    mc(luma_pred, luma_ref, x, y, 16, 16, mv_top_x, mv_top_y,
        rounding_type, 0, top_field_ref, 2);
    mc(luma_pred, luma_ref, x, y, 16, 16, mv_bot_x, mv_bot_y,
        rounding_type, 1, bottom_field_ref, 2);
    mc(cb_pred, cb_ref, x/2, y/2, 8, 8,
        Div2Round(mv_top_x), Div2Round(mv_top_y),
        rounding_type, 0, top_field_ref, 2);
    mc(cr_pred, cr_ref, x/2, y/2, 8, 8,
        Div2Round(mv_top_x), Div2Round(mv_top_y),
        rounding_type, 0, top_field_ref, 2);
    mc(cb_pred, cb_ref, x/2, y/2, 8, 8,
        Div2Round(mv_bot_x), Div2Round(mv_bot_y),
        rounding_type, 0, top_field_ref, 2);
    mc(cr_pred, cr_ref, x/2, y/2, 8, 8,
        Div2Round(mv_bot_x), Div2Round(mv_bot_y),
        rounding_type, 0, top_field_ref, 2);
}
```

In the case that OBMC flag is set to "1", the OBMC is not applied if the current MB is field-predicted. If the current MB is frame-predicted (including 8x8 mode) and some adjacent MBs are field-predicted, the motion vectors of those field-predicted MBs for OBMC are computed in the same manner as the candidate predictor vectors for field-predicted MBs are.

7.6.2.2 Motion vector decoding in B-VOP

For interlaced B-VOPs, a macroblock can be coded using (1) direct coding, (2) 16x16 motion compensation (includes forward, backward & bidirectional modes), or (3) field motion compensation (includes forward, backward & bidirectional modes). Forward, backward and bidirectional coding modes work in the same manner as in MPEG-1 / 2 with the difference that a VOP is used for prediction instead of a picture. Motion vector in half sample accuracy will be employed for a 16x16 macroblock being coded. Chrominance vectors are derived by scaling of luminance vectors using the rounding tables described in Table 7-10 (i.e. by applying *Div2Round* to the luminance motion vectors). These coding modes except direct coding mode allow switching of quantizer from the one previously in use. Specification of DQUANT, a differential quantizer involves a 2-bit overhead as discussed earlier. In direct coding mode, the quantizer value for previous coded macroblock is used.

For interlaced B-VOP motion vector predictors, four prediction motion vectors (PMVs) are used:

Table 7-11 Prediction motion vector allocation for interlaced P-VOPs

Function	**PMV**
Top field forward	0

Bottom field forward	1
Top field backward	2
Bottom field backward	3

These PMVs are used as follows for the different macroblock prediction modes:

Table 7-12 Prediction motion vectors for interlaced B-VOP decoding

Macroblock mode	PMVs used	PMVs updated
Direct	none	none
Frame forward	0	0,1
Frame backward	2	2,3
Frame bidirectional	0,2	0,1,2,3
Field forward	0,1	0,1
Field backward	2,3	2,3
Field bidirectional	0,1,2,3	0,1,2,3

The PMVs used by a macroblock are set to the value of current macroblock motion vectors after being used.

When a frame macroblock is decoded, the two field PMVs (top and bottom field) for each prediction direction are set to the same frame value. The PMVs are reset to zero at the beginning of each row of macroblocks. The predictors are not zeroed by skipped macroblocks or direct mode macroblocks.

The frame based motion compensation modes are described in section 7.5. The field motion compensation modes are calculated using the “field_motion_compensate_one_reference()” pseudo code function described above. The field forward mode is denoted by mb_type == “0001” and field_prediction == “1”. The PMV update and calculation of the motion compensated prediction is shown below. The luma_fwd_ref_VOP[][], cb_fwd_ref_VOP[][], cr_fwd_ref_VOP[][] denote the entire forward (past) anchor VOP pixel arrays. The coordinates of the upper left corner of the luminance macroblock is given by (x, y) and MVD[].x and MVD[].y denote an array of the motion vector differences in the order they occur in the bitstream for the current macroblock.

```
    PMV[0].x = PMV[0].x + MVD[0].x;
    PMV[0].y = 2 * (PMV[0].y / 2 + MVD[0].y);
    PMV[1].x = PMV[1].x + MVD[1].x;
    PMV[1].y = 2 * (PMV[1].y / 2 + MVD[1].y);
    field_motion_compensate_one_reference(
        luma_pred, cb_pred, cr_pred,
        luma_fwd_ref_VOP, cb_fwd_ref_VOP, cr_fwd_ref_VOP,
        PMV[0].x, PMV[0].y, PMV[1].x, PMV[1].y,
        forward_top_field_reference,
        forward_bottom_field_reference,
        x, y, 0);
```

The field backward mode is denoted by mb_type == “001” and field_prediction == “1”. The PMV update and prediction calculation is outlined the following pseudo code. The luma_bak_ref_VOP[][], cb_bak_ref_VOP[][], cr_bak_ref_VOP[][] denote the entire forward (past) anchor VOP pixel arrays.

```
    PMV[2].x = PMV[2].x + MVD[0].x;
    PMV[2].y = 2 * (PMV[2].y / 2 + MVD[0].y);
    PMV[3].x = PMV[1].x + MVD[1].x;
    PMV[3].y = 2 * (PMV[3].y / 2 + MVD[1].y);
    field_motion_compensate_one_reference(
        luma_pred, cb_pred, cr_pred,
        luma_bak_ref_VOP, cb_bak_ref_VOP, cr_bak_ref_VOP,
        PMV[2].x, PMV[2].y, PMV[3].x, PMV[3].y,
        backward_top_field_reference,
        backward_bottom_field_reference,
        x, y, 0);
```

The bidirectional field prediction is used when mb_type == “01” and field_prediction == “1”. The prediction macroblock (in luma_pred[][], cb_pred[][], and cr_pred[][]) is calculated by:

```
for (mv = 0; mv < 4; mv++) {
        PMV[mv].x = PMV[mv].x + MVD[mv].x;
        PMV[mv].y = 2 * (PMV[mv].y / 2 + MVD[mv].y);
    }
    field_motion_compensate_one_reference(
        luma_pred_fwd, cb_pred_fwd, cr_pred_fwd,
        luma_fwd_ref_VOP, cb_fwd_ref_VOP, cr_fwd_ref_VOP,
        PMV[0].x, PMV[0].y, PMV[1].x, PMV[1].y,
        forward_top_field_reference,
        forward_bottom_field_reference,
        x, y, 0);
    field_motion_compensate_one_reference(
        luma_pred_bak, cb_pred_bak, cr_pred_bak,
        luma_bak_ref_VOP, cb_bak_ref_VOP, cr_bak_ref_VOP,
        PMV[2].x, PMV[2].y, PMV[3].x, PMV[3].y,
        backward_top_field_reference,
        backward_bottom_field_reference,
        x, y, 0);
    for (iy = 0; iy < 16; iy++) {
        for (ix = 0; ix < 16; ix++) {
            luma_pred[ix][iy] = (luma_pred_fwd[ix][iy] +
                                 luma_pred_bak[ix][iy] + 1) >> 1;
        }
    }
    for (iy = 0; iy < 8; iy++) {
        for (ix = 0; ix < 8; ix++) {
            cb_pred[ix][iy] = (cb_pred_fwd[ix][iy] +
                               cb_pred_bak[ix][iy] + 1) >> 1;
            cr_pred[ix][iy] = (cr_pred_fwd[ix][iy] +
                               cr_pred_bak[ix][iy] + 1) >> 1;
        }
    }
```

The direct mode prediction can be either progressive (see section 7.5.9.5) or interlaced as described below. Interlaced direct mode is used when ever the co-located macroblock (macroblock with the same coordinates) of the future anchor VOP has field_predition flag is “1”. Note that if the future macroblock is skipped, or intra, the direct mode prediction is progressive. Otherwise, interlaced direct mode prediction is used.

Interlaced direct coding mode is an extension of progressive direct coding mode. Four derived field motion vectors are calculated from the forward field motion vectors of the co-located future anchor VOP, a single differential motion vector and the temporal position of the B-VOP fields with respect to the fields of the past and future anchor VOPs. The four derived field motion vectors are denoted mvf[0] (top field forward) mvf[1], (bottom field forward), mvb[0] (top field backward), and mvb[1] (bottom field backward). MV[i] is the future anchor picture motion vector for the top (i == 0) and bottom (i == 1) fields. Only one delta motion vector (used for both field), MVD[0], occurs in the bitstream for the field direct mode predicted macroblock. MVD[0] is decoded assuming f_code == 1 regardless of the number in VOP header. The interlaced direct mode prediction (in luma_pred[][], cb_pred[][] and cr_pred[][]) is calculated as shown below.

```
    for (i = 0; i < 2; i++) {
        mvf[i].x = (TRB[i] * MV[i].x) / TRD[i] + MVD[0].x;
        mvf[i].y = (TRB[i] * MV[i].y) / TRD[i] + MVD[0].y;
        mvb[i].x = (MVD[i].x == 0) ?
            (((TRB[i] - TRD[i]) * MV[i].x) / TRD[i]) :
            mvf[i].x - MV[i].x);
        mvb[i].y = (MVD[i].y == 0) ?
            (((TRB[i] - TRD[i]) * MV[i].y) / TRD[i]) :
            mvf[i].y - MV[i].y);
    field_motion_compensate_one_reference(
        luma_pred_fwd, cb_pred_fwd, cr_pred_fwd,
        luma_fwd_ref_VOP, cb_fwd_ref_VOP, cr_fwd_ref_VOP,
        mvf[0].x, mvf[0].y, mvf[1].x, mvf[1].y,
        colocated_future_mb_top_field_reference,
        colocated_future_mb_bottom_field_reference,
        x, y, 0);
    field_motion_compensate_one_reference(
        luma_pred_bak, cb_pred_bak, cr_pred_bak,
        luma_bak_ref_VOP, cb_bak_ref_VOP, cr_bak_ref_VOP,
        mvb[1].x, mvb[1].y, mvb[1].x, mvb[1].y,
        0, 1, x, y, 0);
    for (iy = 0; iy < 16; iy++) {
        for (ix = 0; ix < 16; ix++) {
            luma_pred[ix][iy] = (luma_pred_fwd[ix][iy] +
                                 luma_pred_bak[ix][iy] + 1) >> 1;
        }
    }
    for (iy = 0; iy < 8; iy++) {
        for (ix = 0; ix < 8; ix++) {
            cb_pred[ix][iy] = (cb_pred_fwd[ix][iy] +
                               cb_pred_bak[ix][iy] + 1) >> 1;
            cr_pred[ix][iy] = (cr_pred_fwd[ix][iy] +
                               cr_pred_bak[ix][iy] + 1) >> 1;
        }
    }
```

The temporal references (TRB[i] and TRD[i]) are distances in time expressed in field periods. Figure 7-27 shows how they are defined for the case where i is 0 (top field of the B-VOP). The bottom field is analogously.

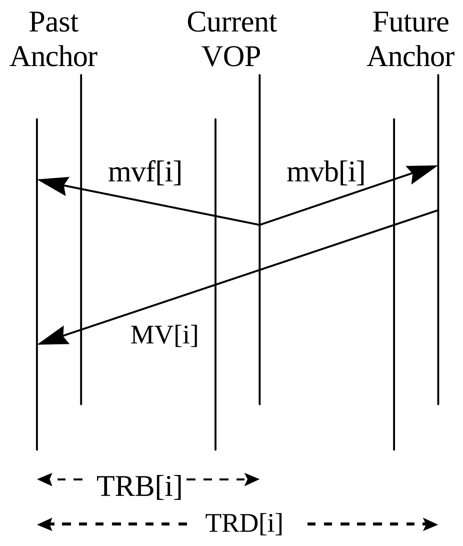


Figure 7-27 Interlaced direct mode

The calculation of TRD[i] and TRB[i] depends not only on the current field, reference field, and frame temporal references, but also on whether the current video is top field first or bottom field first.

$$TRD[i] = 2*(T(future)//Tframe - T(past)//Tframe) + \delta[i]$$

$$TRB[i] = 2*(T(current)//Tframe - T(past)//Tframe) + \delta[i]$$

where T(future), T(current) and T(past) are the cumulative VOP times calculated from modulo_time_base and VOP_time_increment of the future, current and past VOPs in display order. Tframe is the frame period determined by

$$Tframe = T(first\_B\_VOP) - T(past\_anchor\_of\_first\ B\_VOP)$$

where first_B_VOP denotes the first B-VOP following the Video Object Layer syntax. The important thing about Tframe is that the period of time between consecutive fields which constitute an interlaced frame is assuemed to be 0.5 * Tframe for purposes of scaling the motion vectors.

The value of δ is determined from Table 7-13; it is a function of the current field parity (top or bottom), the reference field of the co-located macroblock (macroblock at the same coordinates in the furture anchor VOP), and the value of top_field_first in the B-VOP's video object plane syntax.

Table 7-13 Selection of the parameter *d*

future anchor VOP reference fields of the co-located macroblock		top_field_first == 0		top_field_first == 1	
Top field reference	Bottom field reference	Top field, δ[0]	Bottom field, δ[1]	Top field, δ[0]	Bottom field, δ[1]
0	0	0	-1	0	1
0	1	0	0	0	0

future anchor VOP reference fields of the co-located macroblock		top_field_first == 0		top_field_first == 1	
Top field reference	Bottom field reference	Top field, δ[0]	Bottom field, δ[1]	Top field, δ[0]	Bottom field, δ[1]
1	0	1	-1	-1	1
1	1	1	0	-1	0

The top field prediction is based on the top field motion vector of the P-VOP macroblock of the future anchor picture. The past reference field is the reference field selected by the co-located macroblock of the future anchor picture for the top field. Analogously, the bottom field predictor is the average of pixels obtained from the future anchor's bottom field and the past anchor field referenced by the bottom field motion vector of the corresponding macroblock of the future anchor picture. When interlaced direct mode is used, VOP_time_increment_resolution must be the smallest integer greater than or equal to the number of frames per second. In each VOP, VOP_time_increment counts individual frames within a second.

7.7 Sprite decoding

The clause specifies the additional decoding process for a sprite video object. The sprite decoding can operate in three modes: basic sprite decoding, low-latency sprite decoding and scalable sprite decoding. Figure 7-28 is a diagram of the sprite decoding process. It is simplified for clarity.

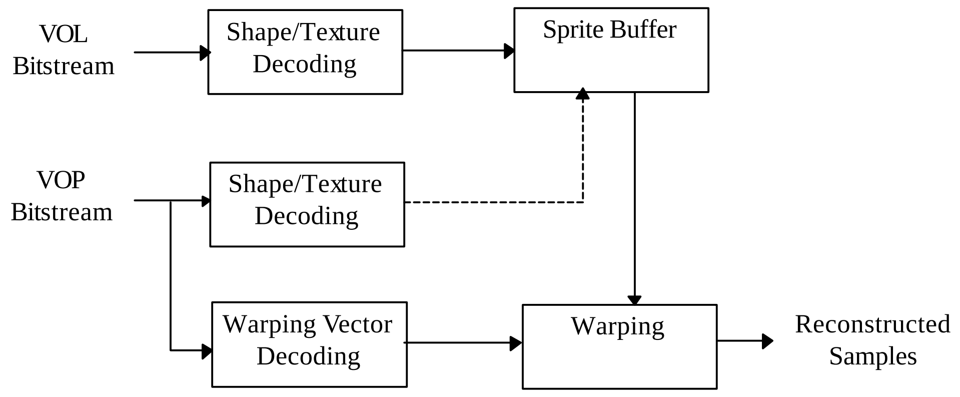


Figure 7-28 The sprite decoding process.

7.7.1 Higher syntactic structures

The various parameters in the VOL and VOP bitstreams shall be interpreted as described in clause 6. When sprite_enable == ‘1’, VOP_coding_type shall be “I” only for the initial VOP in a VOL for basic sprites (i.e. low_latency_sprite_enable == ‘0’), and all the other VOPs shall be S-VOPs (i.e. VOP_coding_type == “S”). The reconstructed I-VOP in a VOL for basic sprites is not displayed but stored in a sprite memory, and will be used by all the remaining S-VOPs in the same VOL. An S-VOP is reconstructed by applying warping to the VOP stored in the sprite memory, using the warping parameters (i.e. a set of motion vectors) embedded in the VOP bitstream. Alternatively, in a VOL for low-latency sprites (i.e. low_latency_sprite_enable == ‘1’), these S-VOPs can update the VOPVOPinformation stored in the sprite memory before applying warping.

7.7.2 Sprite Reconstruction

The luminance and chrominance data of a sprite are stored in two-dimensional arrays. The width and height of the luminance array are specified by sprite_width and sprite_height respectively. The samples in the sprite luminance and chrominance arrays are addressed by two-dimensional integer pairs (*i’*, *j’*) and (i_c’, j_c’) as defined in the following:

- Top left luminance sample
  (*i’*, *j’*) = (sprite_left_coordinate, sprite_top_coordinate)
- Bottom right luminance sample
  (*i’*, *j’*) = ((sprite_left_coordinate + sprite_width – 1),
  ((sprite_top_coordinate + sprite_height – 1))
- Top left chrominance sample
  (i_c’, j_c’) = (sprite_left_coordinate / 2, sprite_top_coordinate / 2)
- Bottom right chrominance sample
  (i_c’, j_c’) = (((sprite_left_coordinate + sprite_width) / 2 – 1),
  ((sprite_top_coordinate + sprite_height) / 2 – 1)).

Likewise, the addresses of the luminance and chrominance samples of the VOP currently being decoded are defined in the following:

- Top left sample of luminance
  (*i*, *j*) = (0, 0) for rectangular VOPs, and
  (*i*, *j*) = (VOP_horizontal_mc_spatial_ref, VOP_vertical_mc_spatial_ref) for non-rectangular VOPs
- Bottom right sample of luminance
  (*i*, *j*) = (video_object_layer_width - 1, video_object_layer_height - 1) for rectangular VOPs, and
  (*i*, *j*) = (VOP_horizontal_mc_spatial_ref + VOP_width - 1,
  VOP_vertical_mc_spatial_ref + VOP_height - 1) for non-rectangular VOPs
- Top left sample of chrominance
  (i_c, j_c) = (0, 0) for rectangular VOPs, and
  (i_c, j_c) = (VOP_horizontal_mc_spatial_ref / 2, VOP_vertical_mc_spatial_ref / 2) for non-rectangular VOPs
- Bottom right sample of chrominance
  (i_c, j_c) = (video_object_layer_width / 2 - 1, video_object_layer_height / 2 - 1) for rectangular VOPs, and
  (i_c, j_c) = ((VOP_horizontal_mc_spatial_ref + VOP_width) / 2 - 1,
  (VOP_vertical_mc_spatial_ref + VOP_height) / 2 - 1) for non-rectangular VOPs

7.7.3 Low-latency sprite reconstruction

This section allows a large static sprite to be reconstructed at the decoder by properly incorporating its corresponding pieces. There are two types of pieces recognized by the decoder—object and update. The decoded sprite object-piece (i.e., embedded in a sprite-VOP with low_latence_sprite_enable==1 and sprite_transmit_mode=="piece") is a highly quantized version of the original sprite piece while the sprite update-piece (i.e., sprite_transmit_mode=="update") is a residual designed to improve upon the quality of decoded object-piece. Sprite pieces are rectangular pieces of texture (and shape for the object-piece) and can contain "holes," corresponding to macroblocks, that do not need to be decoded. Five parameters are required by the decoder to properly incorporate the pieces: piece_quant, piece_width, piece_height, piece_xoffset, and piece_yoffset.

Macroblocks raster scanning is employed to decode each piece. However, whenever the scan encounters a macroblock which has been part of some previously sent sprite piece, then the macroblock is not decoded and its corresponding macroblock layer is empty. In that case, the decoder treats the macroblock as a hole in the current sprite piece. Since a macroblock can be refined as long as there is some available bandwidth, more than one update may be decoded per macroblock and the holes for a given refinement step have no relationship to the holes of later refinement steps. Therefore, the decoding process of a hole for an update piece is different than that for the object-piece. For the object-piece, no information is decoded at all and the decoder must "manage" where "holes" lie. (see clause 7.8.3.1). For the update-piece, the not_coded bit is decoded to indicate whether or not one more refinement should be decoded for this given macroblock. (see clause 7.8.3.2). Note that a hole could be non-transparent and have had shape information decoded previously. Multiple intermingled object-pieces and update-pieces may be decoded at the same current VOP. Part of a sequence could consist for example of rapidly showing a zooming out effect, a panning to the right, a zooming in, and finally a panning to the left. In this case, the first decoded object-piece covers regions on all four sides of the previous VOP transmitted piece, which is now treated as a hole and not decoded again. The second decoded object-piece relates to the right panning, and the third object-piece is a smaller left-panning piece due to the zooming-in effect. Finally, the last piece is different; instead of an object, it contains the update for some previous object-piece of zooming-in (thus, the need to update to refine for higher quality). All four pieces will be decoded within the same VOP. When sprite_transmit_mode = =”pause,” the decoder recognizes that all sprite object-pieces and update-pieces for the current VOP session have been sent. However, when sprite_transmit_mode = "stop," the decoder understands that all object and update-pieces have been sent for the entire video object layer, not just for the current VOP. session. In addition, once all object-pieces or update-pieces have been decoded during a VOP session (i.e., signaled by sprite_transmit_mode == "pause" or sprite_transmit_mode == "stop"), the static sprite is padded (as defined in section 7.5.1), then the portion to be displayed is warped, to complete the current VOP session.

For the S-VOPs (i.e., VOP_coding_type == "S"), the macroblock layer syntax of object-pieces is the same as those of I-VOP. Therefore, shape and texture are decoded using the macroblock layer structure in I-VOPs with the quantization of intra macroblocks. The syntax of the update-pieces is similar to the P-VOP inter-macroblock syntax with the quantization of non-intra macroblocks); however, the differences are indicated in Table 11-1, specifically that there are no motion vectors and shape information included in this decoder syntax structure. In summary, this decoding process supports the construction of any large sprite image progressively, both spatially and in terms of quality.

7.7.3.1 Decoding of holes in sprite object-piece

Implementation of macroblock scanning must account for the possibility that a macroblock uses prediction based on some macroblock sent in a previous piece. When an object-piece with holes is decoded, the decoder in the process of reconstruction acts as if the whole original piece were decoded, but actually only the bitstream corresponding to the "new macroblock" is received. Whenever macroblocks raster scanning encounters a hole, the decoder needs to manage the retrieval of relevant information (e.g. DCT quantization parameters, AC and DC prediction parameters, and BAB bordering values) from the corresponding macroblock decoded earlier.

7.7.3.2 Decoding of holes in sprite update-pieces

In contrast to the send_mb() used by the object-pieces, the update-pieces use the not_coded bit. When not_coded = 1 in the P-VOP syntax, the decoder recognizes that the corresponding macroblock is not refined by the current sprite update-piece. When not_coded = 0 in the P-VOP syntax, the decoder recognizes that this macroblock is refined. The prediction for the update piece is obtained by extracting the "area" of the static sprite defined by (piece_width, piece_height, piece_xoffset, piece_yoffset). This area is then padded and serves as prediction for the update pieces. Since there is no shape information included in an update-piece, the result of its transparent_mb() is retrieved from the corresponding macroblock in the object-piece decoded earlier. In addition, an update macroblock cannot be transmitted before its corresponding object macroblock. As a result, the very first sprite piece transmitted in the low-latency mode shall be an object-piece.

7.7.4 Sprite reference point decoding

The syntatic elements in encode_sprite_trajectory () and below shall be interpreted as specified in clause 6. du[i] and dv[i] (0 =< i < no_sprite_point) specifies the mapping between indexes of some reference points in the VOP and the corresponding reference points in the sprite. These points are referred to as VOP reference points and sprite reference points respectively in the rest of the specification.

The index values for the VOP reference points are defined as:

(i0, j0) = (0, 0) when video_object_layer_shape == 'rectangle', and
(VOP_horizontal_mc_spatial_ref, VOP_vetical_mc_spatial_ref) otherwise,
(i1, j1) = (i0+W, j0),
(i2, j2) = (i0, j0 + H),
(i3, j3) = (i0+W, j0+H)

where W = video_object_layer_width and H = video_object_layer_height when video_object_layer_shape == 'rectangle' or W = VOP_width and H = VOP_height otherwise. Only the index values with subscripts less than no_sprite_point shall be used for the rest of the decoding process.

The index values for the sprite reference points shall be calculated as follows:

$(i_0', j_0') = (s / 2) (2\, i_0 + du[0], 2\, j_0 + dv[0])$
$(i_1', j_1') = (s / 2) (2\, i_1 + du[1] + du[0], 2\, j_1 + dv[1] + dv[0])$
$(i_2', j_2') = (s / 2) (2\, i_2 + du[2] + du[0], 2\, j_2 + dv[2] + dv[0])$
$(i_3', j_3') = (s / 2) (2\, i_3 + du[3] + du[2] + du[1] + du[0], 2\, j_3 + dv[3] + dv[2] + dv[1] + dv[0])$

where i_0', j_0', etc are integers in $\frac{1}{s}$ pel accuracy, where s is specified by sprite_warping_accuracy. Only the index values with substcripts less than no_sprite_point need to be calculated.

When no_of_sprite_warping_points == 2 or 3, the index values for the *virtual sprite points* are additionally calculated as follows:

(i1'', j1'') = (16 (i0 + W') + ((W – W') (r i0' – 16 i0) + W' (r i1' – 16 i1)) // W,
16 j0 + ((W – W') (r j0' – 16 j0) + W' (r j1' – 16 j1)) // W)
(i2'', j2'') = (16 i0 + ((H – H') (r i0' – 16 i0) + H' (r i2' – 16 i2)) // H,
16 (j0 + H') + ((H – H') (r j0' – 16 j0) + H' (r j2' – 16 j2)) // H)

where i_1'', j_1'', i_2'', and j_2'' are integers in $\frac{1}{16}$ pel accuracy, and $r = 16/s$. W' and H' are defined as the smallest integers that satisfy the following condition::

$W' = 2\alpha$, $H' = 2\beta$, $W' \geq W$, $H' \geq H$, $\alpha > 0$, $\beta > 0$, both α and β are integers.

The calculation of i_2'', and j_2'' is not necessary when no_of_sprite_warping_points == 2.

7.7.5 Warping

For any pixel (i, j) inside the VOP boundary, $(F(i, j), G(i, j))$ and $(F_c(i_c, j_c), G_c(i_c, j_c))$ are computed as described in the following. These quantities are then used for sample reconstruction as specified in clause 7.7.6. The following notations are used to simplify the description:

I = i - i0,
J = j - j0,
Ic = 4 ic - 2 i0 + 1,
Jc = 4 jc - 2 i0 + 1,

When no_of_sprite_warping_point == 0,

(F(i, j), G(i, j)) = (s i, s j),
(Fc(ic, jc), Gc(ic, jc)) = (s ic, s jc).

When no_of_sprite_warping_point == 1,

(F(i, j), G(i, j)) = (i0' + s i, j0' + s j),
(Fc(ic, jc), Gc(ic, jc)) = (s ic + i0' /// 2, s jc + j0' /// 2).

When no_of_sprite_warping_points == 2,

(F(i, j), G(i, j)) = (i0' + ((–r i0' + i1'') I + (r j0' – j1'') J) /// (W' r) ,
j0' + ((–r j0' + j1'') I + (–r i0' + i1'') J) /// (W' r)),
(Fc(ic, jc), Gc(ic, jc)) = (((–r i0' + i1 '') Ic + (r j0' – j1'') Jc + 2 W' r i0' – 16W') /// (4 W' r),
((–r j0' + j1'') Ic + (–r i0' + i1'') Jc + 2 W' r j0' – 16W') /// (4 W' r)).

According to the definition of *W'* and *H'* (i.e. $W' = 2^\alpha$ and $H' = 2^\beta$), the divisions by "///" in these functions can be replaced by binary shift operations. By this replacement, the above equations can be rewritten as:

(F(i, j), G(i, j)) = (i0' + (((–r i0' + i1'') I + (r j0' – j1'') J + 2α+ρ-1) >> (α+ρ)) ,
j0' + (((–r j0' + j1'') I + (–r i0' + i1'') J + 2α+ρ-1) >> (α+ρ)),
(Fc(ic, jc), Gc(ic, jc)) = (((–r i0' + i1 '') Ic + (r j0' – j1'') Jc + 2 W' r i0' – 16W' + 2α+ρ+1) >> (α+ρ+2),
((–r j0' + j1'') Ic + (–r i0' + i1'') Jc + 2 W' r j0' – 16W' + 2α+ρ+1) >> (α+ρ+2)),
where 2ρ=r.

When no_of_sprite_warping_points == 3,

(F(i, j), G(i, j)) = (i0' + ((–r i0' + i1'') H' I + (–r i0'+ i2'')W' J) /// (W'H'r),
j0' + ((–r j0' + j1'') H' I + (–r j0'+ j2'')W' J) /// (W'H'r)),
(Fc(ic, jc), Gc(ic, jc)) = (((–r i0' + i1'') H' Ic + (–r i0'+ i2'')W' Jc + 2 W'H'r i0' – 16W'H') /// (4W'H'r),
((–r j0' + j1'') H' Ic + (–r j0'+ j2'')W' Jc + 2 W'H'r j0' – 16W'H') /// (4W'H'r)).

According to the definition of *W'* and *H'*, the computation of these functions can be simplified by dividing the denominator and numerator of division beforehand by *W'* (when *W'* < *H'*) or *H'* (when *W'* ≥ *H'*). As in the case of no_of_sprite_warping_points == 2, the divisions by "///" in these functions can be replaced by binary shift operations. For example, when *W'* ≥ *H'* (i.e. $a \geq b$) the above equations can be rewritten as:

(F(i, j), G(i, j)) = (i0' +(((–r i0' + i1'') I + (–r i0'+ i2'') $2^{\alpha-\beta}$ J + $2^{\alpha+\rho-1}$) >> (α+ρ)),
j0' + (((–r j0' + j1'') I + (–r j0'+ j2'') $2^{\alpha-\beta}$ J + $2^{\alpha+\rho-1}$) >> (α+ρ))),
(Fc(ic, jc), Gc(ic, jc)) = (((–r i0' + i1'') Ic + (–r i0'+ i2'') $2^{\alpha-\beta}$ Jc + 2W'r i0' – 16W' + $2^{\alpha+\rho+1}$) >> (α+ρ+2),
((–r j0' + j1'') Ic + (–r j0'+ j2'') $2^{\alpha-\beta}$ Jc + 2W'r j0' – 16W' + $2^{\alpha+\rho+1}$) >> (α+ρ+2)).

When no_of_sprite_warping_point == 4,

(F(i, j), G(i, j)) = ((a i + b j + c) /// (g i + h j + D W H),
(d i + e j + f) /// (g i + h j + D W H)),
(Fc(ic, jc), Gc(ic, jc)) = ((2 a Ic + 2 b Jc + 4 c – (g Ic + h Jc + 2 D W H) s) /// (4gIc +4 hJc +8D W H),
(2 d Ic + 2 e Jc + 4 f – (g Ic + h Jc + 2 D W H) s) /// (4 g Ic +4 hJc +8D W H))

where

g = ((i0' – i1' – i2' + i3') (j2' – j3') – (i2' – i3') (j0' – j1' – j2' + j3')) H ,
h = ((i1' – i3') (j0' – j1' – j2' + j3') – (i0' – i1' – i2' + i3') (j1' – j3')) W ,
D = (i1' – i3') (j2' – j3') – (i2' – i3') (j1' – j3'),
a = D (i1' – i0') H + g i1' ,
b = D (i2' – i0') W + h i2',
c = D i0' W H,
d = D (j1' – j0') H + g j1',
e = D (j2' – j0') W + h j2',
f = D j0' W H.

The implementor should be aware that a 32bit register may not be sufficient for representing the denominator or the numerator in the above transform functions for affine and perspective transform. The usage of a 64 bit floating point representation should be sufficient in such case.

7.7.6 Sample reconstruction

The reconstructed value *Y* of the luminance sample (*i*, *j*) in the currently decoded VOP shall be defined as

$$Y = ((s - r_j)((s - r_i) Y_{00} + r_i Y_{01}) + r_j ((s - r_i) Y_{10} + r_i Y_{11})) // s^2,$$

where Y_{00}, Y_{01}, Y_{10}, Y_{11} represent the sprite luminance sample at (*F*(*i, j*)////s, *G*(*i, j*)////s), (*F*(*i, j*)////s + 1,*G*(*i, j*)////s), (*F*(*i, j*)////s, *G*(*i, j*)////s + 1), and (*F*(*i, j*)////s + 1,*G*(*i, j*)////s + 1) respectively, and r_i =*F*(*i, j*) –(*F*(*i, j*)////*s*)*s* and r_j =*G*(*i, j*) – (*G*(*i, j*)////*s*)*s*. Figure 7-29 illustrates this process.

In case any of Y_{00}, Y_{01}, Y_{10} and Y_{11} lies outside the sprite luminance binary mask, it shall be obtained by the padding process as defined in section 7.5.1.

When brightness_change_in_sprite == 1, the final reconstructed luminance sample (*i*, *j*) is further computed as *Y* = *Y* * (brightness_change_factor * 0.01 + 1), clipped to the range of [0, 255].

Similarly, the reconstructed value C of the chrominance sample (i_c, j_c) in the currently decoded VOP shall be define as

$$C = ((s - r_j)((s - r_i)\, C_{00} + r_i\, C_{01}) + r_j\, ((s - r_i)\, C_{10} + r_i\, C_{11}))\ //\ s^2,$$

where C_{00}, C_{01}, C_{10}, C_{11} represent the sprite chrominance sample at ($F_c(i_c, j_c)$////s, $G_c(i_c, j_c)$////s), ($F_c(i_c, j_c)$////s + 1, $G_c(i_c, j_c)$////s), ($F_c(i_c, j_c)$////s, $G_c(i_c, j_c)$////s + 1), and ($F_c(i_c, j_c)$////s + 1, $G_c(i_c, j_c)$////s + 1) respectively, and $r_i = F_c(i_c, j_c) - (F_c(i_c, j_c))////s)s$ and $r_j = G_c(i_c, j_c) - (G_c(i_c, j_c))////s)s$. In case any of C_{00}, C_{01}, C_{10} and C_{11} lies outside the sprite chrominance binary mask, it shall be obtained by the padding process as defined in section 7.5.1.

The reconstructed value of luminance binary mask sample $BY(i,j)$ shall be computed following the identical process for the luminance sample. However, corresponding binary mask sample values shall be used in place of luminance samples Y_{00}, Y_{01}, Y_{10}, Y_{11}. Assume the binary mask sample opaque is equal to 255 and the binary mask sample transparent is equal to 0. If the computed value is bigger or equal to 128, $BY(i, j)$ is defined as opaque. Otherwise, $BY\ (i, j)$ is defined as transparent. The chrominance binary mask samples shall be reconstructed by downsampling of the luminance binary mask samples as specified in 7.5.2.

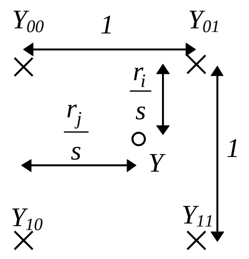


Figure 7-29 Pixel value interpolation (it is assumed that sprite samples are located on an integer grid).

7.7.7 Scalable sprite decoding

The reconstruction of a temporal enhancement VOP follows the clause 7.8.1. If sprite has already been reconstructed for the lower layers, it needs not to be done again. Otherwise, sprite shall be reconstructed from lower layer bitstream following clause 7.8.1.

Spatial enhancement to sprite shall be decoded as described in the clause for spatial scalability. Sprite reference points decoded from lower layer bitstream shall be upsampled according to the spatial enhancement factor. The reconstruction of VOPs follows clause 7.8.2.

7.8 Generalized scalable decoding

This clause specifies the additional decoding process required for decoding scalable coded video.

The scalability framework is referred to as generalized scalability which includes the spatial and the temporal scalabilities. The temporal scalability offers scalability of the temporal resolution, and the spatial scalability offers scalability of the spatial resolution. Each type of scalability involves more than one layer. In the case of two layers, consisting of a lower layer and a higher layer; the lower layer is referred to as the base layer and the higher layer is called the enhancement layer.

In the case of temporal scalability, both rectangular VOPs as well as arbitrary shaped VOPs are supported. In the case of spatial scalability, only rectangular VOPs are supported. Figure 7-30 shows a high level decoder structure for generalized scalability.

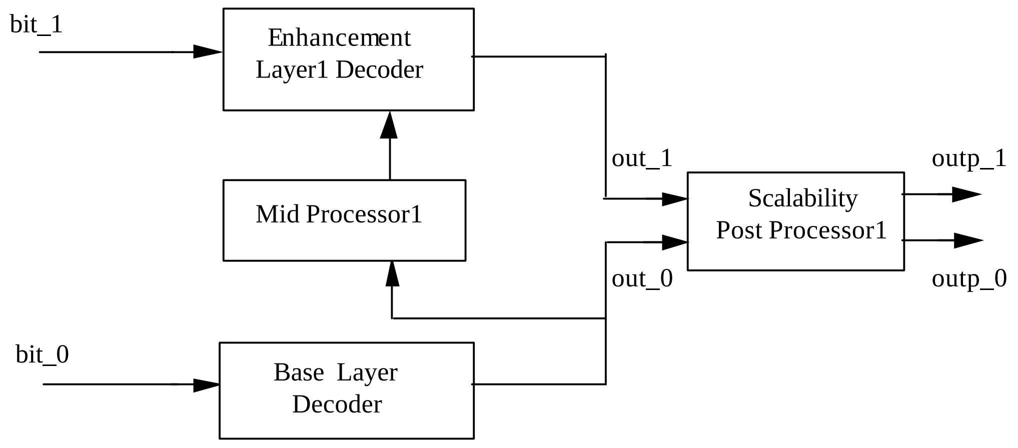


Figure 7-30 High level decoder structure for generalized scalability.

The base layer and enhancement layer bitstreams are input for decoding by the corresponding base layer decoder and enhancement layer decoder.

When spatial scalability is to be performed, mid processor 1 performs spatial up or down sampling of input. The scalability post processor performs any necessary operations such as spatial up or down sampling of the decoded base layer for display resulting at outp_0 while the enhancement layer without resolution conversion may be output as outp_1.

When temporal scalability is to be performed, the decoding of base and enhancement layer bitstreams occurs in the corresponding base and enhancement layer decoders as shown. In this case, mid processor 1 does not perform any spatial resolution conversion. The post processor simply outputs the base layer VOPs without any conversion, but temporally multiplexes the base and enhancement layer VOPs to produce higher temporal resolution enhancement layer.

The reference VOPs for prediction are selected by reference_select_code as specified in Table 7-14 and Table 7-15. In coding of P-VOPs belonging to an enhancement layer, the forward reference is one of the following four: the most recently decoded VOP of enhancement layer, the most recent VOP of the reference layer in display order, the next VOP of the lower layer in display order, or the temporally coincident VOP in the reference layer.

In B-VOPs, the forward reference is one of the following two: the most recently decoded enhancement VOP or the most recent lower layer VOP in display order. The backward reference is one of the following three: the temporally coincident VOP in the lower layer, the most recent lower layer VOP in display order, or the next lower layer VOP in display order.

Table 7-14 Prediction reference choices in enhancement layer P-VOPs for scalability

ref_select_code	forward prediction reference
00	Most recently decoded enhancement VOP belonging to the same layer.
01	Most recently VOP in display order belonging to the reference layer.
10	Next VOP in display order belonging to the reference layer.
11	Temporally coincident VOP in the reference layer (no motion vectors)

Table 7-15 Prediction reference choices in enhancement layer B-VOPs for scalability

ref_select_code	forward temporal reference	backward temporal reference
00	Most recently decoded enhancement VOP of the same layer	Temporally coincident VOP in the reference layer (no motion vectors)
01	Most recently decoded enhancement VOP of the same layer.	Most recent VOP in display order belonging to the reference layer.
10	Most recently decoded enhancement VOP of the same layer.	Next VOP in display order belonging to the reference layer.
11	Most recently VOP in display order belonging to the reference layer.	Next VOP in display order belonging to the reference layer.

7.8.1 Temporal scalability

Temporal scalability involves two layers, a lower layer and an enhancement layer. Both the lower and the enhancement layers process the same spatial resolution. The enhancement layer enhances the temporal resolution of the lower layer and if temporally remultiplexed with the lower layer provides full temporal rate.

7.8.1.1 Base layer and enhancement layer

In the case of temporal scalability, the decoded VOPs of the enhancement layer are used to increase the frame rate of the base layer. Figure 7-31 shows a simplified diagram of the motion compensation process for the enhancement layer using temporal scalability.

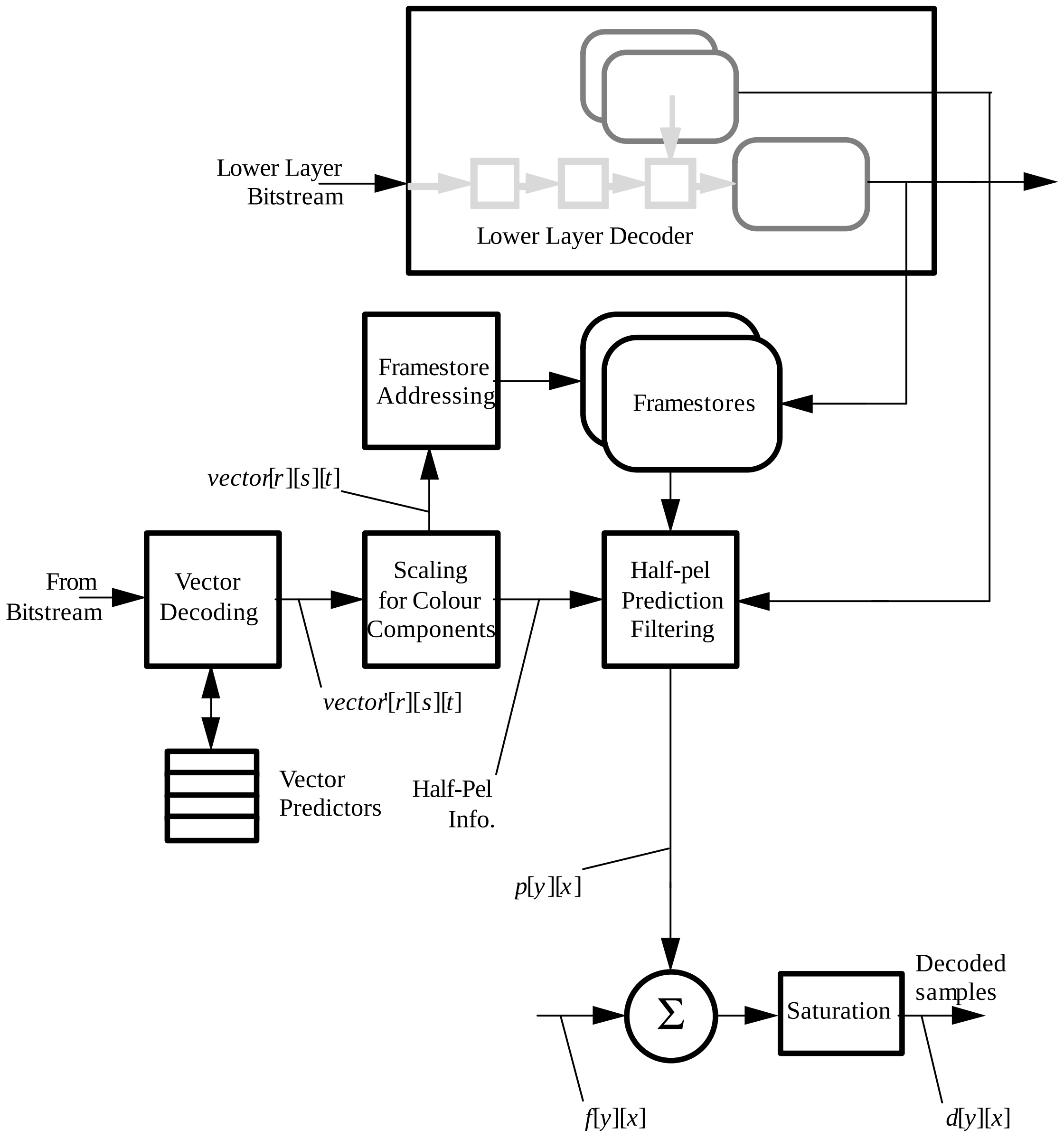


Figure 7-31 Simplified motion compensation process for temporal scalability.

Predicted samples p[y][x] are formed either from frame stores of base layer or from frame stores of enhancement layer. The difference data samples f[y][x] are added to p[y][x] to form the decoded samples d[y][x].

There are two types of enhancement structures indicated by the “enhancement_type” flag. When the value of enhancement_type is “1”, the enhancement layer increases the temporal resolution of a partial region of the base layer. When the value of enhancement_type is “0”, the enhancement layer increases the temporal resolution of an entire region of the base layer.

7.8.1.2 Base layer

The decoding process of the base layer is the same as non-scalable decoding process.

7.8.1.3 Enhancement layer

The VOP of the enhancement layer is decoded as either I-VOP, P-VOP or B-VOP. The shape of the VOP is either rectangular (video_object_layer_id is “00”) or arbitrary (video_object_layer_id is “01”).

7.8.1.3.1 Decoding of I-VOPs

The decoding process of I-VOPs in enhancement layer is the same as non-scalable decoding process.

7.8.1.3.2 Decoding of P-VOPs

The reference layer is indicated by ref_layer_id in Video Object Layer class. Other decoding process is the same as non-scalable P-VOPs except the process specified in 7.8.1.3.4 and 7.8.1.3.5.

For P-VOPs, the ref_select_code is either “00”, “01” or “10”.

When the value of ref_select_code is “00”, the prediction reference is set by the most recently decoded VOP belonging to the same layer.

When the value of ref_select_code is “01”, the prediction reference is set by the previous VOP in display order belonging to the reference layer.

When the value of ref_select_code is “10”, the prediction reference is set by the next VOP in display order belonging to the reference layer.

7.8.1.3.3 Decoding of B-VOPs

The reference layer is indicated by ref_layer_id in Video Object Layer class. Other decoding process is the same as non-scalable B-VOPs except the process specified in 7.8.1.3.4 and 7.8.1.3.5.

For B-VOPs, the ref_select_code is either “01”, “10” or “11”.

When the value of ref_select_code is “01”, the forward prediction reference is set by the most recently decoded VOP belonging to the same layer and the backward prediction reference is set by the previous VOP in display order belonging to the reference layer.

When the value of ref_select_code is “10”, the forward prediction reference is set by the most recently decoded VOP belonging to the same layer, and the backward prediction reference is set by the next VOP in display order belonging to the reference layer.

When the value of ref_select_code is “11”, the forward prediction reference is set by the previous VOP in display order belonging to the reference layer and the backward prediction reference is set by the next VOP in display order belonging to the reference layer. The picture type of the reference VOP shall be either I or P (VOP_coding_type = “00” or “01”).

When the value of ref_select_code is “01” or “10”, direct mode is not allowed. MODB shall always exist in each macroblock, i.e. the macroblock is not skipped even if the co-located macroblock is skipped.

7.8.1.3.4 Decoding of arbitrary shaped VOPs

Prediction for arbitrary shape in P-VOPs or in B-VOPs is formed from a forward reference VOP defined by the value of ref_select_code.

For arbitrary shaped VOPs with the value of enhancement_type being “1”, the shape of the reference VOP is defined as an all opaque rectangle whose size is the same as the reference layer when the shape of reference layer is rectangular (video_object_layer_shape = “00”).

When the value of ref_select_code is “11” and the value of enhancement_type is “1”, MODB shall always exist in each macroblock, i.e. the macroblock is not skipped even if the co-located macroblock is skipped.

7.8.1.3.5 Decoding of backward and forward shape

Backward shape and forward shape are used in the background composition process specified in section 8.1. The backward shape is the shape of the enhanced object at the next VOP in display order belonging to the reference layer. The forward shape is the shape of the enhanced object at the previous VOP in display order belonging to the reference layer.

For the VOPs with the value of enhancement_type being “1”, backward shape is decoded when the load_backward_shape is “1” and forward shape is decoded when load_forward_shape is “1”.

When the value of load_backward_shape is “1” and the value of load_forward_shape is “0”, the backward shape of the previous VOP is copied to the forward shape for the current VOP. When the value of load_backward_shape is “0”, the backward shape of the previous VOP is copied to the backward shape for the current VOP and the forward shape of the previous VOP is copied to the forward shape for the current VOP.

The decoding process of backward and forward shape is the same as the decoding process for the shape of I-VOP with binary only mode (video_object_layer_shape = “10”).

7.8.2 Spatial scalability

7.8.2.1 Base Layer and Enhancement Layer

In the case of spatial scalability, the enhancement bitstream is used to increase the resolution of the image. When the output with lower resolution is required, only the base layer is decoded. When the output with higher resolution is required, both the base layer and the enhancement layer are decoded.

Figure 7-32 is a diagram of the video decoding process with spatial scalability.

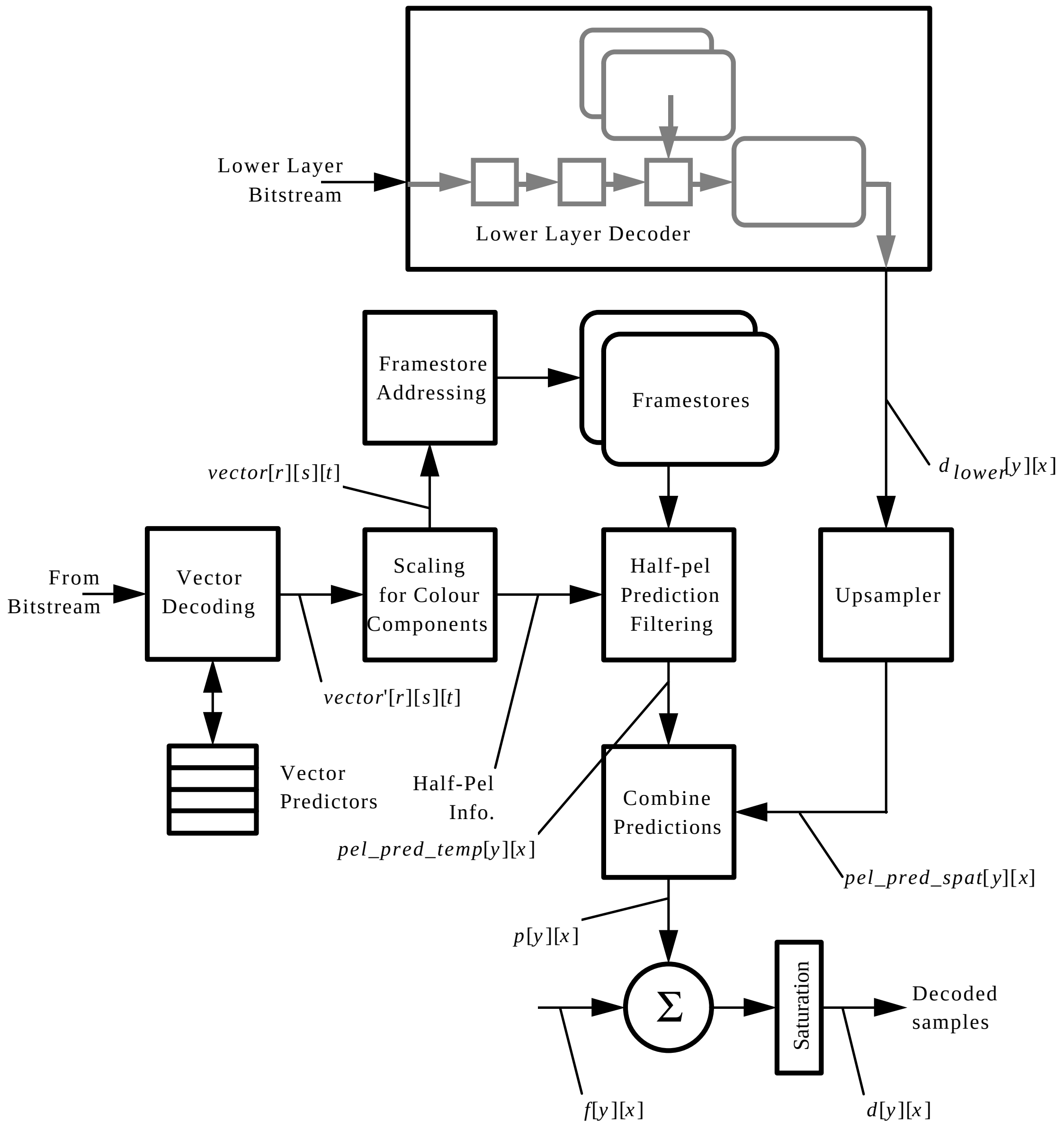


Figure 7-32 Simplified motion compensation process for spatial scalability

7.8.2.2 Decoding of Base Layer

The decoding process of the base layer is the same as nonscalable decoding process.

7.8.2.3 Prediction in the enhancement layer

A motion compensated temporal prediction is made from reference VOPs in the enhancement layer. In addition, a spatial prediction is formed from the lower layer decoded frame (d_{lower}[y][x]). These predictions are selected individually or combined to form the actual prediction.

In the enhancement layer, the forward prediction in P-VOP and the backward prediction in B-VOP are used as the spatial prediction. The reference VOP is set to the temporally coincident VOP in the base layer. The forward prediction in B-VOP is used as the temporal prediction from the enhancement layer VOP. The reference VOP is set to the most recently decoded VOP of the enhancement layer. The interpolate prediction in B-VOP is the combination of these predictions.

In the case that a macroblock is not coded, either because the entire macroblock is skipped or the specific macroblock is not coded there is no coefficient data. In this case f[y][x] is zero, and the decoded samples are simply the prediction, p[y][x].

7.8.2.4 Formation of spatial prediction

Forming the spatial prediction requires definition of the spatial resampling process. The formation is performed at the mid-processor. The resampling process is defined for a whole VOP, however, for decoding of a macroblock, only the 16x16 region in the upsampled VOP, which corresponds to the position of this macroblock, is needed.

The spatial prediction is made by resampling the lower layer reconstructed VOP to the same sampling grid as the enhancement layer. In the first step, the lower layer VOP is subject to vertical resampling. Then, the vertically resampled image is subject to horizontal resampling.

7.8.2.5 Vertical resampling

The image subject to vertical resampling, $d_{lower}[y][x]$, is resampled to the enhancement layer vertical sampling grid using linear interpolation between the sample sites according to the following formula, where vert_pic is the resulting image:

vert_pic[y_h][x] = (16 - phase) * d_{lower} [y1][x] + phase * d_{lower} [y2][x]

where

y_h	=	output sample coordinate in vert_pic
y1	=	(y_h * vertical_sampling_factor_m) / vertical_sampling_factor_n
y2	=	y1 + 1 if y1 < video_object_layer_height - 1
		y1 otherwise
phase	=	(16 * ((y_h * vertical_sampling_factor_m) vertical_sampling_factor_n))
		// vertical_sampling_factor_n

where video_object_layer_width is the width of the reference VOL.

Samples which lie outside the lower layer reconstructed frame which are required for upsampling are obtained by border extension of the lower layer reconstructed frame.

NOTE - The calculation of phase assumes that the sample position in the enhancement layer at y_h = 0 is spatially coincident with the first sample position of the lower layer. It is recognised that this is an approximation for the chrominance component if the chroma_format == 4:2:0.

7.8.2.6 Horizontal resampling

The image subject to horizontal resampling, $vert\_pict[y][x]$, is resampled to the enhancement layer horizontal sampling grid using linear interpolation between the sample sites according to the following formula, where hor_pic is the resulting image:

```
hor_pic[y][xh]   = ((16 - phase) * vert_pic[y][x1] + phase * vert_pic[y][x2]) // 256
where
     xh          =   output sample coordinate in hor_pic
     x1          =   (xh * horizontal_sampling_factor_m) / horizontal_sampling_factor_n
     x2          =   x1 + 1    if x1 < video_object_layer_width - 1
                     x1   otherwise
     phase       =   (16 *  (( xh * horizontal_sampling_factor_m) %
                          horizontal_sampling_factor_n)) // h_subs_n
```

where video_object_layer_width is the width of the reference VOL.

Samples which lie outside the lower layer reconstructed frame which are required for upsampling are obtained by border extension of the lower layer reconstructed frame.

7.8.2.7 Selection and combination of spatial and temporal predictions

The spatial and temporal predictions can be selected or combined to form the actual prediction in B-VOP. The spatial prediction is referred to as “backward prediction”, while the temporal prediction is referred to as “forward prediction”. The combination of these predictions can be used as “interpolate prediction”. In the case of P-VOP, only the spatial prediction (prediction from the lower layer) can be used as the forward prediction. The prediction in the enhancement layer is defined in the following formulae.

```
pel_pred[y][x] = pel_pred_temp[y][x]                               (forward in B-VOP)
pel_pred[y][x] = pel_pred_spat[y][x] = hor_pict[y][x]              (forward in P-VOP and
                                                                      backward in B-VOP)
pel_pred[y][x] = (pel_pred_temp[y][x] + pel_pred_spat[y][x])//2   (Interpolate in B-VOP)
```

pel_pred_temp[y][x] is used to denote the temporal prediction (formed within the enhancement layer). pel_pred_spat[y][x] is used to denote the prediction formed from the lower layer. pel_pred[y][x] is denoted the resulting prediction.

7.8.2.8 Decoding process of enhancement layer

The VOP in the enhancement layer is decoded as either I-VOP, P-VOP or B-VOP.

7.8.2.9 Decoding of I-VOPs

The decoding process of the I-VOP in the enhancement layer is the same as the non_scalable decoding process.

7.8.2.10 Decoding of P-VOPs

In P-VOP, the ref_select_code shall be “11”, i.e., the prediction reference is set to the temporally coincident VOP in the base layer. The reference layer is indicated by ref_layer_id in VideoObjectLayer class. In the case of spatial prediction, the motion vector shall be set to 0 at the decoding process and is not encoded in the bitstream.

A variable length codeword giving information about the macroblock type and the coded block pattern for chrominance is MCBPC. The codewords for MCBPC in the enhancement layer are the same as the base layer and shown in Table 11-6. MCBPC shall be included in coded macroblocks.

The macroblock type gives information about the macroblock and which data elements are present. Macroblock types and included elements in the enhancement layer bitstream are listed in Table 11-1-2.

In the case of the enhancement layer of spatial scalability, INTER4V shall not be used. The macroblock of INTER or INTER+Q is encoded using the spatial prediction.

7.8.2.11 Decoding of B-VOPs

In B-VOP, the ref_select_code shall be “00”, i.e., the backward prediction reference is set to the temporally coincident VOP in the base layer, and the forward prediction reference is set to the most recently decoded VOP in the enhancement layer. In the case of spatial prediction, the motion vector shall be set to 0 at the decoding process and is not encoded in the bitstream.

MODB shall be present in coded macroblocks belonging to B-VOPs. The codeword is the same as the base layer and is shown in Table 11-2. In case MBTYPE does not exist the default shall be set to "Forward MC" (prediction from the last decoded VOP in the same reference layer). MODB shall be encoded in all macroblocks. If its value is equal to ‘0’, further information is not transmitted for this macroblock. The decoder treats the prediction of this macroblock as forward MC with motion vector equal to zero.

MBTYPE is present only in coded macroblocks belonging to B-VOPs. The MBTYPE gives information about the macroblock and which data elements are present. MBTYPE and included elements in the enhancement layer bitstream are listed in Table 11-4.

In the case of the enhancement layer of spatial scalability, direct mode shall not be used. The decoding process of the forward motion vectors are the same as the base layer.

7.9 Still texture object decoding

The block diagram of the decoder is shown in Figure 7-33.

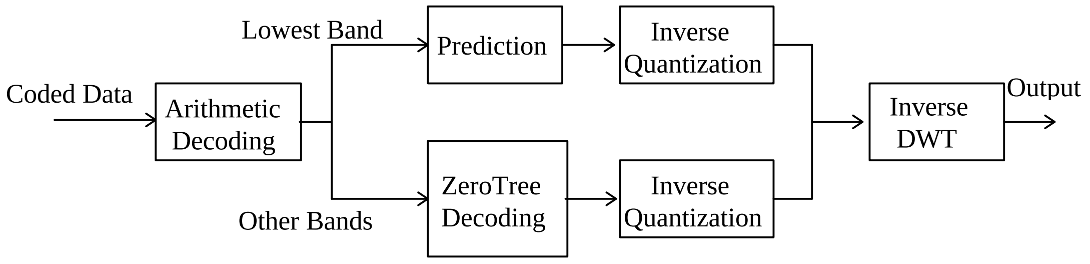


Figure 7-33 Block diagram of the encoder.

The basic modules of a zero-tree wavelet based decoding scheme are as follows:

1. Decoding of the DC subband using a predictive scheme.

2. Arithmetic decoding of the bitstream into quantized wavelet coefficients and the significance map for AC subbands.

3. Zero-tree decoding of the higher subband wavelet coefficients.

4. Inverse quantization of the wavelet coefficients.

5. Composition of the texture using inverse discrete wavelet transform (IDWT).

7.9.1 Decoding of the DC subband

The wavelet coefficients of DC band are decoded independently from the other bands. First the magnitude of the minimum value of the coefficients "band_offset" and the maximum value of the coefficients "band_max_value" are decoded from bitstream. The parameter "band_offset" is negative or zero integer and the parameter "band_max" is a positive integer, so only the magnitude of these parameters are read from the bitstream.

The arithmetic coder model is initialized with a uniform distribution of "band_max_value-band_offset +1" number of symbols. Then, the quantized wavelet coefficients are decoded using the arithmetic decoder in a raster scan order, starting from the upper left coefficient and ending to the lowest right one. The model is updated with decoding of each predicted wavelet coefficient to adopt the probability model to the statistics of DC band.

The "band_offset" is added to all the decoded values, and an inverse predictive scheme is applied. Each of the current coefficients w_X is predicted from three other quantized coefficients in its neighborhood, i.e. w_A, w_B, and w_C (see Figure 7-34), and the predicted value is added to the current decoded coefficient. That is,

```
if   ($|w_A-w_B|$) < $|w_B-w_C|$)
        $\hat{w}_x = w_C$
else
        $\hat{w}_x = w_A$
$w_x = w_x + \hat{w}_x$
```

If any of nodes A, B or C is not in the image, its value is set to zero for the purpose of the inverse prediction. Finally, the inverse quantization scheme is applied to all decoded values to obtain the wavelet coefficients of DC band.

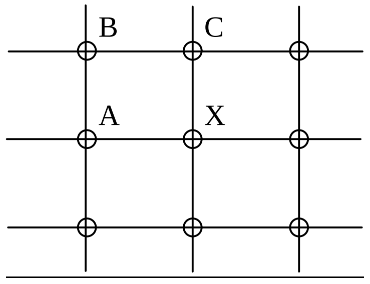


Figure 7-34 DPCM decoding of DC band coefficients

7.9.2 ZeroTree Decoding of the Higher Bands

The zero-tree algorithm is based on the strong correlation between the amplitudes of the wavelet coefficients across scales, and on the idea of partial ordering of the coefficients. The coefficient at the coarse scale is called the *parent*, and all coefficients at the same spatial location, and of similar orientation, at the next finer scale are that parent's children. Figure 7-35 shows a wavelet tree where the parents and the children are indicated by dots and connected by lines. Since the DC subband (shown at the upper left in Figure 7-35) is coded separately using a DPCM scheme, the wavelet trees start from the adjacent higher bands.

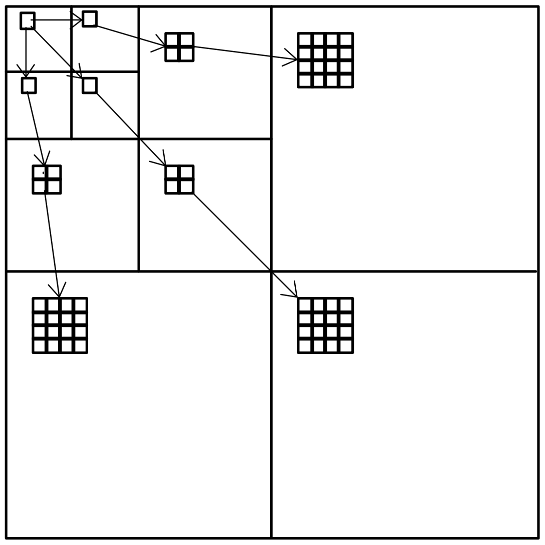


Figure 7-35 Parent-child relationship of wavelet coefficients

In transform-based coding, it is typically true that a large percentage of the transform coefficients are quantized to zero. A substantial number of bits must be spent either encoding these zero-valued quantized coefficients, or else encoding the location of the non-zero-valued quantized coefficients. ZeroTree Coding uses a data structure called a *zerotree*, built on the parent-child relationships described above, and used for encoding the location of non-zero quantized wavelet coefficients. The zerotree structure takes advantage of the principle that if a wavelet coefficient at a coarse scale is “insignificant” (quantized to zero), then all wavelet coefficients of the same orientation at the same spatial location at finer wavelet scales are also likely to be “insignificant”. Zerotrees exist at any tree node where the coefficient is zero and all its descendents are also zero.

The wavelet trees are efficiently represented and coded by scanning each tree from the root in the 3 lowest AC bands through the children, and assigning one of four symbols to each node encountered: *zerotree root (ZTR), value zerotree root (VZTR), isolated zero (IZ)* or *value (VAL)*. A *ZTR* denotes a coefficient that is the root of a zerotree. Zerotrees do not need to be scanned further because it is known that all coefficients in such a tree have amplitude zero. A *VZTR* is a node where the coefficient has a nonzero amplitude, and all four children are zerotree roots. The scan of this tree can stop at this symbol. An *IZ* identifies a coefficient with amplitude zero, but also with some nonzero descendant. A *VAL* symbol identifies a coefficient with amplitude nonzero, and with some nonzero descendant. The symbols and quantized coefficients are losslessly encoded using an adaptive arithmetic coder. Table 7-16 shows the mapping of indices of the arithmetic decoding model into the zerotree symbols:

Table 7-16 The indexing of zerotree symbols

index	**Symbol**
0	IZ
1	VAL
2	ZTR
3	VZTR

In order to achieve a wide range of scalability levels efficiently as needed by different applications, three different zerotree scaaning and associated inverse quantization methods are employed. The encoding mode is speficied in bitstream with quantization_type field as one of 1) single_quant, 2) multi_quant or 3) bilevel_quant:

Table 7-17 The quantization types

code	**quantization_type**
01	single_quant
10	multi _quant
11	bilevel_quant

In single_quant mode, the bitstream contains only one zero-tree map for the wavelet coefficients. After arithmetic decoding, the inverse quantization is applied to obtain the reconstructed wavelet coefficients and at the end, the inverse wavelet transform is applied to those coefficients.

In multi_quant mode, a multiscale zerotree decoding scheme is employed. Figure 7-36 shows the concept of this technique.

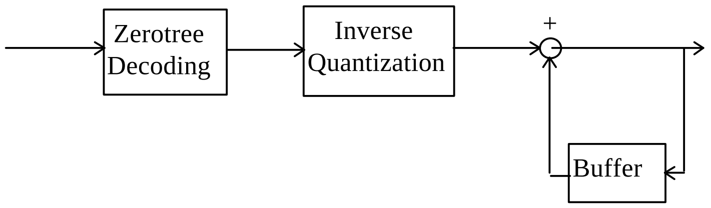


Figure 7-36. Multiscale Zerotree decoding

The wavelet coefficients of the first spatial (and/or SNR) layer are read from the bitstream and decoded using the arithmetic decoder. Zerotree scanning is used for decoding the significant maps and quantized coefficients and locating them in their corresponding positions in trees.. These values are saved in the buffer to be used for quantization refinement at the next scalability layer. Then, an inverse quantization is applied to these indices to obtain the quantized wavelet coefficients. An inverse wavelet transform can also be applied to these coefficients to obtain the first decoded image. The above procedure is applied for the next spatial/SNR layers.

The bilevel_quant mode enables fine granular SNR scalability by encoding the wavelet coefficients in a bitplane by bitplane fashion. This mode uses the same zerotree symbols as the multi_quant mode. In this mode, a zero-tree map is decoded for each bitplane, indicating which wavelet coefficients are nonzero relative to that bitplane. The inverse quantization is also performed bitplane by bitplane. After the zero-tree map, additional bits are decoded to refine the accuracy of the previously decoded coefficients.

7.9.2.1 Zerotree Scanning

In single_quant mode, the wavelet coefficients are scanned in the tree-depth fashion, meaning that all coefficients of each tree is decoded before starting decoding of the next tree. In single_quant mode, the wavelet coefficients are scanned in the tree-depth fashion, meaning that all coefficients of each tree is decoded before starting decoding of the next tree.

Figure 7-37 shows the scanning order for a 16x16 image, with 3 levels of decomposition. In this figure, the indecis 0,1,2,3 represent the DC band coefficients which are decoded separately. The remaining coefficients are decoded in the order shown in this figure. As an example, indices 4,5,..., 24 represent one tree. At first, coefficients in this tree are decoded starting from index 4 and ending at index 24. Then, the coefficients in the second tree are decoded starting from index 25 and ending at 45. The third tree is decoded starting from index 46 and ending at index 66 and so on.

0	1	4	67	5	10	68	73	6	7	11	12	69	70	74	75
2	3	130	193	15	20	78	83	8	9	13	14	71	72	76	77
25	88	46	109	131	136	194	199	16	17	21	22	79	80	84	85
151	214	172	235	141	146	204	209	18	19	23	24	81	82	86	87
26	31	89	94	47	52	110	115	132	133	137	138	195	196	200	201
36	41	99	104	57	62	120	125	134	135	139	140	197	198	202	203
152	157	215	220	173	178	236	241	142	143	147	148	205	206	210	211
162	167	225	230	183	188	246	251	144	145	149	150	207	208	212	213
27	28	32	33	90	91	95	96	48	49	53	54	111	112	116	117
29	30	34	35	92	93	97	98	50	51	55	56	113	114	118	119

37	38	42	43	100	101	105	106	58	59	63	64	121	122	126	127
39	40	44	45	102	103	107	108	60	61	65	66	123	124	128	129
153	154	158	159	216	217	221	222	174	175	179	180	237	238	242	243
155	156	160	161	218	219	223	224	176	177	181	182	239	240	244	245
163	164	168	169	226	227	231	232	184	185	189	190	247	248	252	253
165	166	170	171	228	229	233	234	186	187	191	192	249	250	254	255

Figure 7-37 Scanning order of a wavelet block in the single_quant mode

In multi_quant mode, the wavelet coefficients are decoded in multi scalability layers. In this mode, the wavelet coefficients are scanned in the subband by subband fashion, from the lowest to the highest frequency subbands. Figure 7-38 shows an example of decoding order for a 16x16 image with 3 levels of decomposition..The DC band is located at upper left corner (with indices 0, 1,2, 3) and is decoded separately as described in DC band decoding. The remainig coeffcinets are decoded on the order which is shown in the figure, starting from index 4 and ending at index 255. At first scalability layer, the zerotree symbols and the corresponding values are decoded for the wavelet coefficients of that scalability layer. For the next scalability layers, the zerotree map is updated along with the corresponding value refinements. In each scalability layer, a new zerotree symbol is decoded for a coefficient only if it was decoded as ZTR, VZTR or IZ in previous scalability layer. If the coefficient was decoded as VAL in previous layer, a VAL symbol is also assigned to it at the current layer and only its refinement value is decoded from bitstream.

0	1	4	5	16	17	18	19	64	65	66	67	68	69	70	71
2	3	6	7	20	21	22	23	72	73	74	75	76	77	78	79
8	9	12	13	24	25	26	27	80	81	82	83	84	85	86	87
10	11	14	15	28	29	30	31	88	89	90	91	92	93	94	95
32	33	34	35	48	49	50	51	96	97	98	99	100	101	102	103
36	37	38	39	52	53	54	55	104	105	106	107	108	109	110	111
40	41	42	43	56	57	58	59	112	113	114	115	116	117	118	119
44	45	46	47	60	61	62	63	120	121	122	123	124	125	126	127
128	129	130	131	132	133	134	135	192	193	194	195	196	197	198	199
136	137	138	139	140	141	142	143	200	201	202	203	204	205	206	207
144	145	146	147	148	149	150	151	208	209	210	211	212	213	214	215

152	153	154	155	156	157	158	159	216	217	218	219	220	221	222	223
160	161	162	163	164	165	166	167	224	225	226	227	228	229	230	231
168	169	170	171	172	173	174	175	232	233	234	235	236	237	238	239
176	177	178	179	180	181	182	183	240	241	242	243	244	245	246	247
184	185	186	187	188	189	190	191	248	249	250	251	252	253	254	255

Figure 7-38 Scanning order for multi_qaunt and bilevel_quant modes

In bilevel_quant mode, the band by band scanning is also employed, similar to the multi_quant mode. When bi-level quantization is applied, the coefficients that are already found significant are replaced with zero symbols for the purpose of zero-tree forming in later scans.

7.9.2.2 Entropy Decoding

The zero-tree symbols and quantized coefficient values are all decoded using an adaptive arithmetic decoder and a given symbol alphabet. The arithmetic decoder adaptively tracks the statistics of the zerotree symbols and decoded values. The arithmetic decoder and all models are initialized in the beginning of each color loop. In order to avoid start code emulation, the arithmetic encoder always starts with stuffing one bit ‘1’ at the beginning of the entropy encoding. It also stuffs one bit ‘1’ immediately after it encodes every 22 successive ‘0’s. It stuffs one bit ‘1’ to the end of bitstream in the case in which the last output bit of arithmetic encoder is ‘0’. Thus, the arithmetic decoder reads and discards one bit before starts entropy decoding. During the decoding, it also reads and discards one bit after receiving every 22 successive ‘0’s. The arithmetic decoder reads one bit and discards it if the last input bit to the arithmetic decoder is ‘0’.

In single_quant mode, four adaptive probability models are used to decode the coefficients of the higher bands. These models are: 1) *type* to decode the zero-tree symbols, 2) *root* to decode the values of the nonzero quantized coefficients of the first three AC bands, 3) *valnz* to decode the values of the nonzero quantized coefficients of the all other bands except the ones which are leaves (have no children), and finally 4) *valz* to decode the values of the quantized coefficients of the three highest bands, i.e. the coefficients that have no children. If the number of wavelet decomposition level is one, *valz* is used. If it is two, then *root* and *valz* are used. All above models are initialized with the uniform probability distribution at the beginning and are updated with decoding of each corresponding coefficient by appropriately switching between the models. For each model, the alphabet range is read from bitstream before decoding the wavelet coefficients. This value, max_alphabet, is read from the bitstream in the following format:

extension (1 bit)	value (7 bits)
. . .	. . .

The following scripts shows how **max_alphabet** is decoded:

```
max_alphabet = 0;
count=0;
read (byte);
while ( byte/128){
    max_alphabet += (byte-128) <<(count*7);
    read (byte);
    count++;
}
max_alphabet += (byte-128) <<(count*7);
```

Note that the quantized coefficients which are decoded using *root* or *valnz* models, can not be zero. At the encoder, to increase the coding efficiency, all the values which are coded using these models are decremented by one before encoding.Therefore, at the decoder, after decoding these coefficients, all are incremented by one.The wavelet coefficients are decoded in the order which described in previous section. For each coefficient except the one in the leaf-bands, its zerotree symbol is decoded first and if necessary, then its value is decoded. The value is decoded in two steps. First, its sign is decoded using a binary probability model with '0' meaning positive and '1' meaning negative sign. Then, the absolute value is decoded using the appropriate probability model. The sign model is initialized to the uniform probability distribution. The sign model is initialized to the uniform probability distribution.

For the coefficients which are leaves and are descendent of a VAL or IZ, only a value is decoded using *valz* model. The value is decoded in two steps. First, the absolute value is decoded. Then, if the absolute value is non-zero, its sign is decoded.

In multi_quant mode, one additional probability model, *residual*, is used for decoding the refinements of the coefficients that were decoded with *VAL or VZTR* symbol in any previous scalability layers. If a node is currently not in SKIP mode then we have the following: If in the previous layer, a *VAL* symbol was assigned, the same symbol is kept for the current pass and no zerotree symbol is decoded. If in the previous layer, a *VZTR* symbol was assigned, a new symbol is decoded for the current layer, but it can only be *VAL* or *VZTR*. Similarly, if the nonzero value of a leaf coefficient was decoded in any previous scalability layer, the residual model is used to decode the coefficient refinements. The residual model, same as the other probability models, is also initialized to the uniform probability distribution at the beginning of each scalability layer and color. The numbers of bins for the *residual* model is calculated based on the ratio of the quantization step sizes of the current and previous scalability layers (defined in the inverse quantization section). When a residual model is used, only the magnitude of the refinement are decoded as these values are always zero or positive integers. If the number of wavelet decomposition levels is one, then only models *valz* and residual are used. If it is two, then models *root*, *valz and residual* are used . If a node is in SKIP mode, then its new significant symbol is decoded from bitstream, but no value is decoded for the node and its value in the current scalability layer is assumed to be zero.

For the bi-level quantization mode, the zero-tree map is decoded for each bitplane, indicating which wavelet coefficients are zeros relative to the current quantization step size. Different probability models for the arithmetic decoder are used and updated according to the local contexts. For instance, if a coefficient is a descendant of ZTR in the previous pass, then its probability of being zero in the current layer is significantly higher than in the case where it is the descendant of VZTR. The additional symbols DZ and DV are used for switching the models only, where DZ refers to the descendant of a ZTR symbol, DV refers to the descendant of a VZTR symbol.

After the zero-tree map, additional bits are received to refine the accuracy of the coefficients that are already marked significant by previously received information at the decoder. For each significant coefficient, the 1-bit bi-level quantized refinement values are entropy coded using the arithmetic coder.

In order to avoid start code emulation, the arithmetic encoder stuffs one bit ‘1’ immediately after it encodes 22 successive ‘0’s. It also stuffs one bit ‘1’ to the end of bitstream in the case in which the last output bit of arithmetic encoder is ‘0’. Thus, the arithmetic decoder reads and discards one bit if it receives 22 successive ‘0’s. For the same reason, the arithmetic decoder reads one bit and discards it if the last input bit to the arithmetic decoder is ‘0’.

7.9.3 Inverse Quantization

Different quantization step sizes (one for each color component) are specified for each level of scalability.

The quantizer of the DC band is a uniform mid-rise quantizer with a dead zone equal to the quantization step size. The quantization index is a signed integer number and the quantization reconstructed value is obtained using the following equation:

$$V = id * Qdc,$$

where V is the reconstructed value, id is the decoded index and Qdc is the quantization step size.

All the quantizers of the higher bands (in all quantization modes) are uniform mid-rise quantizer with a dead zone 2 times the quantization step size. For the single quantization mode, the quantization index is an signed integer. The reconstructed value is obtained using the following algorithm:

```
if (id == 0)
    V =0;
else if ( id > 0 )
    V = id*Q+Q/2;
else
    V = id*Q-Q/2;
```

where V is the reconstructed value, id is the decoded index and Q is the quantization step size.

In the multi quantization mode, when the quantization index of a nonzero coefficient is decoded for the first time, it is used to reconstruct the coefficients exactly in same procedure as in single quantization mode (as described above). For successive scalability layers, the reconstructed value is refined. The refinement information are called residuals and are calculated by calculating the number of refinement levels

$$m=ROUND(prevQ/curQ)$$

where, prevQ is the previous layer’s Q value, which may have been revised from the Q value extracted from the bitstream, and curQ is the current layer’s Q value which was extracted from the bitstream, ROUND rounds to the nearest integer except in the case where the nearest integer is zero in which case it is one (i.e. ROUND(x) = MAX(nearest integer of x, 1)). The revision formula for the Q values is:

$$revisedQ = CEIL(prevQ \div m)$$

where CEIL rounds up to the nearest integer. This revisedQ becomes prevQ in the next scalability layer.

For m larger than 1, the inverse range of the quantized value is partitioned from the previous layer in such a way that makes the partitions *as uniform as possible* based on the previously calculated number of quantization refinement levels, m. This partitioning always leaves a discrepancy of zero between the partition sizes if prevQ is evenly divisible by curQ (e.g. prevQ = 20 and curQ = 10). If prevQ is not evenly divisible by curQ (e.g. prevQ = 20 and curQ = 7) then a maximum discrepancy of 1 occurs between partitions (in this case, 7, 7, 6). The larger partitions are always the ones closer to zero. The partitions are indexed from 0 to m-1, starting from the partition closer to zero (in this case, 0, 1, 2 for 7, 7, 6 partitions). These indices are decoded from the bitstream. At the decoder, the midpoint of any partition is assigned as the reconstruction level of that partition (in this example, 23 for the 20-26 partition, 30 for 26-33 partition, and 37 for the 34-39). For m=1, no refinement is decoded from the bitstream.

The reconstruction formula is:

PrevLevStart + (sign)(revisedQ)//2

Where PrevLevStart is the value of lowest magnitude of the previous quantization level, sign is the polarity of the value being quantized, revisedQ is the value described above, and // is integer division. Note: PrevLevStart and sign are known from previous scalability layers.

In the bilevel_quant mode, quant and SNR_scalability_levels are also defined in the bitstream. The initial quantization step size is calculated using the following equation:

Q0 = quant * (1<< SNR_scalability_levels)

The quantization step size at each successive bitplane is half of that at previous bitplane. These quantizers are also uniform mid-rise quantizers with dead zones 2 times the quantization step sizes. The wavelet coefficients are reconstructed as described in the mulit_quant mode for the case in which m=2.

7.9.3.1 Shape adaptive zerotree decoding

Decoding shape adaptive wavelet coefficients is the same as decoding regular wavelet coefficients except keep track of the locations of where to put the decoded wavelet coefficients according to the shape information. Similar to decoding of regular wavelet coefficients, the decoded zerotree symbols at a lower subband are used to determine whether decoding is needed at higher subbands. The difference is now that some zerotree nodes correspond to the pixel locations outside the shape boundary and no bits are to be decoded for these out_nodes. Root layer is defined as the lowest three AC subbands, leaf layer is defined as the highest three AC subbands. For decomposition level of one, the overlapped root layer and leaf laver shall be treated as leaf layer. The following description for shape adaptive zerotree decoding is the decoding process in the single quantization mode.

7.9.3.1.1 Root layer

At the root layer (the lowest 3 AC bands), the shape information is examined for every node to determine whether a node is an out_node.

If it is an out_node,

- no bits are decoded for this node;
- the four children nodes of this node are marked “to_be_decoded” (TBD);

otherwise,

- a zerotree symbol is decoded for this node using an adaptive arithmetic decoder.

If the decoded symbol for the node is either isolated_zero (IZ) or value (VAL),

- the four children nodes of this node are marked TBD;

otherwise,

- the symbol is either zerotree_root (ZTR) or valued_zerotree_root (VZTR) and the four children nodes of this node are marked “no_code” (NC).

If the symbol is VAL or VZTR,

- a non-zero wavelet coefficient is decoded for this node by root model;

otherwise,

- the symbol is either IZ or ZTR and the wavelet coefficient is set to zero for this node.

7.9.3.1.2 Between root and leaf layer

At any layer between the root layer and the leaf layer, the shape information is examined for every node to determine whether a node is an out_node.

If it is an out_node,

- no bits are decoded for this node;
- the four children nodes of this node are marked as either TBD or NC depending on whether this node itself is marked TBD or NC respectively;

otherwise, if it is marked NC,

- no bits are decoded for this node;
- the wavelet coefficient is set to zero for this node;
- the four children nodes are marked NC;

otherwise,

- a zerotree symbol is decoded for this node using an adaptive arithmetic decoder.

If the decoded symbol for the node is either isolated_zero (IZ) or value (VAL),

- the four children nodes of this node are marked TBD;

otherwise,

- the symbol is either zerotree_root (ZTR) or valued_zerotree_root (VZTR) and the four nodes of this node are marked “no_code” (NC).

If the symbol is VAL or VZTR,

- a non-zero wavelet coefficient is decoded for this node by valnz model;

otherwise,

- the symbol is either IZ or ZTR and the wavelet coefficient is set to zero for this node.

7.9.3.1.3 Leaf layer

At the leaf layer, the shape information is examined for every node to determine whether a node is an out_node.

If it is an out_node,

- no bits are decoded for this node;

otherwise, if it is marked NC,

- no bits are decoded for this node;
- the wavelet coefficient is set to zero for this node;

otherwise,

- * a wavelet coefficient is decoded for this node by valz adaptive arithmetic model;

7.9.3.2 Shape decomposition

The shape information for both shape adaptive zerotree decoding and the inverse shape adaptive wavelet transform is obtained by decomposing the reconstructed shape from the shape decoder. Assuming binary shape with 0 or 1 indicating a pixel being outside or inside the arbitrarily shaped object, the shape decomposition procedure can be described as follows:

1. For each horizontal line, collect all even-indexed shape pixels together as the shape information for the horizontal low-pass band and collect all odd-indexed shape pixels together as the shape information for the horizontal high-pass band, except for the special case where the number of consecutive 1's is one.
2. For an isolated 1 in a horizontal line, whether at an even-indexed location or at an odd-indexed location, it is always put together with the shape pixels for the low-pass band and a 0 is put at the corresponding position together with the shape pixels for the high-pass band.
3. Perform the above operations for each vertical line after finishing all horizontal lines.
4. Use the above operations to decompose the shape pixels for the horizontal and vertical low-pass band further until the number of decomposition levels is reached.

7.10 Mesh object decoding

An overview of the decoding process is show in Figure 7-39.

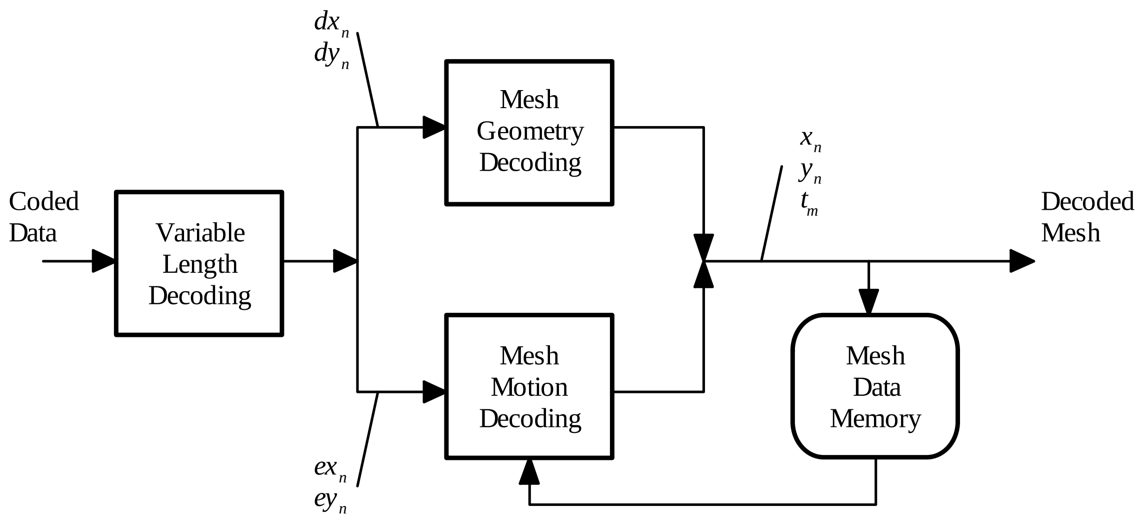


Figure 7-39 Simplified 2D Mesh Object Decoding Process

Variable length decoding takes the coded data and decodes either node point location data or node point motion data. Node point location data is denoted by dx_n, dy_n and node point motion data is denoted by ex_n, ey_n, where n is the node point index (n = 0, ..., N-1). Next, either mesh geometry decoding or mesh motion decoding is applied. Mesh geometry decoding computes the node point locations from the location data and reconstructs a triangular mesh from the node point locations. Mesh motion decoding computes the node point motion vectors from the motion data and applies these motion vectors to the node points of the previous mesh to reconstruct the current mesh.

The reconstructed mesh is stored in the mesh data memory, so that it may be used by the motion decoding process for the next mesh. Mesh data consists of node point locations (x_n, y_n) and triangles t_m, where m is the triangle index (m = 0, ..., M-1) and each triangle t_m contains a triplet <i, j, k> which stores the indices of the node points that form the three vertices of that triangle.

After the mesh_object_start_code has been decoded, a sequence of mesh object planes is decoded, until a mesh_object_end_code is detected. The new_mesh_flag of the mesh object plane class determines whether the data that follows specifies the initial geometry of a new dynamic mesh, or that it specifies the motion of the previous mesh to the current mesh, in a sequence of meshes. Firstly, the decoding of mesh geometry is described; then, the decoding of mesh motion is described. In this specification, a pixel-based coordinate system is assumed, with the x-axis points to the right from the origin, and the y-axis points down from the origin.

7.10.1 Mesh geometry decoding

Since the initial 2D triangular mesh is either a uniform mesh or a Delaunay mesh, the mesh triangular structure (i.e. the connections between node points) is not coded explicitly. Only a few parameters are coded for the uniform mesh; only the 2D node point coordinates $\vec{p}_n = (x_n, y_n)$ are coded for the Delaunay mesh. In each case, the coded information defines the triangular structure of the mesh implicitly, such that it can be computed uniquely by the decoder. The mesh_type_code specifies whether the initial mesh is uniform or Delaunay.

7.10.1.1 Uniform mesh

In the case of a uniform mesh, five parameters are used to specify the complete triangular structure and node point locations. A 2D uniform mesh can be thought of as consisting of a set of rectangles, where each rectangle in turn consists of two triangles. An example of a 2D uniform mesh is given in Figure 7-40; in this example, the nr_mesh_nodes_hor is equal to 5 and nr_mesh_nodes_vert is equal to 4 and triangle_split_code equal to ‘00’. The meaning of mesh_rect_size_hor and mesh_rect_size vert is indicated by the arrows.

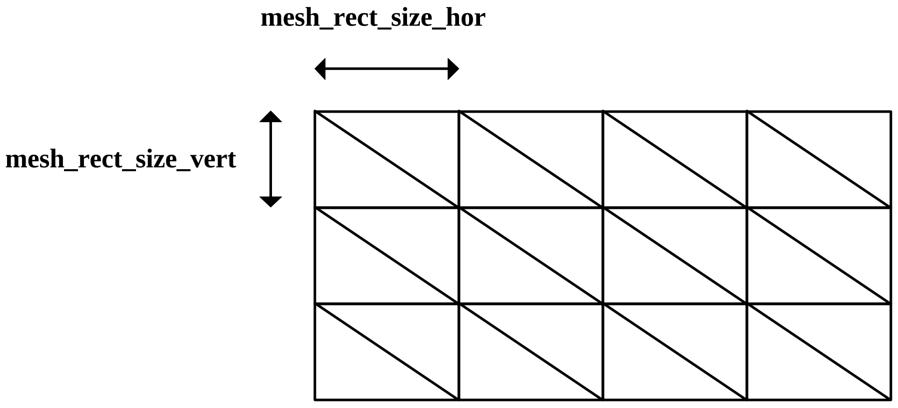
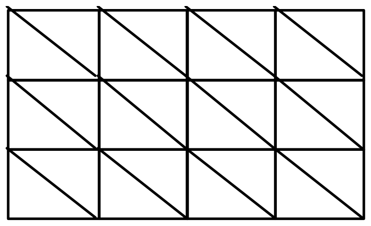
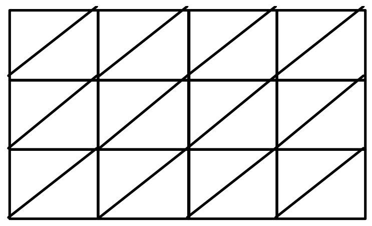
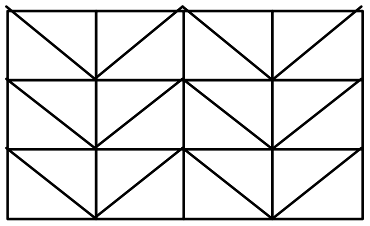


Figure 7-40 Specification of a uniform 2D mesh

In the case of a uniform mesh, the top-left node point of the initial mesh coincides with the origin of a local coordinate system. The first two decoded parameters specify the number of nodes in the horizontal, resp. vertical direction of the uniform mesh. The next two decoded parameters specify the horizontal, resp. vertical size of each rectangle (containing two triangles) in half pixel units. This specifies the layout and dimensions of the mesh. The last parameter specifies how each rectangle is split to form two triangles; four types are allowed as illustrated in Figure 7-41.


triangle_split_code == ‘00’


triangle_split_code == ‘01’


triangle_split_code == ‘10’

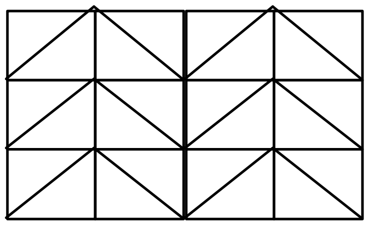

triangle_split_code == ‘11’

Figure 7-41 Illustration of the types of uniform meshes defined.

7.10.1.2 Delaunay mesh

First, the total number of node points in the mesh N is decoded; then, the number of node points that are on the boundary of the mesh N_b is decoded. Note that N is the sum of the number of nodes in the interior of the mesh, N_i and the number of nodes on the boundary, N_b,

$$N = N_i + N_b \ .$$

Now, the locations of boundary and interior node points are decoded, where we assume the origin of the local coordinate system is at the top left of the bounding box surrounding the initial mesh. The x-, resp. y-coordinate of the first node point, $\vec{p}_0 = (x_0, y_0)$, is decoded directly, where x_0 and y_0 are specified w.r.t. to the origin of the local coordinate system. All the other node point coordinates are computed by adding a dx_n, resp. dy_n value to, resp. the x- and y-coordinate of the previously decoded node point. Thus, the coordinates of the initial node point $\vec{p}_0 = (x_0, y_0)$ is decoded as is, whereas the coordinates of all other node points , $\vec{p}_n = (x_n, y_n)$, $n = 1, ..., N - 1$, are obtained by adding a decoded value to the previously decoded node point coordinates:

$$x_n = x_{n-1} + dx_n \quad \text{and} \quad y_n = y_{n-1} + dy_n \ .$$

The ordering in the sequence of decoded locations is such that the first N_b locations correspond to boundary nodes. Thus, after receiving the first N_b locations, the decoder is able to reconstruct the boundary of the mesh by connecting each pair of successive boundary nodes, as well as the first and the last, by straight-line edge segments. The next $N - N_b$ values in the sequence of decoded locations correspond to interior node points. Thus, after receiving N nodes, the locations of both the boundary and interior nodes can be reconstructed, in addition to the polygonal shape of the boundary. This is illustrated with an example in Figure 7-42.

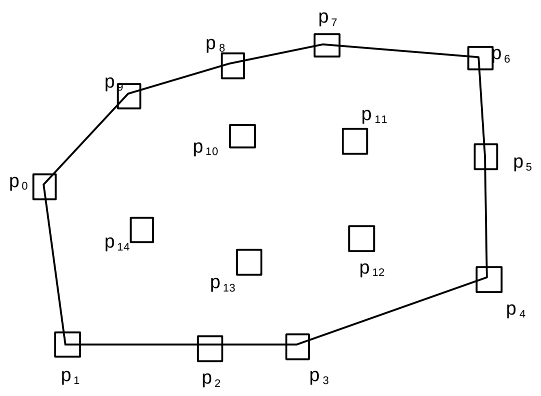


Figure 7-42 Decoded node points and mesh boundary edge.

The mesh is finally obtained by applying constrained Delaunay triangulation to the set of decoded node points, where the polygonal mesh boundary is used as a constraint. A constrained triangulation of a set of node points $\vec{p}_n$ contains the line segments between successive node points on the boundary as edges and contains triangles only in the interior of the region defined by the boundary. Each triangle $t_k \quad \left\langle p_l, \vec{\ } \ , \vec{p}_n \right\rangle$

t_k does not contain in its interior any node point $\vec{p}_r$ visible from all three vertices of t_k. A node point is visible from another node point if a straight line drawn between them falls entirely inside or exactly on the constraining polygonal boundary. The Delaunay triangulation process is defined as any algorithm that is equivalent to enumerating all Delaunay triangles as defined above and inserting them into the mesh. An example of a mesh obtained by constrained triangulation of the node points of Figure 7-42 is shown in Figure 7-43.

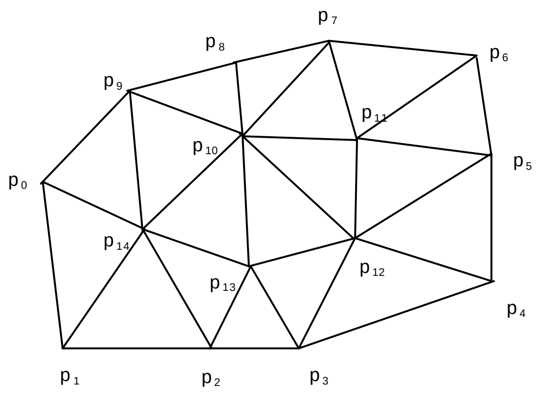


Figure 7-43 Decoded triangular mesh obtained by constrained Delaunay triangulation

7.10.2 Decoding of mesh motion vectors

Each node point $\vec{p}_n$ of a 2D Mesh Object Plane numbered *k* in the sequence of Mesh Object Planes has a 2D motion vector $\vec{v}_n = (vx_n, vy_n)$, defined from Mesh Object Plane *k* to *k*+1. By decoding these motion vectors, one is able to reconstruct the locations of node points in Mesh Object Plane numbered *k*+1. The triangular topology of the mesh remains the same throughout the sequence. Node point motion vectors are decoded according to a predictive method, i.e., the components of each motion vector are predicted using the components of already decoded motion vectors of other node points.

7.10.2.1 Motion vector prediction

To decode the motion vector of a node point $\vec{p}_n$ that is part of a triangle $t_k = \langle \vec{p}_l, \vec{p}_m, \vec{p}_n \rangle$, where the two motion vectors vectors $\vec{v}_l$ and $\vec{v}_m$ of the nodes $\vec{p}_l$ and $\vec{p}_m$ have already been decoded, one can use the values of $\vec{v}_l$ and $\vec{v}_m$ to predict $\vec{v}_n$ and add the prediction vector to a decoded prediction error vector. Starting from an initial triangle t_k of which all three node motion vectors have been decoded, there must be at least one other, neighboring, triangle t_w that has two nodes in common with t_k. Since the motion vectors of the two nodes that t_k and t_w have in common have already been decoded, one can use these two motion vectors to predict the motion vector of the third node in t_w. The actual prediction vector $\vec{w}_n$ is computed by averaging of the two prediction motion vectors and the components of the prediction vector are rounded to half-pixel accuracy, as follows:

$$\vec{w}_n = 0.5 \times \left(\left\lfloor \vec{v}_m + \vec{v}_l + 0.5 \right\rfloor \right),$$

$$\vec{v}_n = \vec{w}_n + \vec{e}_n .$$

Here, $\vec{e}_n = (ex_n, ey_n)$ denotes the prediction error vector, the components of which are decoded from variable length codes. This procedure is repeated while traversing the triangles and nodes of the mesh, as explained below. While visiting all triangles of the mesh, the motion vector data of each node is decoded from the bitstream one by one. Note that no prediction is used to code the first motion vector,

$$\vec{v}_{n_0} = \vec{e}_{n_0} ,$$

and that only the first coded motion vector is used as a predictor to code the second motion vector,

$$\vec{v}_{n_1} = \vec{v}_{n_0} + \vec{e}_{n_1} .$$

Note further that the prediction error vector is specified only for node points with a nonzero motion vector. For all other node points, the motion vector is simply $\vec{v}_n = (0,0)$.

7.10.2.2 Mesh traversal

We use a *breadth-first traversal* to order all the triangles and nodes in the mesh numbered *k*, and to decode the motion vectors defined from mesh *k* to *k*+1. The breadth-first traversal is determined uniquely by the topology and geometry of the mesh. The breadth-first traversal of the mesh triangles is defined as follows (see for an illustration).

First, define the *initial triangle* as follows. Define the top left mesh node as the node *n* with minimum $x_n + y_n$, assuming the origin of the local coordinate system is at the top left. If there is more than one node with the same value of $x_n + y_n$, then choose the node point among these with minimum *y*. The initial triangle is the triangle that contains the edge between the top-left node of the mesh and the next clockwise node on the boundary. Label the initial triangle with the number 0.

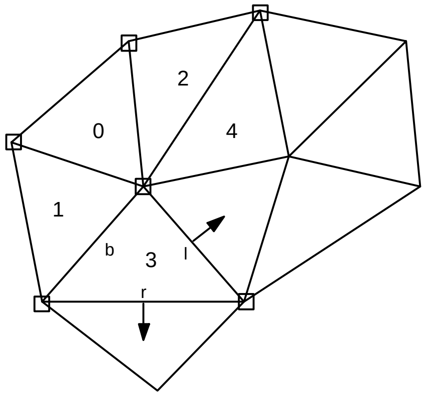
Next, all other triangles are iteratively labeled with numbers 1, 2, ..., *M* - 1, where *M* is the number of triangles in the mesh, as follows.

> Among all labeled triangles that have adjacent triangles which are not yet labeled, find the triangle with the lowest number label. This triangle is referred to in the following as the *current triangle*. Define the *base edge* of this triangle as the edge that connects this triangle to the already labeled neighboring triangle with the lowest number. In the case of the initial triangle, the base edge is defined as the edge between the top-left node and the next clockwise node on the boundary. Define the *right edge* of the current triangle as the next counterclockwise edge of the current triangle with respect to the base edge; and define the *left edge* as the next clockwise edge of the current triangle with respect to the base edge. That is, for a triangle $t_k = \left\langle \vec{p}_l, \vec{p}_m, \vec{p}_n \right\rangle$, where the vertices are in clockwise order, if $\left\langle \vec{p}_l \vec{p}_m \right\rangle$ is the base edge, then $\left\langle \vec{p}_l \vec{p}_n \right\rangle$ is the right edge and $\left\langle \vec{p}_m \vec{p}_n \right\rangle$ is the left edge.
>
> Now, check if there is an unlabeled triangle adjacent to the current triangle, sharing the right edge. If there is such a triangle, label it with the next available number. Then check if there is an unlabeled triangle adjacent to the current triangle, sharing the left edge. If there is such a triangle, label it with the next available number.

This process is continued iteratively until all triangles have been labeled with a unique number *m*.

The ordering of the triangles according to their assigned label numbers implicitly defines the order in which the motion vector data of each node point is decoded, as described in the following. Initially, motion vector data for the top-left node of the mesh is retrieved from the bitstream. No prediction is used for the motion vector of this node, hence this data specifies the motion vector itself. Then, motion vector data for the second node, which is the next clockwise node on the boundary w.r.t. the top-left node, is retrieved from the bitstream. This data contains the prediction error for the motion vector of this node, where the motion vector of the top-left node is used as a prediction. Mark these first two nodes (that form the base edge of the initial triangle) with the label 'done'.

Next, process each triangle as determined by the label numbers. For each triangle, the base edge is determined as defined above. The motion vectors of the two nodes of the base edge of a triangle are used to form a prediction for the motion vector of the third node of that triangle. If that third node is not yet labeled 'done', motion vector data is retrieved and used as prediction error values, i.e. the decoded values are added to the prediction to obtain the actual motion vector. Then, that third node is labeled 'done'. If the third note is already labeled 'done', then it is simply ignored and no data is retrieved. Note that due to the ordering of the triangles as defined above, the two nodes on the base edge of a triangle are guaranteed to be labeled 'done' when that triangle is processed, signifying that their motion vectors have already been decoded and may be used as predictors.



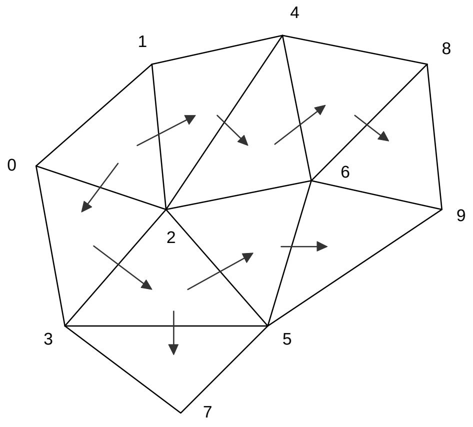


Figure 7-44 Breadth-first traversal of a 2D triangular example mesh.

In Figure 7-44 an example is shown of breadth-first traversal. On the left, the traversal is halfway through the mesh - five triangles have been labeled (with numbers) and the motion vectors of six node points have been decoded (marked with a box symbol). The triangle which has been labeled '3' is the 'current triangle'; the base edge is 'b'; the right and left edge are 'r' and 'l'. The triangles that will be labeled next are the triangles sharing the right, resp. left edge with the current triangle. After those triangles are labeled, the triangle which has been labeled '4' will be the next 'current triangle' and another motion vector will be decoded. On the right, the traversed 2D triangular mesh is shown, illustrating the transitions between triangles and final order of node points according to which respective motion vectors are decoded.

7.11 Face object decoding

7.11.1 Frame based face object decoding

This clause specifies the additional decoding process required for face object decoding.

The coded data is decoded by an arithmetic decoding process. The arithmetic decoding process is described in detail in Annex B. Following the arithmetic decoding, the data is de-quantized by an inverse quantization process. The FAPs are obtained by a predictive decoding scheme as shown in Figure 7-45.

The base quantization step size QP for each FAP is listed in Table 12-1. The quantization parameter FAP_QUANT is applied uniformly to all FAPs. The magnitude of the quantization scaling factor ranges from 1 to 8. The value of FAP_QUANT == 0 has a special meaning, it is used to indicate lossless coding mode, so no dequantization is applied. The quantization stepsize is obtained as follows:

```
if (FAP_QUANT)
    qstep = QP * FAP_QUANT
else
    qstep = 1
```

The dequantized FAP'(t) is obtained from the decoded coefficient FAP''(t) as follows:

FAP'(t) = qstep * FAP''(t)

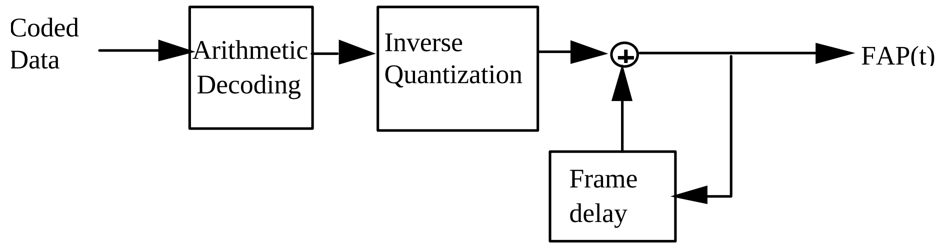


Figure 7-45 FAP decoding

7.11.1.1 Decoding of faps

For a given frame FAPs in the decoder assume one of three of the following states:

1. set by a value transmitted by the encoder
2. retain a value previously sent by the encoder
3. interpolated by the decoder

FAP values which have been initialized in an intra coded FAP set are assumed to retain those values if subsequently masked out unless a special mask mode is used to indicate interpolation by the decoder. FAP values which have never been initialized must be estimated by the decoder. For example, if only FAP group 2 (inner lip) is used and FAP group 8 (outer lip) is never used, the outer lip points must be estimated by the decoder. In a second example the FAP decoder is also expected to enforce symmetry when only the left or right portion of a symmetric FAP set is received (e.g. if the left eye is moved and the right eye is subject to interpolation, it is to be moved in the same way as the left eye).

7.11.2 DCT based face object decoding

The bitstream is decoded into segments of FAPs, where each segment is composed of a temporal sequence of 16 FAP object planes. The block diagram of the decoder is shown in Figure 7-46.

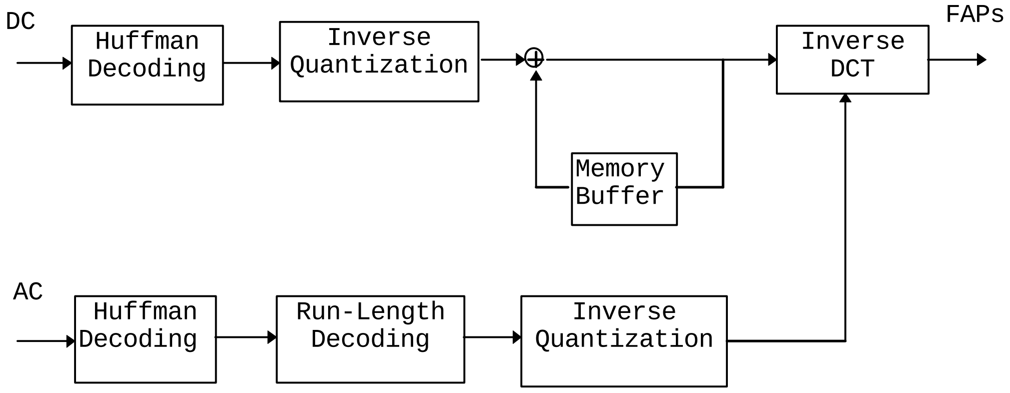


Figure 7-46 Block diagram of the DCT-based decoding process.

The DCT-based decoding process consists of the following three basic steps:

1. Differential decoding the DC coefficient of a segment.
2. Decoding the AC coefficients of the segment
3. Determining the 16 FAP values of the segment using inverse discrete cosine transform (IDCT).

A uniform quantization step size is used for all AC coefficients. The quantization step size for AC coefficients is obtained as follows:

qstep[i] = fap_scale[fap_quant_inex] * DCTQP[i]

where DCTQP[i] is the base quantization step size and its value is defined in Section 6.3.11.10. The quantization step size of the DC coefficient is one-third of the AC coefficients. Different quantization step sizes are used for different FAPs.

The DCT-based decoding process is applied to all FAP segments except the viseme (FAP #1) and expression (FAP #2) parameters. The latter two parameters are differential decoded without transform. The decoding of viseme and expression segments are described at the end of this section.

For FAP #3 to FAP #68, the DC coefficient of an intra coded segment is stored as a 16-bit signed integer if its value is within the 16-bit range. Otherwise, it is stored as a 31-bit signed integer. For an inter coded segment, the DC coefficient of the previous segment is used as a prediction of the current DC coefficient. The prediction error is decoded using a Huffman table of 512 symbols. . An "ESC" symbol, if obtained, indicates that the prediction error is out of the range [-255, 255]. In this case, the next 16 bits extracted from the bitstream are represented as a signed 16-bit integer for the prediction error. If the value of the integer is equal to -256*128, it means that the value of the prediction error is over the 16-bit range. Then the following 32 bits from the bitstream are extracted as a signed 32-bit integer, in twos complement format and the most significant bit first

The AC coefficients, for both inter and intra coded segments, are decoded using Huffman tables. The run-length code indicates the number of leading zeros before each non-zero AC coefficient. The run-length ranges from 0 to 14 and proceeds the code for the AC coefficient. The symbol 15 in the run length table indicates the end of non-zero symbols in a segment. Therefore, the Huffman table of the run-length codes contains 16 symbols. The values of non-zero AC coefficients are decoded in a way similar to the decoding of DC prediction errors but with a different Huffman table.

The bitstreams corresponding to viseme and expression segments are basically differential decoded without IDCT. For an intra coded segment, the quantized values of of the first viseme_select1, viseme_select2, viseme_blend, expression_select1, expression_select2, expression_intensity1, and expression_intensity2 within the segment are decoded using fixed length code. These first values are used as the prediction for the second viseme_select1, viseme_select2, … etc of the segment and the prediction error are differential decoded using Huffman tables. For an inter coded segment, the last viseme_select1, for example, of the previous decoded segment is used to predict the first viseme_select1 of the current segment. In general, the decoded values (before inverse quantization) of differential coded viseme and expression parameter fields are obtained

byviseme_segment_select1Q[k] = viseme_segment_select1Q[k-1] + viseme_segment_select1Q_diff[k] - 14
viseme_segment_select2Q[k] = viseme_segment_select2Q[k-1] + viseme_segment_select2Q_diff[k] - 14
viseme_segment_blendQ[k] = viseme_segment_blendQ[k-1] + viseme_segment_blendQ_diff[k] - 63
expression_segment_select1Q[k] = expression_segment_select1Q[k-1] + expression_segment_select1Q_diff[k] - 6
expression_segment_select2Q[k] = expression_segment_select2Q[k-1] + expression_segment_select2Q_diff[k] - 6
expression_segment_intensity1Q[k] = expression_segment_intensity1Q[k-1] + expression_segment_intensity1Q_diff[k] - 63
expression_segment_intensity2Q[k] = expression_segment_intensity2Q[k-1] + expression_segment_intensity2Q_diff[k] - 63

7.11.3 Decoding of the viseme parameter fap 1

Fourteen visemes have been defined for selection by the Viseme Parameter FAP 1, the definition is given in Annex C. The viseme parameter allows two visemes from a standard set to be blended together. The viseme parameter is composed of a set of values as follows.

viseme () {	**Range**
viseme_select1	0-14
viseme_select2	0-14
viseme_blend	0-63
viseme_def	0-1
}	

Viseme_blend is quantized (step size = 1) and defines the blending of viseme1 and viseme2 in the decoder by the following symbolic expression where viseme1 and 2 are graphical interpretations of the given visemes as suggested in the non-normative annex.

final viseme = (viseme 1) * (viseme_blend / 63) + (viseme 2) * (1 - viseme_blend / 63)

The viseme can only have impact on FAPs that are currently allowed to be interpolated.

If the viseme_def bit is set, the current mouth FAPs can be used by the decoder to define the selected viseme in terms of a table of FAPs. This FAP table can be used when the same viseme is invoked again later for FAPs which must be interpolated.

7.11.4 Decoding of the viseme parameter fap 2

The expression parameter allows two expressions from a standard set to be blended together.The expression parameter is composed of a set of values as follows.

expression () {	**Range**
expression_select1	0-6
expression_intensity1	0-63
expression_select2	0-6
expression_intensity2	0-63
init_face	0-1
expression_def	0-1
}	

Expression_intensity1 and expression_intensity2 are quantized (step size = 1) and define excitation of expressions 1 and 2 in the decoder by the following equations where expressions 1 and 2 are graphical interpretations of the given expression as suggested by the non-normative reference:

final expression = expression1 * (expression_intensity1 / 63)+ expression2 * (expression_intensity2 / 63)

The decoder displays the expressions according to the above fomula as a superposition of the 2 expressions.

The expression can only have impact on FAPs that are currently allowed to be interpolated. If the init_face bit is set, the neutral face may be modified within the neutral face constraints of mouth closure, eye opening, gaze direction, and head orientation before FAPs 3-68 are applied. If the expression_def bit is set, the current FAPs can be used to define the selected expression in terms of a table of FAPs. This FAP table can then be used when the same expression is invoked again later.

7.11.5 Fap masking

The face is animated by sending a stream of facial animation parameters. FAP masking, as indicated in the bitstream, is used to select FAPs. FAPs are selected by using a two level mask hierarchy. The first level contains two bit code for each group indicating the following options:

1. no FAPs are sent in the group.
2. a mask is sent indicating which FAPs in the group are sent. FAPs not selected by the group mask retain their previous value if any previously set value (not interpolated by decoder if previously set)
3. a mask is sent indicating which FAPs in the group are sent. FAPs not selected by the group mask retain must be interpolated by the decoder.
4. all FAPs in the group are sent.

7.12 Output of the decoding process

This section describes the output of the theoretical model of the decoding process that decodes bitstreams conforming to this specification.

The visual decoding process input is one or more coded visual bitstreams (one for each of the layers). The visual layers are generally multiplexed by the means of a system stream that also contains timing information.

7.12.1 Video data

The output of the video decoding process is a series of VOPs that are normally the input of a display process. The order in which fields or VOPs are output by the decoding process is called the display order, and may be different from the coded order (when B-VOPs are used).

7.12.2 2D Mesh data

The output of the decoding process is a sequence of meshes, defined for each time instant, a series of one or more mesh object planes. The meshes are normally inpout to a compositor that maps the texture of a related video object onto the mesh. The coded order and the displayed order of the mesh are identical. Mesh object planes can be used to deform a video object plane or still texture object by piece-wise warping.

7.12.3 Face animation parameter data

The output of the decoding process is a sequence of facial animation parameters. They are input to a display process that uses the parameters to animate a face object.

8. Visual-Systems Composition Issues

8.1 Temporal Scalability Composition

Background composition is used in forming the background region for objects at the enhancement layer of temporal scalability when the value of both enhancement_type and background_composition is one. This process is useful when the enhancement VOP corresponds to the partial region of the VOP belonging to the reference layer. In this process, the background of a current enhancement VOP is composed using the previous and the next VOPs in display order belonging to the reference layer.

Figure 8-1 shows the background composition for the current frame at the enhancement layer. The dotted line represents the shape of the selected object at the previous VOP in the reference layer (called “forward shape”). As the object moves, its shape at the next VOP in the reference layer is represented by a broken line (called “backward shape”).

For the region outside these shapes, the pixel value from the nearest VOP at the reference layer is used for the composed background. For the region occupied only by the forward shape, the pixel value from the next VOP at the reference layer is used for the composed frame. This area is shown as lightly shaded in Figure 8-1. On the other hand, for the region occupied only by the backward shape, pixel values from the previous VOP in the reference layer are used. This is the area shaded dark in Figure 8-1. For the region where the areas enclosed by these shapes overlap, the pixel value is given by padding from the surrounding area. The pixel value which is outside of the overlapped area should be filled before the padding operation.

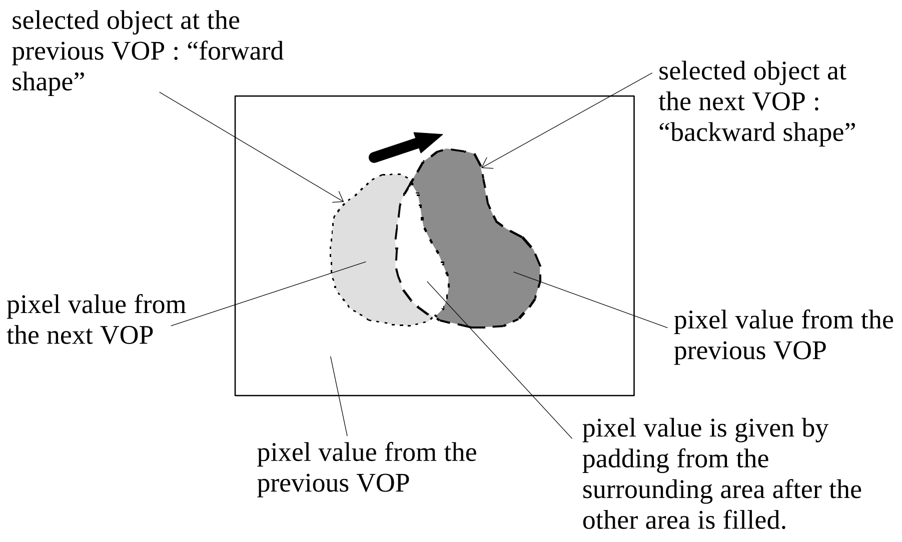

selected object at the previous VOP : “forward shape”
selected object at the next VOP : “backward shape”
pixel value from the next VOP
pixel value from the previous VOP
pixel value from the previous VOP
pixel value is given by padding from the surrounding area after the other area is filled.

Figure 8-1– Background composition

The following process is a mathematical description of the background composition method.

```
If s(x,y,ta)=0 and s(x,y,td)=0
        fc(x,y,t) = f(x,y,td)      (|t-ta|>|t-td|)
        fc(x,y,t) = f(x,y,ta)      (otherwise),
if s(x,y,ta)=1 and s(x,y,td)=0
        fc(x,y,t) = f(x,y,td)
if s(x,y,ta)=0 and s(x,y,td)=1
        fc(x,y,t) = f(x,y,ta)
if s(x,y,ta)=1 and s(x,y,td)=1
        The pixel value of fc(x,y,t) is given by repetitive padding from the surrounding area.
```

where

fc	composed background
f	decoded VOP at the reference layer
s	shape information (alpha plane) , 0: transparent, 1: opaque
(x,y)	the spatial coordinate
t	time of the current VOP
ta	time of the previous VOP
td	time of the next VOP

Two types of shape information, *s*(*x*, *y*, *ta*) and *s*(*x*, *y*, *td*), are necessary for the background composition. *s*(*x*, *y*, *ta*) is called a "forward shape" and *s*(*x*, *y*, *td*) is called a "backward shape". If f(*x*, *y*, *td*) is the last VOP in the bitstream of the reference layer, it should be made by copying f(*x*, *y*, *ta*). In this case, two shapes s(*x*, *y*, *ta*) and s(*x*, *y*, *td*) should be identical to the previous backward shape.

8.2 Sprite Composition

The static sprite technology enables to encode very efficiently video objects which content is expected not to vary in time along a video sequence. For example, it is particularly well suited to represent backgrounds of scenes (decor, landscapes) or logos.

A static sprite (sometimes referred as mosaic in the literature) is a frame containing spatial information for a single object, obtained by gathering information for this object throughout the sequence in which it appears. A static sprite can be a very large frame: it can correspond for instance to a wide angle view of a panorama.

The MPEG-4 syntax defines a dedicated coding mode to obtain VOPs from static sprites: the so-called "Sprite-VOPs". Sprite-VOPs are extracted from a static sprite using a warping operation consisting in a global spatial transformation driven by few motion parameters (0,2,4 or 8).

For composition with other VOPs, there are no special rules for Sprite-VOPs. However, it is classical to use Sprite-VOPs as background objects over which "classical" objects are superimposed.

9. Profiles and Levels

NOTE - In this Specification the word “profile” is used as defined below. It should not be confused with other definitions of “profile” and in particular it does not have the meaning that is defined by JTC1/SGFS.

Profiles and levels provide a means of defining subsets of the syntax and semantics of this Specification and thereby the decoder capabilities required to decode a particular bitstream. A profile is a defined sub-set of the entire bitstream syntax that is defined by this Specification. A level is a defined set of constraints imposed on parameters in the bitstream. Conformance tests will be carried out against defined profiles at defined levels.

The purpose of defining conformance points in the form of profiles and levels is to facilitate bitstream interchange among different applications. Implementers of this Specification are encouraged to produce decoders and bitstreams which correspond to those defined conformance regions. The discretely defined profiles and levels are the means of bitstream interchange between applications of this Specification.

In this clause the constrained parts of the defined profiles and levels are described. All syntactic elements and parameter values which are not explicitly constrained may take any of the possible values that are allowed by this Specification. In general, a decoder shall be deemed to be conformant to a given profile at a given level if it is able to properly decode all allowed values of all syntactic elements as specified by that profile at that level.

9.1 Visual Object Profiles

The following table lists the tools included in each of the Object Profiles. Bitstreams that represent a particular object corresponding to an Object Profile shall not use any of the tools for which the table does not have an ‘X’.

Visual Tools	**Visual Object Profiles**									
	Simple	Core	Main	Simple Scalable	12-Bit	Basic Anim. 2D Texture	Anim. 2D Mesh	Simple Face	Simple Scalable Texture	Core Scalable Texture
Intra Coding Mode (I-VOP)	X	X	X	X	X		X			
Inter Prediction Mode (P-VOP)	X	X	X	X	X		X			
AC/DC Prediction	X	X	X	X	X		X			
Slice Resynchronization	X	X	X	X	X		X			
Data Partitioning	X	X	X	X	X		X			
Reversible VLC	X	X	X	X	X		X			
4MV, Unrestricted MV	X	X	X	X	X		X			
Binary Shape Coding		X	X		X	X	X			
H.263/MPEG-2 Quantization Tables		X	X		X		X			
P-VOP based temporal scalability Rectangular Shape		X	X	X	X		X			
P-VOP based temporal scalability Arbitrary Shape		X	X		X		X			
Bi-directional Pred. Mode (B-VOP)		X	X	X			X			
OBMC			X							
Temporal Scalability Rectangular Shape				X						
Temporal Scalability Arbitrary Shape										
Spatial Scalability Rectangular Shape				X						
Static Sprites (includes low latency mode)			X							
Interlaced tools			X							
Grayscale Alpha Shape Coding			X							
4- to 12-bit pixel depth					X					
2D Dynamic Mesh with uniform topology						X	X			
2D Dynamic Mesh with Delaunay topology							X			
Facial Animation Parameters								X		
Scalable Wavelet Texture (rectangular, Spatial & SNR Scalable)						X	X			X
Scalable Wavelet Texture (spatial scaleable)							X		X	X
Scalable Wavelet Texture (all tools, including Shape Adaptive)						X	X			

Table 9-1: Object Profiles

9.2 Visual Combination Profiles

Decoders that conform to a combination Profile shall be able to decode all objects that comply to the Object Profiles for which the table lists an ‘X’.

	Object Profiles Combination Profiles	Simple	Core	Main	Simple Scale.	12-Bit	Basic Anim. 2D Texture	Anim. 2D Mesh	Simple Face	Simple Scalable Texture	Core Scalable Texture
1.	Simple	X									
2.	Simple B-VOP Scaleable	X			X						
3.	Core	X	X								
4.	Main	X	X	X						X	X
5.	12-Bit	X	X			X					
6.	Simple Scaleable Texture									X	
7.	Simple FA								X		
8.	Hybrid	X	X				X	X	X	X	X
9.	Basic Animated 2D Texture						X		X	X	X

Table 9-2 -Visual Combination Profiles

Note that the Combination Profiles can be grouped into three categories: Natural Visual (Combination Profile numbers 1-5), Synthetic Visual (Combination Profile numbers 6 and 7), and Synthetic/Natural Hybrid Visual (Combination Profile numbers 8 and 9).

9.3 Visual Combination Profiles@Levels

9.3.1 Natural Visual

The table that describes the natural visual combination profiles is given in Annex M.

9.3.2 Synthetic Visual

9.3.2.1 Simple Texture CP

Level 1:

- Quantization modes: restricted to mode 1
- Default Integer wavelet;
- number of objects = t.b.d.
- Maximum number of pixels: 414720 (note: although this corresponds to the amount pixels in an ITU-R Rec. BT.601 frame, i.e. 576 * 720 pixels, there are no restrictions on the aspect ratio)

- Maximum decoding time: 1 second

Level 2:

- t.b.d.

2.4.2.2 Simple Face Animation CP

Level 1:

- number of objects: 1
- The total FAP decode frame-rate in the bitstream shall not exceed 72 Hz
- The decoder shall be capable of a face model rendering update of at least 15 Hz;
- maximum bitrate 16 kbit/s;

Level 2:

- maximum number of objects: 4
- The FAP decode frame-rate in the bitstream shall not exceed 72 Hz (this means that the FAP decode framerate is to be shared among the objects);
- The decoder shall be capable of rendering the face models with the update rate of at least 60Hz, sharable between faces, with the constraint that the update rate for each individual face is not required to exceed 30Hz.
- Maximum bitrate 32 kbit/s;

2.4.3 Synthetic/Natural Hybrid Visual

The *Levels* of the Combination Profiles which support both Natural Visual Object Profiles and Synthetic Visual Object Profiles are specified by giving bounds for the natural objects and for the synthetic objects. Parameters like bitrate can be combined across natural and synthetic objects.

2.4.3.1 Basic Animated 2D Texture

Level 1 = Facial Animation CP @ Level 1 + restrictions on Texture objects: t.b.d.

Level 2= Facial Animation CP @ Level 2 + restrictions on Texture objects: t.b.d.

2.4.3.2 Hybrid CP

Level 1 = Core Visual Combination Profile @ Level 1 + Basic Animated 2D Texture @ Level 1 +

- Max number of Mesh objects tbd
- Number of nodes in the Meshes (4 times # of macroblocks in the visual session) = t.b.d. (proposal: 792);
- max. frame-rate of the mesh - 30 Hz;
- max. bitrate t.b.d.

Level 2 = Core Visual Combination Profile @ Level 2 + Basic Animated 2D Texture @ Level 2 +

- max number of Mesh objects tbd
- Number of nodes in the Meshes (4 times # of macroblocks in the visual session) = t.b.d. (proposal: 3168);
- max. frame-rate of the mesh: t.b.d.
- Max. bitrate: t.b.d.
- Restrictions on texture t.b.d.

10. Annex A
Coding Transforms

(This annex forms an integral part of the International Standard)

10.1 Discrete cosine transform for video texture

The NxN two dimensional DCT is defined as:

$$F(u,v) = \frac{2}{N} C(u)C(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) \cos\frac{(2x+1)up}{2N} \cos\frac{(2y+1)vp}{2N}$$

with u, v, x, y = 0, 1, 2, … N-1

where x, y are spatial coordinates in the sample domain

u, v are coordinates in the transform domain

$$C(u), C(v) = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } u,v = 0 \\ 1 & \text{otherwise} \end{cases}$$

The inverse DCT (IDCT) is defined as:

$$f(x,y) = \frac{2}{N} \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} C(u)C(v)F(u,v) \cos\frac{(2x+1)up}{2N} \cos\frac{(2y+1)vp}{2N}$$

The input to the forward transform and output from the inverse transform is represented with 9 bits. The coefficients are represented in 12 bits. The dynamic range of the DCT coefficients is [-2048:+2047].

The N by N inverse discrete transform shall conform to IEEE Standard Specification for the Implementations of 8 by 8 Inverse Discrete Cosine Transform, Std 1180-1990, December 6, 1990.

NOTES -

1 Clause 2.3 Std 1180-1990 “Considerations of Specifying IDCT Mismatch Errors” requires the specification of periodic intra-picture coding in order to control the accumulation of mismatch errors. Every macroblock is required to be refreshed before it is coded 132 times as predictive macroblocks. Macroblocks in B-pictures (and skipped macroblocks in P-pictures) are excluded from the counting because they do not lead to the accumulation of mismatch errors. This requirement is the same as indicated in 1180-1990 for visual telephony according to ITU-T Recommendation H.261.

2 Whilst the IEEE IDCT standard mentioned above is a necessary condition for the satisfactory implementation of the IDCT function it should be understood that this is not sufficient. In particular attention is drawn to the following sentence from 5.4 of this specification: “Where arithmetic precision is not specified, such as the calculation of the IDCT, the precision shall be sufficient so that significant errors do not occur in the final integer values.”

10.2 Discrete wavelet transform for still texture

10.2.1 Adding the mean

Before applying the inverse wavelet transform, the mean of each color component (“mean_y”, “mean_u”, and “mean_v”) is added to the all wavelet coefficients of dc band.

10.2.2 wavelet filter

A 2-D separable inverse wavelet transfrom is used to synthesize the still texture. The default wavelet composition is performed using Daubechies (9,3) tap biorthogonal filter bank. The inverse DWT is performed either in floating or integer operations depending on the field “wavelet_filter_type”, defined in the syntax.

The floating filter coefficients are:

Lowpass	**g[ ] =**	
[0.35355339059327	0.70710678118655	0.35355339059327]

Highpas	**h[ ] =**	
[0.03314563036812	0.06629126073624	-0.17677669529665
-0.41984465132952	0.99436891104360	-0.41984465132952
-0.17677669529665	0.06629126073624	0.03314563036812]

The integer filter coefficients are:

Lowpass	**g[ ] =**	
32	64	32

Highpass	**h[ ] =**	
3	6	-16
-38	90	-38
-16	6	3

The synthesis filtering operation is defined as follows:

$$y[n] = \sum_{i=-1}^{1} L[n+i]*g[i+1] \quad + \quad \sum_{i=-4}^{4} H[n+i]*h[i+4]$$

where

- n = 0, 1, ... N-1, and N is the number of output points;
- L[2*i] = xl[i] and L[2*i+1] = 0 for i=0,1,...,N/2-1, and {xl[i]} are the N/2 input wavelet coefficients in the low-pass band;
- H[2*i+1] = xh[i] and H[2*i] = 0 for i=0,1,...,N/2-1, and {xh[i]} are the N/2 input wavelet coefficients in the high-pass band.

Note:

- the index range for h[] is from 0 to 8;
- the index range for g[] is from 0 to 2;

- the index range for L[] is from -1 to N;
- the index range for H[] is from -4 to N+3; and
- the values of L[] and H[] for indexes less than 0 or greater than N-1 are obtained by symmetric extension described in the following section.

In the case of integer wavelet, the outputs at each composition level are scaled down with dividing by 8096 with rounding to the nearest integer.

10.2.3 Symmetric extension

A symmetric extension of the input wavelet coefficients is performed before up-sampling and applying the wavelet composition at each level. Two types of symmetric extensions are needed, both mirror the boundary pixels. Type A replicates the edge pixel and Type B does not replicate the edge pixel. This is illustrated in Figure 10-1 and Figure 10-2, where the edge pixel is indicated by z. The types of extension for the input data to the wavelet filters are shown in Table 10-1.

Type A **…v w x y z** | z y x w v…

Type B **…...v w x y** | z y x w v…

Figure 10-1 Symmetrical extensions at leading boundary

Type A …v w x y z | **z y x w v…**

Type B …v w x y z | **y x w v…**

Figure 10-2 Symmetrical extensions at the trailing boundary

Table 10-1Extension method for the input data to the synthesis filters

	boundary	Extension
lowpass input xl[] to 3-tap filter g[]	leading trailing	TypeB TypeA
highpass input xh[] to 9-tap filter h[]	leading trailing	TypeA TypeB

10.2.4 Decomposition level

The number of decomposition levels of the luminance component is defined in the input bitstream. The number of decompostion level for the chrominance components is one level less than the luminance components. If texture_object_layer width or texture_object_layer height cannot be divisible by (2 ^ decomposition_levels), then shape adaptive wavelet is applied.

10.2.5 Shape adaptive wavelet filtering and symmetric extension

10.2.5.1 Shape adaptive wavelet

The 2-D inverse shape adaptive wavelet transform uses the same wavelet filter as specified in Table 10-1. According to the shape information, segments of consecutive output points are reconstructed and put into the correct locations. The filtering operation of shape adaptive wavelet is a generalization of that for the regular wavelet. The generalization allows the number of output points to be an odd number as well as an even number. Relative to the bounding box, the starting point of the output is also allowed to be an odd number as well as an even number according to the shape information. Within the generalized wavelet filtering, the regular wavelet filtering is a special case where the number of output points is an even number and the starting point is an even number (0) too. Another special case is for reconstruction of rectangular textures with an arbitrary size where the number of output points may be even or odd and the starting point is always even (0).

The same synthesis filtering is applied for shape-adaptive wavelet composition, i.e:

$$y[n] = \sum_{i=-1}^{1} L[n+i]*g[i+1] \quad + \sum_{i=-4}^{4} H[n+i]*h[i+4]$$

where

- n = 0, 1, ... N-1, and N is the number of output points;
- L[2*i+s] = xl[i] and L[2*i+1-s] = 0 for i=0,1,...,(N+1-s)/2-1, and {xl[i]} are the (N+1-s)/2 input wavelet coefficients in the low-pass band;
- H[2*i+1-s] = xh[i] and H[2*i+s] = 0 for i=0,1,...,(N+s)/2-1, and {xh[i]} are the (N+s)/2 input wavelet coefficients in the high-pass band.

The only difference from the regular synthesis filtering is to introduce a binary parameter s in up-sampling, where s = 0 if the starting point of the output is an even number and s = 1 if the starting point of the output is an odd number.

The symmetric extension for the generalized synthesis filtering is specified in Table 10-2 if N is an even number and in Table 10-3 if N is an odd number.

Table 10-2 Extension method for the data to the synthesis wavelet filters if N is even

	Boundary	extension (s=0)	extension(s=1)
lowpass input xl[]	Leading	TypeB	TypeA
to 3-tap filter g[]	Trailing	TypeA	TypeB
highpass input xh[]	Leading	TypeA	TypeB
to 9-tap filter h[]	Trailing	TypeB	TypeA

Table 10-3 Extension method for the data to the synthesis wavelet filters if N is odd

	Boundary	extension(s=0)	extension(s=1)
lowpass input xl[]	Leading	TypeB	TypeA
to 3-tap filter g[]	Trailing	TypeB	TypeA
highpass input xh[]	Leading	TypeA	TypeB
to 9-tap filter h[]	Trailing	TypeA	TypeB

11. Annex B
Variable length codes and Arithmetic Decoding

(This annex forms an integral part of the International Standard)

11.1 Variable length codes

11.1.1 Macroblock type

Table 11-1Macroblock types and included data elements for I- and P-VOPs in combined motion-shape-texture coding

VOP type	mb type	Name	not_coded	mcbc	cbpy	dquant	mvd	mvd_{2-4}
P	not coded	-	1					
P	0	inter	1	1	1		1	
P	1	inter+q	1	1	1	1	1	
P	2	inter4v	1	1	1		1	1
P	3	intra	1	1	1			
P	4	intra+q	1	1	1	1		
P	stuffing	-	1	1				
I	3	intra		1	1			
I	4	intra+q		1	1	1		
I	stuffing	-		1				
S (update)	not_coded	-	1					
S (update)	0	inter	1	1	1			
S (update)	1	inter+q	1	1	1	1		
S (update)	3	intra	1	1	1			
S (update)	4	intra+q	1	1	1	1		
S (update)	stuffing	-	1	1				
S (piece)	3	intra		1	1			
S (piece)	4	intra+q		1	1	1		
S (piece)	stuffing	-		1				

Note: “1” means that the item is present in the macroblock
S (piece) indicates S-VOPs with low_latency_sprite_enable == 1 and sprite_transmit_mode == “piece”
S (update) indicates S-VOPs with low_latency_sprite_enable == 1 and sprite_transmit_mode == “update”

Table 11-2 Macroblock types and included data elements for a P-VOP (scalability && ref_select_code == ‘11’)

VOP Type	mb_type	Name	COD	MCBPC	CBPY	DQUANT	MVD	MVD$_{2-4}$
P	not coded	-	1					
P	1	INTER	1	1	1			
P	2	INTER +Q	1	1	1	1		
P	3	INTRA	1	1	1			
P	4	INTRA +Q	1	1	1	1		
P	stuffing	-	1	1				

Note: "1" means that the item is present in the macroblock

Table 11-3 --- VLC table for MODB in combined motion-shape-texture coding

Code	cbpb	mb_type
0		
10		1
11	1	1

Table 11-4 --- MBTYPES and included data elements in coded macroblocks in B-VOPs (ref_select_code != '00'||scalability=='0') for combined motion-shape-texture coding

Code	dquant	mvd$_f$	mvd$_b$	mvdb	MBTYPE
1				1	direct
01	1	1	1		interpolate mc+q
001	1		1		backward mc+q
0001	1	1			forward mc+q

Table 11-5 --- MBTYPES and included data elements in coded macroblocks in B-VOPs (ref_select_code == '00'&&scalability!='0') for combined motion-shape-texture coding

Code	dquant	mvd$_f$	mvd$_b$	MBTYPE
01	1	1		interpolate mc+q
001	1			backward mc+q
1	1	1		forward mc+q

11.1.2 Macroblock pattern

Table 11-6-- VLC table for MCBPC for I-VOPs in combined-motion-shape-texture coding and sprite-VOPs with low_latence_sprite_enable==1 and sprite_transmit_mode=="piece"

Code	mbtype	cbpc (56)
1	3	00
001	3	01
010	3	10
011	3	11
0001	4	00
0000 01	4	01
0000 10	4	10
0000 11	4	11
0000 0000 1	Stuffing	--

Table 11-7 --- VLC table for MCBPC for P-VOPs in combined-motion-shape-texture and sprite-VOPs with low_latence_sprite_enable==1 and sprite_transmit_mode==”update”

Code	MB type	CBPC (56)
1	0	00
0011	0	01
0010	0	10
0001 01	0	11
011	1	00
0000 111	1	01
0000 110	1	10
0000 0010 1	1	11
010	2	00
0000 101	2	01
0000 100	2	10
0000 0101	2	11
0001 1	3	00
0000 0100	3	01
0000 0011	3	10
0000 011	3	11
0001 00	4	00
0000 0010 0	4	01
0000 0001 1	4	10
0000 0001 0	4	11
0000 0000 1	Stuffing	--

Table 11-8 --- VLC table for CBPY in the case of four non-transparent macroblocks

Code	CBPY(intra-MB) (12 34)	CBPY(inter-MB), (12 34)
0011	00 00 0 00	11 11
0010 1	00 01	11 10
0010 0	00 10	11 01
1001	00 11	11 00
0001 1	01 00	10 11
0111	01 01	10 10
0000 10	01 10	10 01
1011	01 11	10 00
0001 0	10 00	01 11
0000 11	10 01	01 10
0101	10 10	01 01
1010	10 11	01 00
0100	11 00	00 11
1000	11 01	00 10
0110	11 10	00 01
11	11 11	00 00

Table 11-9 VLC table for CBPY in the case of two non transparent blocks.

Code	CBPY (I)(intra-MB)	CBPY (inter-MB)
0001	00	11
001	01	10
01	10	01
1	11	00

Table 11-10 VLC table for CBPY in the case of three non transparent blocks

Code	CBPY (I)(intra-MB)	CBPY (inter-MB)
011	000	111
000001	001	110
00001	010	101
010	011	100
00010	100	011
00011	101	010
001	110	001
1	111	000

In the case of a single transparent block then,

Code	CBPY (I)(intra-MB)	CBPY (inter-MB)
01	0	1
1	1	0

11.1.3 Motion vector

Table 11-11 --- VLC table for MVD

Codes	Vector differences

0000 0000 0010 1	-16
0000 0000 0011 1	-15.5
0000 0000 0101	-15
0000 0000 0111	-14.5
0000 0000 1001	-14
0000 0000 1011	-13.5
0000 0000 1101	-13
0000 0000 1111	-12.5
0000 0001 001	-12
0000 0001 011	-11.5
0000 0001 101	-11
0000 0001 111	-10.5
0000 0010 001	-10
0000 0010 011	-9.5
0000 0010 101	-9
0000 0010 111	-8.5
0000 0011 001	-8
0000 0011 011	-7.5
0000 0011 101	-7
0000 0011 111	-6.5
0000 0100 001	-6
0000 0100 011	-5.5
0000 0100 11	-5
0000 0101 01	-4.5
0000 0101 11	-4
0000 0111	-3.5
0000 1001	-3
0000 1011	-2.5
0000 111	-2
0001 1	-1.5
0011	-1
011	-0.5
1	0
010	0.5
0010	1
0001 0	1.5
0000 110	2
0000 1010	2.5
0000 1000	3
0000 0110	3.5
0000 0101 10	4
0000 0101 00	4.5
0000 0100 10	5
0000 0100 010	5.5
0000 0100 000	6
0000 0011 110	6.5

0000 0011 100	7
0000 0011 010	7.5
0000 0011 000	8
0000 0010 110	8.5
0000 0010 100	9
0000 0010 010	9.5
0000 0010 000	10
0000 0001 110	10.5
0000 0001 100	11
0000 0001 010	11.5
0000 0001 000	12
0000 0000 1110	12.5
0000 0000 1100	13
0000 0000 1010	13.5
0000 0000 1000	14
0000 0000 0110	14.5
0000 0000 0100	15
0000 0000 0011 0	15.5
0000 0000 0010 0	16

11.1.4 DCT coefficients

Table 11-12 --- Variable length codes for dct_dc_size_luminance

Variable length code	**dct_dc_size_luminance**
011	0
11	1
10	2
010	3
001	4
0001	5
0000 1	6
0000 01	7
0000 001	8
0000 0001	9
0000 0000 1	10
0000 0000 01	11
0000 0000 001	12

Table 11-13 --- Variable length codes for dct_dc_size_chrominance

Variable length code	**dct_dc_size_chrominance**
11	0
10	1
01	2
001	3
0001	4
0000 1	5
0000 01	6
0000 001	7
0000 0001	8
0000 0000 1	9
0000 0000 01	10
0000 0000 001	11
0000 0000 0001	12

Table 11-14 --- Differential DC additional codes

ADDITIONAL CODE	**DIFFERENTIAL DC**	**SIZE**
000000000000 to 011111111111	-2048 to -4095	12
00000000000 to 01111111111	-1024 to -2047	11
0000000000 to 0111111111	-512 to -1023	10
000000000 to 011111111	-256 to -511	9
00000000 to 01111111	-255 to -128	8
0000000 to 0111111	-127 to -64	7
000000 to 011111	-63 to -32	6
00000 to 01111	-31 to -16	5
0000 to 0111	-15 to -8	4
000 to 011	-7 to -4	3
00 to 01	-3 to -2	2
0	-1	1
	0	0
1	1	1
10 to 11	2 to 3	2
100 to 111	4 to 7	3
1000 to 1111	8 to 15	4
10000 to 11111	16 to 31	5
100000 to 111111	32 to 63	6
1000000 to 1111111	64 to 127	7

10000000 to 11111111	128 to 255	8
100000000 to 111111111 *	256 to 511	9
1000000000 to 1111111111 *	512 to 1023	10
1000000000 to 1111111111 *	1024 to 2047	11
1000000000 to 1111111111 *	2048 to 4095	12

In cases where dct_dc_size is greater than 8, marked '*' in , a marker bit is inserted after the dct_dc_additional_code to prevent start code emulations.

Table 11-15 --- VLC Table for Intra Luminance and Chrominance TCOEF

VLC CODE	LAST	RUN	LEVEL	VLC CODE	LAST	RUN	LEVEL
10s	0	0	1	0111 s	1	0	1
1111 s	0	0	3	0000 1100 1s	0	11	1
0101 01s	0	0	6	0000 0000 101s	1	0	6
0010 111s	0	0	9	0011 11s	1	1	1
0001 1111 s	0	0	10	0000 0000 100s	1	0	7
0001 0010 1s	0	0	13	0011 10s	1	2	1
0001 0010 0s	0	0	14	0011 01s	0	5	1
0000 1000 01s	0	0	17	0011 00s	1	0	2
0000 1000 00s	0	0	18	0010 011s	1	5	1
0000 0000 111s	0	0	21	0010 010s	0	6	1
0000 0000 110s	0	0	22	0010 001s	1	3	1
0000 0100 000s	0	0	23	0010 000s	1	4	1
110s	0	0	2	0001 1010 s	1	9	1
0101 00s	0	1	2	0001 1001 s	0	8	1
0001 1110 s	0	0	11	0001 1000 s	0	9	1
0000 0011 11s	0	0	19	0001 0111 s	0	10	1
0000 0100 001s	0	0	24	0001 0110 s	1	0	3
0000 0101 0000s	0	0	25	0001 0101 s	1	6	1
1110 s	0	1	1	0001 0100 s	1	7	1
0001 1101 s	0	0	12	0001 0011 s	1	8	1
0000 0011 10s	0	0	20	0000 1100 0s	0	12	1
0000 0101 0001s	0	0	26	0000 1011 1s	1	0	4
0110 1s	0	0	4	0000 1011 0s	1	1	2
0001 0001 1s	0	0	15	0000 1010 1s	1	10	1
0000 0011 01s	0	1	7	0000 1010 0s	1	11	1
0110 0s	0	0	5	0000 1001 1s	1	12	1
0001 0001 0s	0	4	2	0000 1001 0s	1	13	1
0000 0101 0010s	0	0	27	0000 1000 1s	1	14	1
0101 1s	0	2	1	0000 0001 11s	0	13	1

0000 0011 00s	0	2	4
0000 0101 0011s	0	1	9
0100 11s	0	0	7
0000 0010 11s	0	3	4

0000 0001 10s	1	0	5
0000 0001 01s	1	1	3
0000 0001 00s	1	2	2
0000 0100 100s	1	3	2

VLC CODE	LAST	RUN	LEVEL
0000 0101 0100s	0	6	3
0100 10s	0	0	8
0000 0010 10s	0	4	3
0100 01s	0	3	1
0000 0010 01s	0	8	2
0100 00s	0	4	1
0000 0010 00s	0	5	3
0010 110s	0	1	3
0000 0101 0101s	0	1	10
0010 101s	0	2	2
0010 100s	0	7	1
0001 1100 s	0	1	4
0001 1011 s	0	3	2
0001 0000 1s	0	0	16
0001 0000 0s	0	1	5
0000 1111 1s	0	1	6
0000 1111 0s	0	2	3
0000 1110 1s	0	3	3
0000 1110 0s	0	5	2
0000 1101 1s	0	6	2
0000 1101 0s	0	7	2
0000 0100 010s	0	1	8
0000 0100 011s	0	9	2
0000 0101 0110s	0	2	5
0000 0101 0111s	0	7	3

VLC CODE	LAST	RUN	LEVEL
0000 0100 101s	1	4	2
0000 0100 110s	1	15	1
0000 0100 111s	1	16	1
0000 0101 1000s	0	14	1
0000 0101 1001s	1	0	8
0000 0101 1010s	1	5	2
0000 0101 1011s	1	6	2
0000 0101 1100s	1	17	1
0000 0101 1101s	1	18	1
0000 0101 1110s	1	19	1
0000 0101 1111s	1	20	1
0000 011	escape		

Table 11-16--- VLC table for Inter Lumimance and Chrominance TCOEF

VLC CODE	LAST	RUN	LEVEL
10s	0	0	1
1111 s	0	0	2
0101 01s	0	0	3
0010 111s	0	0	4
0001 1111 s	0	0	5
0001 0010 1s	0	0	6
0001 0010 0s	0	0	7
0000 1000 01s	0	0	8
0000 1000 00s	0	0	9

VLC CODE	LAST	RUN	LEVEL
0111 s	1	0	1
0000 1100 1s	1	0	2
0000 0000 101s	1	0	3
0011 11s	1	1	1
0000 0000 100s	1	1	2
0011 10s	1	2	1
0011 01s	1	3	1
0011 00s	1	4	1
0010 011s	1	5	1

0000 0000 111s	0	0	10
0000 0000 110s	0	0	11
0000 0100 000s	0	0	12
110s	0	1	1
0101 00s	0	1	2
0001 1110 s	0	1	3
0000 0011 11s	0	1	4
0000 0100 001s	0	1	5
0000 0101 0000s	0	1	6
1110 s	0	2	1
0001 1101 s	0	2	2
0000 0011 10s	0	2	3
0000 0101 0001s	0	2	4
0110 1s	0	3	1
0001 0001 1s	0	3	2
0000 0011 01s	0	3	3
0110 0s	0	4	1
0001 0001 0s	0	4	2
0000 0101 0010s	0	4	3
0101 1s	0	5	1
0000 0011 00s	0	5	2
0000 0101 0011s	0	5	3
0100 11s	0	6	1
0000 0010 11s	0	6	2
0000 0101 0100s	0	6	3
0100 10s	0	7	1
0000 0010 10s	0	7	2
0100 01s	0	8	1
0000 0010 01s	0	8	2
0100 00s	0	9	1
0000 0010 00s	0	9	2
0010 110s	0	10	1
0000 0101 0101s	0	10	2
0010 101s	0	11	1
0010 100s	0	12	1
0001 1100 s	0	13	1
0001 1011 s	0	14	1
0001 0000 1s	0	15	1
0001 0000 0s	0	16	1
0000 1111 1s	0	17	1
0000 1111 0s	0	18	1
0000 1110 1s	0	19	1
0000 1110 0s	0	20	1
0000 1101 1s	0	21	1
0000 1101 0s	0	22	1
0000 0100 010s	0	23	1
0010 010s	1	6	1
0010 001s	1	7	1
0010 000s	1	8	1
0001 1010 s	1	9	1
0001 1001 s	1	10	1
0001 1000 s	1	11	1
0001 0111 s	1	12	1
0001 0110 s	1	13	1
0001 0101 s	1	14	1
0001 0100 s	1	15	1
0001 0011 s	1	16	1
0000 1100 0s	1	17	1
0000 1011 1s	1	18	1
0000 1011 0s	1	19	1
0000 1010 1s	1	20	1
0000 1010 0s	1	21	1
0000 1001 1s	1	22	1
0000 1001 0s	1	23	1
0000 1000 1s	1	24	1
0000 0001 11s	1	25	1
0000 0001 10s	1	26	1
0000 0001 01s	1	27	1
0000 0001 00s	1	28	1
0000 0100 100s	1	29	1
0000 0100 101s	1	30	1
0000 0100 110s	1	31	1
0000 0100 111s	1	32	1
0000 0101 1000s	1	33	1
0000 0101 1001s	1	34	1
0000 0101 1010s	1	35	1
0000 0101 1011s	1	36	1
0000 0101 1100s	1	37	1
0000 0101 1101s	1	38	1
0000 0101 1110s	1	39	1
0000 0101 1111s	1	40	1
0000 011	escape		

0000 0100 011s	0	24	1
0000 0101 0110s	0	25	1
0000 0101 0111s	0	26	1

Table 11-17 --- FLC table for RUNS and LEVELS

Code	Run
000 000	0
000 001	1
000 010	2
.	.
.	.
111 111	63

Code	Level
forbidden	-2048
1000 0000 0001	-2047
.	.
1111 1111 1110	-2
1111 1111 1111	-1
forbidden	0
0000 0000 0001	1
0000 0000 0010	2
.	.
0111 1111 1111	2047

Table 11-18 ESCL(a), LMAX values of intra macroblocks

LAST	RUN	LMAX	LAST	RUN	LMAX
0	0	27	1	0	8
0	1	10	1	1	3
0	2	5	1	2-6	2
0	3	4	1	7-20	1
0	4-7	3	1	others	N/A
0	8-9	2			
0	10-14	1			
0	others	N/A			

Table 11-19 ESCL(b), LMAX values of inter macroblocks

LAST	RUN	LMAX	LAST	RUN	LMAX
0	0	12	1	0	3
0	1	6	1	1	2
0	2	4	1	2-40	1
0	3-6	3	1	others	N/A
0	7-10	2			
0	11-26	1			
0	others	N/A			

Table 11-20 ESCR(a), RMAX values of intra macroblocks

LAST	LEVEL	RMAX
0	1	14
0	2	9
0	3	7
0	4	3
0	5	2
0	6-10	1
0	11-27	0
0	others	N/A

LAST	LEVEL	RMAX
1	1	20
1	2	6
1	3	1
1	4-8	0
1	others	N/A

Table 11-21 ESCR(b), RMAX values of inter macroblocks

LAST	LEVEL	RMAX
0	1	26
0	2	10
0	3	6
0	4	2
0	5-6	1
0	7-12	0
0	others	N/A

LAST	LEVEL	RMAX
1	1	40
1	2	1
1	3	0
1	others	N/A

Table 11-22 RVLC table for TCOEF

ESCAPE code is added at the beginning and the end of these fixed-length codes for realizing two-way decode as shown below.

ESCAPE	LAST	RUN	LEVEL	ESCAPE
00001	x	xxxxxx	xxxxxxx	0000s

Note: There are two types for ESCAPE added at the end of these fixed-length codes, and codewords are “0000s”. Also, S=0 : LEVEL is positive and S=1 : LEVEL is negative.

	intra			inter				
INDEX	LAST	RUN	LEVEL	LAST	RUN	LEVEL	BITS	VLC_CODE
0	0	0	1	0	0	1	4	110s
1	0	0	2	0	1	1	4	111s
2	0	1	1	0	0	2	5	0001s
3	0	0	3	0	2	1	5	1010s
4	1	0	1	1	0	1	5	1011s
5	0	2	1	0	0	3	6	00100s
6	0	3	1	0	3	1	6	00101s
7	0	1	2	0	4	1	6	01000s
8	0	0	4	0	5	1	6	01001s

9	1	1	1	1	1	1	6	10010s
10	1	2	1	1	2	1	6	10011s
11	0	4	1	0	1	2	7	001100s
12	0	5	1	0	6	1	7	001101s
13	0	0	5	0	7	1	7	010100s
14	0	0	6	0	8	1	7	010101s
15	1	3	1	1	3	1	7	011000s
16	1	4	1	1	4	1	7	011001s
17	1	5	1	1	5	1	7	100010s
18	1	6	1	1	6	1	7	100011s
19	0	6	1	0	0	4	8	0011100s
20	0	7	1	0	2	2	8	0011101s
21	0	2	2	0	9	1	8	0101100s
22	0	1	3	0	10	1	8	0101101s
23	0	0	7	0	11	1	8	0110100s
24	1	7	1	1	7	1	8	0110101s
25	1	8	1	1	8	1	8	0111000s
26	1	9	1	1	9	1	8	0111001s
27	1	10	1	1	10	1	8	1000010s
28	1	11	1	1	11	1	8	1000011s
29	0	8	1	0	0	5	9	00111100s
30	0	9	1	0	0	6	9	00111101s
31	0	3	2	0	1	3	9	01011100s
32	0	4	2	0	3	2	9	01011101s
33	0	1	4	0	4	2	9	01101100s
34	0	1	5	0	12	1	9	01101101s
35	0	0	8	0	13	1	9	01110100s
36	0	0	9	0	14	1	9	01110101s
37	1	0	2	1	0	2	9	01111000s
38	1	12	1	1	12	1	9	01111001s
39	1	13	1	1	13	1	9	10000010s
40	1	14	1	1	14	1	9	10000011s
41	0	10	1	0	0	7	10	001111100s
42	0	5	2	0	1	4	10	001111101s
43	0	2	3	0	2	3	10	010111100s
44	0	3	3	0	5	2	10	010111101s
45	0	1	6	0	15	1	10	011011100s
46	0	0	10	0	16	1	10	011011101s
47	0	0	11	0	17	1	10	011101100s
48	1	1	2	1	1	2	10	011101101s
49	1	15	1	1	15	1	10	011110100s
50	1	16	1	1	16	1	10	011110101s
51	1	17	1	1	17	1	10	011111000s
52	1	18	1	1	18	1	10	011111001s
53	1	19	1	1	19	1	10	100000010s
54	1	20	1	1	20	1	10	100000011s
55	0	11	1	0	0	8	11	0011111100s
56	0	12	1	0	0	9	11	0011111101s
57	0	6	2	0	1	5	11	0101111100s
58	0	7	2	0	3	3	11	0101111101s
59	0	8	2	0	6	2	11	0110111100s
60	0	4	3	0	7	2	11	0110111101s

61	0	2	4	0	8	2	11	0111011100s
62	0	1	7	0	9	2	11	0111011101s
63	0	0	12	0	18	1	11	0111101100s
64	0	0	13	0	19	1	11	0111101101s
65	0	0	14	0	20	1	11	0111110100s
66	1	21	1	1	21	1	11	0111110101s
67	1	22	1	1	22	1	11	0111111000s
68	1	23	1	1	23	1	11	0111111001s
69	1	24	1	1	24	1	11	1000000010s
70	1	25	1	1	25	1	11	1000000011s
71	0	13	1	0	0	10	12	00111111100s
72	0	9	2	0	0	11	12	00111111101s
73	0	5	3	0	1	6	12	01011111100s
74	0	6	3	0	2	4	12	01011111101s
75	0	7	3	0	4	3	12	01101111100s
76	0	3	4	0	5	3	12	01101111101s
77	0	2	5	0	10	2	12	01110111100s
78	0	2	6	0	21	1	12	01110111101s
79	0	1	8	0	22	1	12	01111011100s
80	0	1	9	0	23	1	12	01111011101s
81	0	0	15	0	24	1	12	01111101100s
82	0	0	16	0	25	1	12	01111101101s
83	0	0	17	0	26	1	12	01111110100s
84	1	0	3	1	0	3	12	01111110101s
85	1	2	2	1	2	2	12	01111111000s
86	1	26	1	1	26	1	12	01111111001s
87	1	27	1	1	27	1	12	10000000010s
88	1	28	1	1	28	1	12	10000000011s
89	0	10	2	0	0	12	13	001111111100s
90	0	4	4	0	1	7	13	001111111101s
91	0	5	4	0	2	5	13	010111111100s
92	0	6	4	0	3	4	13	010111111101s
93	0	3	5	0	6	3	13	011011111100s
94	0	4	5	0	7	3	13	011011111101s
95	0	1	10	0	11	2	13	011101111100s
96	0	0	18	0	27	1	13	011101111101s
97	0	0	19	0	28	1	13	011110111100s
98	0	0	22	0	29	1	13	011110111101s
99	1	1	3	1	1	3	13	011111011100s
100	1	3	2	1	3	2	13	011111011101s
101	1	4	2	1	4	2	13	011111101100s
102	1	29	1	1	29	1	13	011111101101s
103	1	30	1	1	30	1	13	011111110100s
104	1	31	1	1	31	1	13	011111110101s
105	1	32	1	1	32	1	13	011111111000s
106	1	33	1	1	33	1	13	011111111001s
107	1	34	1	1	34	1	13	100000000010s
108	1	35	1	1	35	1	13	100000000011s
109	0	14	1	0	0	13	14	0011111111100s
110	0	15	1	0	0	14	14	0011111111101s
111	0	11	2	0	0	15	14	0101111111100s
112	0	8	3	0	0	16	14	0101111111101s

113	0	9	3	0	1	8	14	0110111111100s
114	0	7	4	0	3	5	14	0110111111101s
115	0	3	6	0	4	4	14	0111011111100s
116	0	2	7	0	5	4	14	0111011111101s
117	0	2	8	0	8	3	14	0111101111100s
118	0	2	9	0	12	2	14	0111101111101s
119	0	1	11	0	30	1	14	0111110111100s
120	0	0	20	0	31	1	14	0111110111101s
121	0	0	21	0	32	1	14	0111111011100s
122	0	0	23	0	33	1	14	0111111011101s
123	1	0	4	1	0	4	14	0111111101100s
124	1	5	2	1	5	2	14	0111111101101s
125	1	6	2	1	6	2	14	0111111110100s
126	1	7	2	1	7	2	14	0111111110101s
127	1	8	2	1	8	2	14	0111111111000s
128	1	9	2	1	9	2	14	0111111111001s
129	1	36	1	1	36	1	14	1000000000010s
130	1	37	1	1	37	1	14	1000000000011s
131	0	16	1	0	0	17	15	00111111111100s
132	0	17	1	0	0	18	15	00111111111101s
133	0	18	1	0	1	9	15	01011111111100s
134	0	8	4	0	1	10	15	01011111111101s
135	0	5	5	0	2	6	15	01101111111100s
136	0	4	6	0	2	7	15	01101111111101s
137	0	5	6	0	3	6	15	01110111111100s
138	0	3	7	0	6	4	15	01110111111101s
139	0	3	8	0	9	3	15	01111011111100s
140	0	2	10	0	13	2	15	01111011111101s
141	0	2	11	0	14	2	15	01111101111100s
142	0	1	12	0	15	2	15	01111101111101s
143	0	1	13	0	16	2	15	01111110111100s
144	0	0	24	0	34	1	15	01111110111101s
145	0	0	25	0	35	1	15	01111111011100s
146	0	0	26	0	36	1	15	01111111011101s
147	1	0	5	1	0	5	15	01111111101100s
148	1	1	4	1	1	4	15	01111111101101s
149	1	10	2	1	10	2	15	01111111110100s
150	1	11	2	1	11	2	15	01111111110101s
151	1	12	2	1	12	2	15	01111111111000s
152	1	38	1	1	38	1	15	01111111111001s
153	1	39	1	1	39	1	15	10000000000010s
154	1	40	1	1	40	1	15	10000000000011s
155	0	0	27	0	0	19	16	001111111111100s
156	0	3	9	0	3	7	16	001111111111101s
157	0	6	5	0	4	5	16	010111111111100s
158	0	7	5	0	7	4	16	010111111111101s
159	0	9	4	0	17	2	16	011011111111100s
160	0	12	2	0	37	1	16	011011111111101s
161	0	19	1	0	38	1	16	011101111111100s
162	1	1	5	1	1	5	16	011101111111101s
163	1	2	3	1	2	3	16	011110111111100s
164	1	13	2	1	13	2	16	011110111111101s

165	1	41	1	1	41	1	16	011111011111100s
166	1	42	1	1	42	1	16	011111011111101s
167	1	43	1	1	43	1	16	011111101111100s
168	1	44	1	1	44	1	16	011111101111101s
169	ESCAPE						5	0000s

Table 11-23 FLC table for RUN

RUN	CODE
0	000000
1	000001
2	000010
:	:
63	111111

Table 11-24 FLC table for LEVEL

LEVEL	CODE
0	FORBIDDEN
1	0000001
2	0000010
:	:
127	1111111

11.1.5 Shape Coding

Table 11-25 Meaning of shape mode

Index	**Shape mode**
0	= "MVDs==0 && No Update"
1	= "MVDs!=0 && No Update"
2	transparent
3	opaque
4	"intraCAE"
5	"interCAE && MVDs==0"
6	"interCAE && MVDs!=0"

Table 11-26 bab_type for I-VOP

Index	(2)	(3)	(4)	Index	(2)	(3)	(4)
0	1	001	01	41	001	01	1
1	001	01	1	42	1	01	001
2	01	001	1	43	001	1	01

3	1	001	01	44	001	01	1
4	1	01	001	45	1	01	001
5	1	01	001	46	001	01	1
6	1	001	01	47	01	001	1
7	1	01	001	48	1	01	001
8	01	001	1	49	001	01	1
9	001	01	1	50	01	001	1
10	1	01	001	51	1	001	01
11	1	01	001	52	001	1	01
12	001	01	1	53	01	001	1
13	1	01	001	54	1	001	01
14	01	1	001	55	01	001	1
15	001	01	1	56	01	001	1
16	1	01	001	57	1	01	001
17	1	01	001	58	1	01	001
18	01	001	1	59	1	01	001
19	1	01	001	60	1	01	001
20	001	01	1	61	1	01	001
21	01	001	1	62	01	001	1
22	1	01	001	63	1	01	001
23	001	01	1	64	001	01	1
24	01	001	1	65	001	01	1
25	001	01	1	66	01	001	1
26	001	01	1	67	001	1	01
27	1	01	001	68	001	1	01
28	1	01	001	69	01	001	1
29	1	01	001	70	001	1	01
30	1	01	001	71	001	01	1
31	1	01	001	72	1	001	01
32	1	01	001	73	001	01	1
33	1	01	001	74	01	001	1
34	1	01	001	75	01	001	1
35	001	01	1	76	001	1	01
36	1	01	001	77	001	01	1
37	001	01	1	78	1	001	01
38	001	01	1	79	001	1	01

39	1	01	001	80	001	01	1
40	001	1	01				

Table 11-27 bab_type for P-VOP and B-VOP

		bab_type in current VOP (n)						
		0	1	2	3	4	5	6
bab_type in previous VOP(n-1)	0	1	01	00001	000001	0001	0010	0011
	1	01	1	00001	000001	001	0000001	0001
	2	0001	001	1	000001	01	0000001	00001
	3	1	0001	000001	001	01	0000001	00001
	4	011	001	0001	00001	1	000001	010
	5	01	0001	00001	000001	001	11	10
	6	001	0001	00001	000001	01	10	11

Table 11-28 VLC table for MVDs

MVDs	Codes
0	0
±1	10s
±2	110s
±3	1110s
±4	11110s
±5	111110s
±6	1111110s
±7	11111110s
±8	111111110s
±9	1111111110s
±10	11111111110s
±11	111111111110s
±12	1111111111110s
±13	11111111111110s
±14	111111111111110s
±15	1111111111111110s
±16	11111111111111110s

Table 11-29 VLC table for MVDs (Horizontal element is 0)

MVDs	Codes
±1	0s

±2	10s
±3	110s
±4	1110s
±5	11110s
±6	111110s
±7	1111110s
±8	11111110s
±9	111111110s
±10	1111111110s
±11	11111111110s
±12	111111111110s
±13	1111111111110s
±14	11111111111110s
±15	111111111111110s
±16	1111111111111110s

s: sign bit (if MVDs is positive s=”1”, otherwise s=”0”).

Table 11-30 VLC for conv_ratio

conv_ratio	Code
1	0
2	10
4	11

These tables contain the probabilities for a binary alpha pixel being equal to 0 for intra and inter shape coding using CAE. All probabilities are normalised to the range [1,65535].

As an example, given an INTRA context number C, the probability that the pixel is zero is given by intra_prob[C].

Table 11-31 Probabilities for arithmetic decoding of shape

```
USInt intra_prob[1024] = {
65267,16468,65003,17912,64573,8556,64252,5653,40174,3932,29789,277,45152,1140,32768,2043,
4499,80,6554,1144,21065,465,32768,799,5482,183,7282,264,5336,99,6554,563,
54784,30201,58254,9879,54613,3069,32768,58495,32768,32768,32768,2849,58982,54613,32768,12892,
31006,1332,49152,3287,60075,350,32768,712,39322,760,32768,354,52659,432,61854,150,
64999,28362,65323,42521,63572,32768,63677,18319,4910,32768,64238,434,53248,32768,61865,13590,
16384,32768,13107,333,32768,32768,32768,32768,32768,32768,1074,780,25058,5461,6697,233,
62949,30247,63702,24638,59578,32768,32768,42257,32768,32768,49152,546,62557,32768,54613,19258,
62405,32569,64600,865,60495,10923,32768,898,34193,24576,64111,341,47492,5231,55474,591,
65114,60075,64080,5334,65448,61882,64543,13209,54906,16384,35289,4933,48645,9614,55351,7318,
49807,54613,32768,32768,50972,32768,32768,32768,15159,1928,2048,171,3093,8,6096,74,
32768,60855,32768,32768,32768,32768,32768,32768,32768,32768,32768,32768,32768,55454,32768,57672,
32768,16384,32768,21845,32768,32768,32768,32768,32768,32768,32768,5041,28440,91,32768,45,
65124,10923,64874,5041,65429,57344,63435,48060,61440,32768,63488,24887,59688,3277,63918,14021,
32768,32768,32768,32768,32768,32768,32768,32768,690,32768,32768,1456,32768,32768,8192,728,
32768,32768,58982,17944,65237,54613,32768,2242,32768,32768,32768,42130,49152,57344,58254,16740,
32768,10923,54613,182,32768,32768,32768,7282,49152,32768,32768,5041,63295,1394,55188,77,
63672,6554,54613,49152,64558,32768,32768,5461,64142,32768,32768,32768,62415,32768,32768,16384,
1481,438,19661,840,33654,3121,64425,6554,4178,2048,32768,2260,5226,1680,32768,565,
60075,32768,32768,32768,32768,32768,32768,32768,32768,32768,32768,32768,32768,32768,32768,32768,
16384,261,32768,412,16384,636,32768,4369,23406,4328,32768,524,15604,560,32768,676,
49152,32768,49152,32768,32768,32768,64572,32768,32768,32768,54613,32768,32768,32768,32768,32768,
4681,32768,5617,851,32768,32768,59578,32768,32768,32768,3121,3121,49152,32768,6554,10923,
32768,32768,54613,14043,32768,32768,32768,3449,32768,32768,32768,32768,32768,32768,32768,32768,
57344,32768,57344,3449,32768,32768,32768,3855,58982,10923,32768,239,62259,32768,49152,85,
58778,23831,62888,20922,64311,8192,60075,575,59714,32768,57344,40960,62107,4096,61943,3921,
39862,15338,32768,1524,45123,5958,32768,58982,6669,930,1170,1043,7385,44,8813,5011,
59578,29789,54613,32768,32768,32768,32768,32768,32768,32768,32768,32768,58254,56174,32768,32768,
64080,25891,49152,22528,32768,2731,32768,10923,10923,3283,32768,1748,17827,77,32768,108,
62805,32768,62013,42612,32768,32768,61681,16384,58982,60075,62313,58982,65279,58982,62694,62174,
32768,32768,10923,950,32768,32768,32768,32768,5958,32768,38551,1092,11012,39322,13705,2072,
54613,32768,32768,11398,32768,32768,32768,145,32768,32768,32768,29789,60855,32768,61681,54792,
32768,32768,32768,17348,32768,32768,32768,8192,57344,16384,32768,3582,52581,580,24030,303,
62673,37266,65374,6197,62017,32768,49152,299,54613,32768,32768,32768,35234,119,32768,3855,
31949,32768,32768,49152,16384,32768,32768,32768,24576,32768,49152,32768,17476,32768,32768,57445,
51200,50864,54613,27949,60075,20480,32768,57344,32768,32768,32768,32768,32768,45875,32768,32768,
11498,3244,24576,482,16384,1150,32768,16384,7992,215,32768,1150,23593,927,32768,993,
65353,32768,65465,46741,41870,32768,64596,59578,62087,32768,12619,23406,11833,32768,47720,17476,
32768,32768,2621,6554,32768,32768,32768,32768,32768,32768,5041,32768,16384,32768,4096,2731,
63212,43526,65442,47124,65410,35747,60304,55858,60855,58982,60075,19859,35747,63015,64470,25432,
58689,1118,64717,1339,24576,32768,32768,1257,53297,1928,32768,33,52067,3511,62861,453,
64613,32768,32768,32768,64558,32768,32768,2731,49152,32768,32768,32768,61534,32768,32768,35747,
32768,32768,32768,32768,13107,32768,32768,32768,32768,32768,32768,32768,20480,32768,32768,32768,
32768,32768,32768,54613,40960,5041,32768,32768,32768,32768,32768,3277,64263,57592,32768,3121,
32768,32768,32768,32768,32768,10923,32768,32768,32768,8192,32768,32768,5461,6899,32768,1725,
63351,3855,63608,29127,62415,7282,64626,60855,32768,32768,60075,5958,44961,32768,61866,53718,
32768,32768,32768,32768,32768,32768,6554,32768,32768,32768,32768,32768,2521,978,32768,1489,
58254,32768,58982,61745,21845,32768,54613,58655,60075,32768,49152,16274,50412,64344,61643,43987,
32768,32768,32768,1638,32768,32768,32768,24966,54613,32768,32768,2427,46951,32768,17970,654,
65385,27307,60075,26472,64479,32768,32768,4681,61895,32768,32768,16384,58254,32768,32768,6554,
37630,3277,54613,6554,4965,5958,4681,32768,42765,16384,32768,21845,22827,16384,32768,6554,
65297,64769,60855,12743,63195,16384,32768,37942,32768,32768,32768,32768,60075,32768,62087,54613,
41764,2161,21845,1836,17284,5424,10923,1680,11019,555,32768,431,39819,907,32768,171,
65480,32768,64435,33803,2595,32768,57041,32768,61167,32768,32768,32768,32768,32768,32768,1796,
60855,32768,17246,978,32768,32768,8192,32768,32768,32768,14043,2849,32768,2979,6554,6554,
65507,62415,65384,61891,65273,58982,65461,55097,32768,32768,32768,55606,32768,2979,3745,16913,
61885,13827,60893,12196,60855,53248,51493,11243,56656,783,55563,143,63432,7106,52429,445,
65485,1031,65020,1380,65180,57344,65162,36536,61154,6554,26569,2341,63593,3449,65102,533,
47827,2913,57344,3449,35688,1337,32768,22938,25012,910,7944,1008,29319,607,64466,4202,
64549,57301,49152,20025,63351,61167,32768,45542,58982,14564,32768,9362,61895,44840,32768,26385,
59664,17135,60855,13291,40050,12252,32768,7816,25798,1850,60495,2662,18707,122,52538,231,
65332,32768,65210,21693,65113,6554,65141,39667,62259,32768,22258,1337,63636,32768,64255,52429,
60362,32768,6780,819,16384,32768,16384,4681,49152,32768,8985,2521,24410,683,21535,16585,
65416,46091,65292,58328,64626,32768,65016,39897,62687,47332,62805,28948,64284,53620,52870,49567,
65032,31174,63022,28312,64299,46811,48009,31453,61207,7077,50299,1514,60047,2634,46488,235
};

USInt inter_prob[512] = {
65532,62970,65148,54613,62470,8192,62577,8937,65480,64335,65195,53248,65322,62518,62891,38312,
65075,53405,63980,58982,32768,32768,54613,32768,65238,60009,60075,32768,59294,19661,61203,13107,
63000,9830,62566,58982,11565,32768,25215,3277,53620,50972,63109,43691,54613,32768,39671,17129,
59788,6068,43336,27913,6554,32768,12178,1771,56174,49152,60075,43691,58254,16384,49152,9930,
23130,7282,40960,32768,10923,32768,32768,32768,27307,32768,32768,32768,32768,32768,32768,32768,
36285,12511,10923,32768,45875,16384,32768,32768,16384,23831,4369,32768,8192,10923,32768,32768,
10175,2979,18978,10923,54613,32768,6242,6554,1820,10923,32768,32768,32768,32768,32768,5461,
28459,593,11886,2030,3121,4681,1292,112,42130,23831,49152,29127,32768,6554,5461,2048,
65331,64600,63811,63314,42130,19661,49152,32768,65417,64609,62415,64617,64276,44256,61068,36713,
64887,57525,53620,61375,32768,8192,57344,6554,63608,49809,49152,62623,32768,15851,58982,34162,
55454,51739,64406,64047,32768,32768,7282,32768,49152,58756,62805,64990,32768,14895,16384,19418,
57929,24966,58689,31832,32768,16384,10923,6554,54613,42882,57344,64238,58982,10082,20165,20339,
62687,15061,32768,10923,32768,10923,32768,16384,59578,34427,32768,16384,32768,7825,32768,7282,
58052,23400,32768,5041,32768,2849,32768,32768,47663,15073,57344,4096,32768,1176,32768,1320,
24858,410,24576,923,32768,16384,16384,5461,16384,1365,32768,5461,32768,5699,8192,13107,
46884,2361,23559,424,19661,712,655,182,58637,2094,49152,9362,8192,85,32768,1228,
65486,49152,65186,49152,61320,32768,57088,25206,65352,63047,62623,49152,64641,62165,58986,18304,
64171,16384,60855,54613,42130,32768,61335,32768,58254,58982,49152,32768,60985,35289,64520,31554,
51067,32768,64074,32768,40330,32768,34526,4096,60855,32768,63109,58254,57672,16384,31009,2567,
23406,32768,44620,10923,32768,32768,32099,10923,49152,49152,54613,60075,63422,54613,46388,39719,
58982,32768,54613,32768,14247,32768,22938,5041,32768,49152,32768,32768,25321,6144,29127,10999,
41263,32768,46811,32768,267,4096,426,16384,32768,19275,49152,32768,1008,1437,5767,11275,
5595,5461,37493,6554,4681,32768,6147,1560,38229,10923,32768,40960,35747,2521,5999,312,
17052,2521,18808,3641,213,2427,574,32,51493,42130,42130,53053,11155,312,2069,106,
64406,45197,58982,32768,32768,16384,40960,36864,65336,64244,60075,61681,65269,50748,60340,20515,
58982,23406,57344,32768,6554,16384,19661,61564,60855,47480,32768,54613,46811,21701,54909,37826,
32768,58982,60855,60855,32768,32768,39322,49152,57344,45875,60855,55706,32768,24576,62313,25038,
54613,8192,49152,10923,32768,32768,32768,32768,32768,19661,16384,51493,32768,14043,40050,44651,
59578,5174,32768,6554,32768,5461,23593,5461,63608,51825,32768,23831,58887,24032,57170,3298,
39322,12971,16384,49152,1872,618,13107,2114,58982,25705,32768,60075,28913,949,18312,1815,
48188,114,51493,1542,5461,3855,11360,1163,58982,7215,54613,21487,49152,4590,48430,1421,
28944,1319,6868,324,1456,232,820,7,61681,1864,60855,9922,4369,315,6589,14
};
```

11.1.6 Sprite Coding

Table 11-32 Code table for the first trajectory point

dmv value	SSS	VLC	dmv_code
-16383 … -8192 8192 … 16383	14	111111111 110	00000000000000...01111111111111, 10000000000000...11111111111111
-8191 … -4096 4096 … 8191	13	111111111 10	0000000000000...0111111111111, 1000000000000...1111111111111
-4095 … -2048 2048 … 4095	12	111111111 0	000000000000...011111111111, 100000000000...111111111111
-2047...-1024, 1024...2047	11	111111110	00000000000...01111111111, 10000000000...11111111111
-1023...-512, 512...1023	10	11111110	0000000000...0111111111, 1000000000...1111111111
-511...-256, 256...511	9	1111110	000000000...011111111, 100000000...111111111
-255...-128, 128...255	8	111110	00000000...01111111, 10000000...11111111
-127...-64, 64...127	7	11110	0000000...0111111, 1000000...1111111
-63...-32, 32...63	6	1110	000000...011111, 100000...111111
-31...-16, 16...31	5	110	00000...01111, 10000...1111
-15...-8, 8...15	4	101	0000...0111, 1000...1111
-7...-4, 4...7	3	100	000...011, 100...111
-3...-2, 2...3	2	011	00...01, 10...11
-1, 1	1	010	0, 1
0	0	00	-

Table 11-33 Code table for scaled brightness change factor

brightness_change_factor value	brightness_change_factor_length value	brightness_change_factor_length VLC	brightness_change_factor
-16...-1, 1...16	1	0	00000...01111, 10000...11111
-48...-17, 17...48	2	10	000000...011111, 100000...111111

112...-49, 49...112	3	110	0000000...0111111, 1000000...1111111
113…624	4	1110	000000000...111111111
625...1648	4	1111	0000000000…1111111111

11.1.7 DCT based facial object decoding

Table 11-34 Viseme_select_table, 29 symbols

symbol	bits	code	symbol	bits	code	symbol	bits	code
0	6	001000	10	6	010001	20	6	010000
1	6	001001	11	6	011001	21	6	010010
2	6	001011	12	5	00001	22	6	011010
3	6	001101	13	6	011101	23	5	00010
4	6	001111	14	1	1	24	6	011110
5	6	010111	15	6	010101	25	6	010110
6	6	011111	16	6	010100	26	6	001110
7	5	00011	17	6	011100	27	6	001100
8	6	011011	18	5	00000	28	6	001010
9	6	010011	19	6	011000			

Table 11-35 Expression_select_table, 13 symbols

symbol	bits	code	symbol	bits	code	symbol	bits	code
0	5	01000	5	4	0011	10	5	01110
1	5	01001	6	1	1	11	5	01100
2	5	01011	7	4	0001	12	5	01010
3	5	01101	8	4	0000			
4	5	01111	9	4	0010			

Table 11-36 Viseme and Expression intensity_table, 127 symbols

sym bol	bits	code	sym bol	bits	code	sym bol	bits	code
0	17	10010001101010010	43	16	1001000110100111	86	16	1001000110100110
1	17	10010001101010011	44	8	10011100	87	16	1001000110100100
2	17	10010001101010101	45	11	10010001111	88	16	1001000110100010
3	17	10010001101010111	46	9	100100010	89	16	1001000110100000
4	17	10010001101011001	47	10	1110001011	90	16	1001000110011110
5	17	10010001101011011	48	9	100011011	91	16	1001000110011100
6	17	10010001101011101	49	10	1110001001	92	16	1001000110011010
7	17	10010001101011111	50	9	100011010	93	16	1001000110011000
8	17	10010001101100001	51	9	100111010	94	16	1001000110010110
9	17	10010001101100011	52	10	1110001000	95	16	1001000110010100
10	17	10010001101100101	53	7	1000111	96	16	1001000110010010
11	17	10010001101100111	54	7	1000010	97	16	1001000110010000
12	17	10010001101101001	55	8	10010000	98	16	1001000110001110
13	17	10010001101101011	56	7	1001111	99	16	1001000110001100
14	17	10010001101101101	57	7	1110000	100	16	1001000110001010
15	17	10010001101101111	58	6	100000	101	16	1001000110001000
16	17	10010001101110001	59	6	100101	102	16	1001000110000110
17	17	10010001101110011	60	6	111010	103	16	1001000110000100
18	17	10010001101110111	61	5	11111	104	16	1001000110000010

19	17	10010001101111001	62	3	101	105	16	1001000110000000
20	17	10010001101111011	63	1	0	106	17	10010001101111110
21	17	10010001101111101	64	3	110	107	17	10010001101111100
22	17	10010001101111111	65	5	11110	108	17	10010001101111010
23	16	1001000110000001	66	6	111001	109	17	10010001101111000
24	16	1001000110000011	67	6	111011	110	17	10010001101110110
25	16	1001000110000101	68	6	100010	111	17	10010001101110010
26	16	1001000110000111	69	7	1001100	112	17	10010001101110000
27	16	1001000110001001	70	7	1001001	113	17	10010001101101110
28	16	1001000110001011	71	7	1001101	114	17	10010001101101100
29	16	1001000110001101	72	8	10001100	115	17	10010001101101010
30	16	1001000110001111	73	8	10000111	116	17	10010001101101000
31	16	1001000110010001	74	8	10000110	117	17	10010001101100110
32	16	1001000110010011	75	17	10010001101110100	118	17	10010001101100100
33	16	1001000110010101	76	9	111000110	119	17	10010001101100010
34	16	1001000110010111	77	11	11100010100	120	17	10010001101100000
35	16	1001000110011001	78	11	10011101111	121	17	10010001101011110
36	16	1001000110011011	79	17	10010001101110101	122	17	10010001101011100
37	16	1001000110011101	80	10	1001110110	123	17	10010001101011010
38	16	1001000110011111	81	16	1001000110101000	124	17	10010001101011000
39	16	1001000110100001	82	11	10010001110	125	17	10010001101010110
40	16	1001000110100011	83	10	1110001111	126	17	10010001101010100
41	11	11100010101	84	11	10011101110			
42	16	1001000110100101	85	10	1110001110			

Table 11-37 Runlength_table, 16 symbols

symbol	bits	code	symbol	bits	code	symbol	bits	code
0	1	1	6	9	000001011	12	8	00000000
1	2	01	7	9	000001101	13	8	00000010
2	3	001	8	9	000001111	14	9	000001110
3	4	0001	9	8	00000011	15	9	000001100
4	5	00001	10	8	00000001			
5	9	000001010	11	8	00000100			

Table 11-38 DC_table, 512 symbols

sym bol	bits	code	sym bol	bits	code	sym bol	bit s	code
0	17	11010111001101010	171	17	11010111001111001	342	17	11010111001111000
1	17	11010111001101011	172	17	11010111010000001	343	17	11010111001110000
2	17	11010111001101101	173	17	11010111010001001	344	17	11010111001110010
3	17	11010111001101111	174	17	11010111010010001	345	17	11010111001111010
4	17	11010111001110101	175	17	11010111010011001	346	17	11010111010000010
5	17	11010111001110111	176	17	11010111010101001	347	17	11010111010001010
6	17	11010111001111101	177	17	11010111010110001	348	17	11010111010010010
7	17	11010111001111111	178	17	11010111010111001	349	17	11010111010011010
8	17	11010111010000101	179	17	11010111011000001	350	17	11010111010101010
9	17	11010111010000111	180	17	11010111011001001	351	17	11010111010110010
10	17	11010111010001101	181	17	11010111011011001	352	17	11010111010111010
11	17	11010111010001111	182	17	11010111011111001	353	17	11010111011000010
12	17	11010111010010101	183	17	11010111100000001	354	17	11010111011001010
13	17	11010111010010111	184	17	11010111100001001	355	17	11010111011011010
14	17	11010111010011101	185	17	11010111100011001	356	17	11010111011111010
15	17	11010111010011111	186	17	11010111100100001	357	17	11010111100000010
16	17	11010111010101101	187	17	11010111100101001	358	17	11010111100001010
17	17	11010111010101111	188	17	11010111100111001	359	17	11010111100011010
18	17	11010111010110111	189	17	11010111101000001	360	17	11010111100100010
19	17	11010111010111101	190	17	11010111101001001	361	17	11010111100101010
20	17	11010111010111111	191	17	11010111101011001	362	17	11010111100111010
21	17	11010111011000111	192	17	11010111101111001	363	17	11010111101000010
22	17	11010111011001101	193	17	11010111110000001	364	17	11010111101001010
23	17	11010111011001111	194	17	11010111110001001	365	17	11010111101011010
24	17	11010111011011101	195	17	11010111110011001	366	17	11010111101111010
25	17	11010111011011111	196	17	11010111110111001	367	17	11010111110000010
26	17	11010111011111101	197	17	11010111111100001	368	17	11010111110001010
27	17	11010111011111111	198	17	11010111111101001	369	17	11010111110011010
28	17	11010111100000111	199	17	11010111111111001	370	17	11010111110111010
29	17	11010111100001101	200	16	1101011100000001	371	17	11010111111100010
30	17	11010111100001111	201	16	1101011100001001	372	17	11010111111101010
31	17	11010111100011101	202	16	1101011100011001	373	17	11010111111111010
32	17	11010111100011111	203	17	11010111111001001	374	16	1101011100000010
33	17	11010111100100101	204	17	11010111111010001	375	16	1101011100001010
34	17	11010111100100111	205	17	11010111111011001	376	16	1101011100011010
35	17	11010111100101101	206	16	1101011100101001	377	17	11010111111001010
36	17	11010111100101111	207	17	11010111110100001	378	17	11010111111010010
37	17	11010111100111101	208	17	11010111110101001	379	17	11010111111011010
38	17	11010111100111111	209	17	11010111101101001	380	16	1101011100101010
39	17	11010111101000101	210	17	11010111011100001	381	17	11010111110100010
40	17	11010111101000111	211	16	1101011100100000	382	17	11010111110101010
41	17	11010111101001101	212	16	1101011100100001	383	17	11010111101101010
42	17	11010111101001111	213	17	11010111111000001	384	17	11010111011100010
43	17	11010111101011101	214	16	1101011100010001	385	17	11010111011101010
44	17	11010111101011111	215	17	11010111111110001	386	17	11010111011101000
45	17	11010111101111101	216	17	11010111110110001	387	16	1101011100100010
46	17	11010111101111111	217	17	11010111110010001	388	17	11010111111000010
47	17	11010111110000101	218	11	11101100101	389	16	1101011100010010
48	17	11010111110000111	219	11	11011111011	390	17	11010111111110010
49	17	11010111110001101	220	11	11011110001	391	17	11010111110110010
50	17	11010111110001111	221	10	1101110011	392	17	11010111110010010

51	17	11010111110011101	222	17	11010111101110001	393	17	11010111101110010
52	17	11010111110011111	223	17	11010111010100000	394	17	11010111101010010
53	17	11010111110111101	224	17	11010111010100001	395	17	11010111101010000
54	17	11010111110111111	225	17	11010111011110100	396	17	11010111100010010
55	17	11010111111100101	226	17	11010111011110101	397	17	11010111100010000
56	17	11010111111100111	227	17	11010111011110001	398	17	11010111011010010
57	17	11010111111101101	228	17	11010111100010101	399	17	11010111011010000
58	17	11010111111101111	229	17	11010111100110000	400	16	1101011100110010
59	17	11010111111111101	230	17	11010111100110001	401	16	1101011100110000
60	17	11010111111111111	231	17	11010111101010101	402	17	11010111010100110
61	16	1101011100000101	232	11	11101100111	403	17	11010111010100100
62	16	1101011100000111	233	17	11010111101110101	404	17	11010111010100010
63	16	1101011100001101	234	11	11101100110	405	17	11010111011010110
64	16	1101011100001111	235	17	11010111110110101	406	17	11010111011010100
65	16	1101011100011101	236	17	11010111111000100	407	17	11010111011110110
66	16	1101011100011111	237	8	11010110	408	17	11010111011110010
67	17	11010111111001101	238	11	11011110010	409	17	11010111100010110
68	17	11010111111001111	239	9	110010100	410	17	11010111100110110
69	17	11010111111010101	240	10	1101110001	411	17	11010111100110100
70	17	11010111111010111	241	9	110001111	412	17	11010111100110010
71	17	11010111111011101	242	10	1101111100	413	17	11010111101010110
72	17	11010111111011111	243	9	110010101	414	17	11010111101110110
73	16	1101011100101101	244	9	110111111	415	17	11010111110010110
74	16	1101011100101111	245	10	1101110100	416	17	11010111110010100
75	17	11010111110100101	246	7	1100100	417	17	11010111110110110
76	17	11010111110100111	247	8	11101101	418	17	11010111111110110
77	17	11010111110101101	248	8	11001011	419	17	11010111111110100
78	17	11010111110101111	249	7	1101100	420	16	1101011100010110
79	17	11010111101101101	250	7	1101101	421	16	1101011100010100
80	17	11010111101101111	251	7	1110111	422	17	11010111111000110
81	17	11010111011100101	252	6	110100	423	16	1101011100100110
82	17	11010111011100111	253	6	111001	424	16	1101011100100100
83	17	11010111011101101	254	5	11111	425	17	11010111101100110
84	17	11010111011101111	255	3	100	426	17	11010111101100100
85	17	11010111101100001	256	1	0	427	17	11010111101100010
86	17	11010111101100011	257	3	101	428	17	11010111101100000
87	17	11010111101100101	258	5	11110	429	17	11010111011101110
88	17	11010111101100111	259	6	111000	430	17	11010111011101100
89	16	1101011100100101	260	6	111010	431	17	11010111011100110
90	16	1101011100100111	261	6	110000	432	17	11010111011100100
91	17	11010111111000111	262	7	1100111	433	17	11010111101101110
92	16	1101011100010101	263	7	1100110	434	17	11010111101101100
93	16	1101011100010111	264	7	1101010	435	17	11010111110101110
94	17	11010111111110101	265	8	11000101	436	17	11010111110101100
95	17	11010111111110111	266	8	11000110	437	17	11010111110100110
96	17	11010111110110111	267	8	11000100	438	17	11010111110100100
97	17	11010111110010101	268	17	11010111111000101	439	16	1101011100101110
98	17	11010111110010111	269	9	111011000	440	16	1101011100101100
99	17	11010111101110111	270	11	11011111010	441	17	11010111111011110
100	17	11010111101010111	271	11	11011110101	442	17	11010111111011100
101	17	11010111100110011	272	17	11010111100000101	443	17	11010111111010110
102	17	11010111100110101	273	10	1101111011	444	17	11010111111010100
103	17	11010111100110111	274	17	11010111011000101	445	17	11010111111001110

104	17	11010111100010111	275	11	11011110011	446	17	11010111111001100
105	17	11010111011110011	276	9	110001110	447	16	1101011100011110
106	17	11010111011110111	277	11	11011110000	448	16	1101011100011100
107	17	11010111011010101	278	10	1101110111	449	16	1101011100001110
108	17	11010111011010111	279	17	11010111010110101	450	16	1101011100001100
109	17	11010111010100011	280	16	1101011100110100	451	16	1101011100000110
110	17	11010111010100101	281	10	1101110010	452	16	1101011100000100
111	17	11010111010100111	282	10	1101110000	453	17	11010111111111110
112	16	1101011100110001	283	11	11011101010	454	17	11010111111111100
113	16	1101011100110011	284	17	11010111010110100	455	17	11010111111101110
114	17	11010111011010001	285	17	11010111011000100	456	17	11010111111101100
115	17	11010111011010011	286	17	11010111100000100	457	17	11010111111100110
116	17	11010111100010001	287	11	11011101100	458	17	11010111111100100
117	17	11010111100010011	288	17	11010111110110100	459	17	11010111110111110
118	17	11010111101010001	289	17	11010111101110100	460	17	11010111110111100
119	17	11010111101010011	290	17	11010111101010100	461	17	11010111110011110
120	17	11010111101110011	291	11	11101100100	462	17	11010111110011100
121	17	11010111110010011	292	17	11010111100010100	463	17	11010111110001110
122	17	11010111110110011	293	17	11010111011110000	464	17	11010111110001100
123	17	11010111111110011	294	11	11011110100	465	17	11010111110000110
124	16	1101011100010011	295	11	11011101011	466	17	11010111110000100
125	17	11010111111000011	296	17	11010111101110000	467	17	11010111101111110
126	16	1101011100100011	297	17	11010111110010000	468	17	11010111101111100
127	17	11010111011101001	298	17	11010111110110000	469	17	11010111101011110
128	17	11010111011101011	299	17	11010111111110000	470	17	11010111101011100
129	17	11010111011100011	300	16	1101011100010000	471	17	11010111101001110
130	17	11010111101101011	301	17	11010111111000000	472	17	11010111101001100
131	17	11010111110101011	302	11	11011101101	473	17	11010111101000110
132	17	11010111110100011	303	17	11010111011100000	474	17	11010111101000100
133	16	1101011100101011	304	17	11010111101101000	475	17	11010111100111110
134	17	11010111111011011	305	17	11010111110101000	476	17	11010111100111100
135	17	11010111111010011	306	17	11010111110100000	477	17	11010111100101110
136	17	11010111111001011	307	16	1101011100101000	478	17	11010111100101100
137	16	1101011100011011	308	17	11010111111011000	479	17	11010111100100110
138	16	1101011100001011	309	17	11010111111010000	480	17	11010111100100100
139	16	1101011100000011	310	17	11010111111001000	481	17	11010111100011110
140	17	11010111111111011	311	16	1101011100011000	482	17	11010111100011100
141	17	11010111111101011	312	16	1101011100001000	483	17	11010111100001110
142	17	11010111111100011	313	16	1101011100000000	484	17	11010111100001100
143	17	11010111110111011	314	17	11010111111111000	485	17	11010111100000110
144	17	11010111110011011	315	17	11010111111101000	486	17	11010111011111110
145	17	11010111110001011	316	17	11010111111100000	487	17	11010111011111100
146	17	11010111110000011	317	17	11010111110111000	488	17	11010111011011110
147	17	11010111101111011	318	17	11010111110011000	489	17	11010111011011100
148	17	11010111101011011	319	17	11010111110001000	490	17	11010111011001110
149	17	11010111101001011	320	17	11010111110000000	491	17	11010111011001100
150	17	11010111101000011	321	17	11010111101111000	492	17	11010111011000110
151	17	11010111100111011	322	17	11010111101011000	493	17	11010111010111110
152	17	11010111100101011	323	17	11010111101001000	494	17	11010111010111100
153	17	11010111100100011	324	17	11010111101000000	495	17	11010111010110110
154	17	11010111100011011	325	17	11010111100111000	496	17	11010111010101110
155	17	11010111100001011	326	17	11010111100101000	497	17	11010111010101100
156	17	11010111100000011	327	17	11010111100100000	498	17	11010111010011110

157	17	11010111011111011	328	17	11010111100011000	499	17	11010111010011100
158	17	11010111011011011	329	17	11010111100001000	500	17	11010111010010110
159	17	11010111011001011	330	17	11010111100000000	501	17	11010111010010100
160	17	11010111011000011	331	17	11010111011111000	502	17	11010111010001110
161	17	11010111010111011	332	17	11010111011011000	503	17	11010111010001100
162	17	11010111010110011	333	17	11010111011001000	504	17	11010111010000110
163	17	11010111010101011	334	17	11010111011000000	505	17	11010111010000100
164	17	11010111010011011	335	17	11010111010111000	506	17	11010111001111110
165	17	11010111010010011	336	17	11010111010110000	507	17	11010111001111100
166	17	11010111010001011	337	17	11010111010101000	508	17	11010111001110110
167	17	11010111010000011	338	17	11010111010011000	509	17	11010111001110100
168	17	11010111001111011	339	17	11010111010010000	510	17	11010111001101110
169	17	11010111001110011	340	17	11010111010001000	511	17	11010111001101100
170	17	11010111001110001	341	17	11010111010000000			

Table 11-39 AC_table, 512 symbols

symbol	no_of_bits	code	symbol	no_of_bits	code	symbol	no_of_bits	code
0	16	1000011100011000	171	16	1000011101100001	342	16	1000011101100000
1	16	1000011100011001	172	16	1000011110100001	343	15	100001110000000
2	16	1000011100011011	173	16	1000011111000001	344	16	1000011101101000
3	16	1000011100011101	174	16	1000011111100001	345	16	1000011110101000
4	16	1000011100011111	175	15	100001000100001	346	16	1000011111001000
5	16	1000011100100101	176	15	100001001100001	347	16	1000011111101000
6	16	1000011100100111	177	15	100001011000001	348	15	100001000101000
7	16	1000011100101101	178	15	100001011100001	349	15	100001001101000
8	16	1000011100101111	179	15	100001010100001	350	15	100001011001000
9	16	1000011100111101	180	15	100001010000001	351	15	100001011101000
10	16	1000011100111111	181	15	100001001000001	352	15	100001010101000
11	16	1000011101111101	182	15	100001000000001	353	15	100001010001000
12	16	1000011101111111	183	16	1000011110000001	354	15	100001001001000
13	16	1000011110111111	184	16	1000011101000001	355	15	100001000001000
14	16	1000011111011101	185	16	1000011101010001	356	16	1000011110001000
15	16	1000011111011111	186	16	1000011110010001	357	16	1000011101001000
16	16	1000011111111101	187	15	100001000010001	358	16	1000011101011000
17	16	1000011111111111	188	15	100001001010001	359	16	1000011110011000
18	15	100001000111101	189	15	100001010010001	360	15	100001000011000
19	15	100001000111111	190	15	100001010110001	361	15	100001001011000
20	15	100001001111101	191	15	100001011110001	362	15	100001010011000
21	15	100001001111111	192	15	100001011010001	363	15	100001010111000
22	15	100001011011101	193	15	100001001110001	364	15	100001011111000
23	15	100001011011111	194	15	100001000110001	365	15	100001011011000
24	15	100001011111101	195	16	1000011111110001	366	15	100001001111000
25	15	100001011111111	196	16	1000011111010001	367	15	100001000111000
26	15	100001010111111	197	16	1000011110110001	368	16	1000011111111000
27	15	100001010011101	198	16	1000011101110001	369	16	1000011111011000
28	15	100001010011111	199	16	1000011100110001	370	16	1000011110111000
29	15	100001001011111	200	15	100001110001001	371	16	1000011101111000
30	15	100001000011111	201	16	1000011100110101	372	16	1000011100111000
31	16	1000011110011111	202	16	1000011101110101	373	16	1000011100101000
32	16	1000011101011111	203	16	1000011110110101	374	16	1000011100100000
33	16	1000011101001111	204	16	1000011111010101	375	16	1000011100100010
34	16	1000011110001111	205	16	1000011111110101	376	16	1000011100101010
35	15	100001000001111	206	15	100001000110101	377	16	1000011100111010
36	15	100001001001111	207	15	100001001110101	378	16	1000011101111010
37	15	100001010001111	208	15	100001011010101	379	16	1000011110111010
38	15	100001010101111	209	15	100001011110101	380	16	1000011111011010
39	15	100001011101111	210	15	100001010110101	381	16	1000011111111010
40	15	100001011001111	211	15	100001010010101	382	15	100001000111010
41	15	100001001101111	212	15	100001001010101	383	15	100001001111010
42	15	100001000101111	213	15	100001000010101	384	15	100001011011010
43	16	1000011111101111	214	16	1000011110010101	385	15	100001011111010
44	16	1000011111001111	215	16	1000011101010101	386	15	100001010111010
45	16	1000011110101111	216	16	1000011101000101	387	15	100001010011010
46	16	1000011101101111	217	16	1000011110000101	388	15	100001001011010
47	15	100001110000111	218	15	100001000000101	389	15	100001000011010
48	16	1000011101100111	219	15	100001001000101	390	16	1000011110011010
49	16	1000011110100111	220	15	100001010000101	391	16	1000011101011010
50	16	1000011111000111	221	15	100001010100101	392	16	1000011101001010

51	16	1000011111100111	222	15	100001011100101	393	16	1000011110001010
52	15	100001000100111	223	15	100001011000101	394	15	100001000001010
53	15	100001001100111	224	15	100001001100101	395	15	100001001001010
54	15	100001011000111	225	15	100001000100101	396	15	100001010001010
55	15	100001011100111	226	16	1000011111100101	397	15	100001010101010
56	15	100001010100111	227	16	1000011111000101	398	15	100001011101010
57	15	100001010000111	228	16	1000011110100101	399	15	100001011001010
58	15	100001001000111	229	16	1000011101100101	400	15	100001001101010
59	15	100001000000111	230	15	100001110000101	401	15	100001000101010
60	16	1000011110000111	231	16	1000011101101101	402	16	1000011111101010
61	16	1000011101000111	232	16	1000011110101101	403	16	1000011111001010
62	16	1000011101010111	233	16	1000011111001101	404	16	1000011110101010
63	16	1000011110010111	234	16	1000011111101101	405	16	1000011101101010
64	15	100001000010111	235	15	100001000101101	406	15	100001110000010
65	15	100001001010111	236	15	100001001101101	407	16	1000011101100010
66	15	100001010010111	237	15	100001011001101	408	16	1000011110100010
67	15	100001010110111	238	15	100001011101101	409	16	1000011111000010
68	15	100001011110111	239	15	100001010101101	410	16	1000011111100010
69	15	100001011010111	240	15	100001010001101	411	15	100001000100010
70	15	100001001110111	241	15	100001001001101	412	15	100001001100010
71	15	100001000110111	242	15	100001000001101	413	15	100001011000010
72	16	1000011111110111	243	16	1000011110001101	414	15	100001011100010
73	16	1000011111010111	244	16	1000011101001101	415	15	100001010100010
74	16	1000011110110111	245	16	1000011101011101	416	15	100001010000010
75	16	1000011101110111	246	16	1000011110011101	417	15	100001001000010
76	16	1000011100110111	247	15	100001000011101	418	15	100001000000010
77	15	100001110001011	248	6	100000	419	16	1000011110000010
78	16	1000011100110011	249	15	100001001011101	420	16	1000011101000010
79	16	1000011101110011	250	15	100001010111101	421	16	1000011101010010
80	16	1000011110110011	251	7	1001110	422	16	1000011110010010
81	16	1000011111010011	252	6	100110	423	15	100001000010010
82	16	1000011111110011	253	5	10010	424	15	100001001010010
83	15	100001000110011	254	4	1010	425	15	100001010010010
84	15	100001001110011	255	2	11	426	15	100001010110010
85	15	100001011010011	256	16	1000011110111100	427	15	100001011110010
86	15	100001011110011	257	1	0	428	15	100001011010010
87	15	100001010110011	258	4	1011	429	15	100001001110010
88	15	100001010010011	259	6	100011	430	15	100001000110010
89	15	100001001010011	260	6	100010	431	16	1000011111110010
90	15	100001000010011	261	7	1001111	432	16	1000011111010010
91	16	1000011110010011	262	16	1000011110111101	433	16	1000011110110010
92	16	1000011101010011	263	8	10000110	434	16	1000011101110010
93	16	1000011101000011	264	15	100001010111100	435	16	1000011100110010
94	16	1000011110000011	265	15	100001001011100	436	15	100001110001010
95	15	100001000000011	266	15	100001000011100	437	16	1000011100110110
96	15	100001001000011	267	16	1000011110011100	438	16	1000011101110110
97	15	100001010000011	268	16	1000011101011100	439	16	1000011110110110
98	15	100001010100011	269	16	1000011101001100	440	16	1000011111010110
99	15	100001011100011	270	16	1000011110001100	441	16	1000011111110110
100	15	100001011000011	271	15	100001000001100	442	15	100001000110110
101	15	100001001100011	272	15	100001001001100	443	15	100001001110110
102	15	100001000100011	273	15	100001010001100	444	15	100001011010110
103	16	1000011111100011	274	15	100001010101100	445	15	100001011110110

104	16	1000011111000011	275	15	100001011101100	446	15	100001010110110
105	16	1000011110100011	276	15	100001011001100	447	15	100001010010110
106	16	1000011101100011	277	15	100001001101100	448	15	100001001010110
107	15	100001110000011	278	15	100001000101100	449	15	100001000010110
108	16	1000011101101011	279	16	1000011111101100	450	16	1000011110010110
109	16	1000011110101011	280	16	1000011111001100	451	16	1000011101010110
110	16	1000011111001011	281	16	1000011110101100	452	16	1000011101000110
111	16	1000011111101011	282	16	1000011101101100	453	16	1000011110000110
112	15	100001000101011	283	15	100001110000100	454	15	100001000000110
113	15	100001001101011	284	16	1000011101100100	455	15	100001001000110
114	15	100001011001011	285	16	1000011110100100	456	15	100001010000110
115	15	100001011101011	286	16	1000011111000100	457	15	100001010100110
116	15	100001010101011	287	16	1000011111100100	458	15	100001011100110
117	15	100001010001011	288	15	100001000100100	459	15	100001011000110
118	15	100001001001011	289	15	100001001100100	460	15	100001001100110
119	15	100001000001011	290	15	100001011000100	461	15	100001000100110
120	16	1000011110001011	291	15	100001011100100	462	16	1000011111100110
121	16	1000011101001011	292	15	100001010100100	463	16	1000011111000110
122	16	1000011101011011	293	15	100001010000100	464	16	1000011110100110
123	16	1000011110011011	294	15	100001001000100	465	16	1000011101100110
124	15	100001000011011	295	15	100001000000100	466	15	100001110000110
125	15	100001001011011	296	16	1000011110000100	467	16	1000011101101110
126	15	100001010011011	297	16	1000011101000100	468	16	1000011110101110
127	15	100001010111011	298	16	1000011101010100	469	16	1000011111001110
128	15	100001011111011	299	16	1000011110010100	470	16	1000011111101110
129	15	100001011011011	300	15	100001000010100	471	15	100001000101110
130	15	100001001111011	301	15	100001001010100	472	15	100001001101110
131	15	100001000111011	302	15	100001010010100	473	15	100001011001110
132	16	1000011111111011	303	15	100001010110100	474	15	100001011101110
133	16	1000011111011011	304	15	100001011110100	475	15	100001010101110
134	16	1000011110111011	305	15	100001011010100	476	15	100001010001110
135	16	1000011101111011	306	15	100001001110100	477	15	100001001001110
136	16	1000011100111011	307	15	100001000110100	478	15	100001000001110
137	16	1000011100101011	308	16	1000011111110100	479	16	1000011110001110
138	16	1000011100100011	309	16	1000011111010100	480	16	1000011101001110
139	16	1000011100100001	310	16	1000011110110100	481	16	1000011101011110
140	16	1000011100101001	311	16	1000011101110100	482	16	1000011110011110
141	16	1000011100111001	312	16	1000011100110100	483	15	100001000011110
142	16	1000011101111001	313	15	100001110001000	484	15	100001001011110
143	16	1000011110111001	314	16	1000011100110000	485	15	100001010011110
144	16	1000011111011001	315	16	1000011101110000	486	15	100001010011100
145	16	1000011111111001	316	16	1000011110110000	487	15	100001010111110
146	15	100001000111001	317	16	1000011111010000	488	15	100001011111110
147	15	100001001111001	318	16	1000011111110000	489	15	100001011111100
148	15	100001011011001	319	15	100001000110000	490	15	100001011011110
149	15	100001011111001	320	15	100001001110000	491	15	100001011011100
150	15	100001010111001	321	15	100001011010000	492	15	100001001111110
151	15	100001010011001	322	15	100001011110000	493	15	100001001111100
152	15	100001001011001	323	15	100001010110000	494	15	100001000111110
153	15	100001000011001	324	15	100001010010000	495	15	100001000111100
154	16	1000011110011001	325	15	100001001010000	496	16	1000011111111110
155	16	1000011101011001	326	15	100001000010000	497	16	1000011111111100
156	16	1000011101001001	327	16	1000011110010000	498	16	1000011111011110

157	16	1000011110001001	328	16	1000011101010000	499	16	1000011111011100
158	15	100001000001001	329	16	1000011101000000	500	16	1000011110111110
159	15	100001001001001	330	16	1000011110000000	501	16	1000011101111110
160	15	100001010001001	331	15	100001000000000	502	16	1000011101111100
161	15	100001010101001	332	15	100001001000000	503	16	1000011100111110
162	15	100001011101001	333	15	100001010000000	504	16	1000011100111100
163	15	100001011001001	334	15	100001010100000	505	16	1000011100101110
164	15	100001001101001	335	15	100001011100000	506	16	1000011100101100
165	15	100001000101001	336	15	100001011000000	507	16	1000011100100110
166	16	1000011111101001	337	15	100001001100000	508	16	1000011100100100
167	16	1000011111001001	338	15	100001000100000	509	16	1000011100011110
168	16	1000011110101001	339	16	1000011111100000	510	16	1000011100011100
169	16	1000011101101001	340	16	1000011111000000	511	16	1000011100011010
170	15	100001110000001	341	16	1000011110100000			

11.2 Arithmetic Decoding

11.2.1 Aritmetic decoding for still texture object

To fully initialize the decoder, the function ac_decoder_init is called followed by ac_model_init respectively:

```
void ac_decoder_init (ac_decoder *acd) {
    int i, t;
    acd->bits_to_go = 0;
    acd->total_bits = 0;
    acd->value = 0;
    for (i=1; i<=Code_value_bits; i++)  {
        acd->value = 2*acd->value + input_bit(acd);
    }
    acd->low = 0;
    acd->high = Top_value;
    return;
}
```

```
void ac_model_init (ac_model *acm, int nsym) {
    int i;

    acm->nsym = nsym;

    acm->freq = (unsigned short *) malloc (nsym*sizeof (unsigned short));
    check (!acm->freq, "arithmetic coder model allocation failure");
    acm->cfreq = (unsigned short *) calloc (nsym+1, sizeof (unsigned short));
    check (!acm->cfreq, "arithmetic coder model allocation failure");

    for (i=0; i<acm->nsym; i++) {
        acm->freq[i] = 1;
        acm->cfreq[i] = acm->nsym - i;
    }
    acm->cfreq[acm->nsym] = 0;

    return;
}
```

The acd is structures which contains the decoding variables and whose addresses act as handles for the decoded symbol/bit streams. The fields bits_to_go, buffer, bitstream, and bitstream_len are used to manage the bits in memory. The low, high, and fbits fields describe the scaled range corresponding to the symbols which have been decoded. The value field contains the currently seen code value inside the range. The total_bits field contains the total number of bits encoded or used for decoding so far. The values Code_value_bits and Top_value describe the maximum number of bits and the maximum size of a coded value respectively. The ac_model structure contains the variables used for that particular probability model and it's address acts as a handle. The nsym field contains the number of symbols in the symbol set, the freq field contains the table of frequency counts for each of the nsym symbols, and the cfreq field contains the cumulative frequency count derived from freq.

The bits are read from the bitstream using the function:

```
static int input_bit (ac_decoder *acd) {
    int t;
    unsigned int tmp;

    if (acd->bits_to_go==0)  {
        acd->buffer =  ace->bitstream[ace->bitstream_len++];
        acd->bits_to_go = 8;
    }

    t = acd->buffer & 0x080;
    acd->buffer <<= 1;
    acd->buffer &=  0x0ff;
    acd->total_bits += 1;
    acd->bits_to_go -= 1;
    t = t >> 7;

    return t;
}
```

The decoding process has four main steps. The first step is to decode the symbol based on the current state of the probability model (frequency counts) and the current code value (value) which is used to represent (and is a member of) the current range. The second step is to get the new range. The third step is to rescale the range and simultaneously load in new code value bits. The fourth step is to update the model. To decode symbols, the following function is called:

```
int ac_decode_symbol (ac_decoder *acd, ac_model *acm) {
    long range;
    int cum;
    int sym;

    range = (long)(acd->high-acd->low)+1;

    /*--- decode symbol ---*/
    cum = (((long)(acd->value-acd->low)+1)*(int)(acm->cfreq[0])-1)/range;
    for (sym = 0; (int)acm->cfreq[sym+1]>cum; sym++)
        /* do nothing */ ;

    check (sym<0||sym>=acm->nsym, "symbol out of range");

    /*--- Get new range ---*/
    acd->high = acd->low + (range*(int)(acm->cfreq[sym]))/(int)(acm->cfreq[0])-1;
    acd->low = acd->low +  (range*(int)(acm->cfreq[sym+1]))/(int)(acm->cfreq[0]);

    /*--- rescale and load new code value bits ---*/
    for (;;)  {
        if (acd->high<Half)  {
            /* do nothing */
        }  else if (acd->low>=Half)  {
            acd->value -= Half;
            acd->low -= Half;
            acd->high -= Half;
        }  else if (acd->low>=First_qtr && acd->high<Third_qtr)  {
            acd->value -= First_qtr;
            acd->low -= First_qtr;
            acd->high -= First_qtr;
        }  else
            break;
        acd->low = 2*acd->low;
        acd->high = 2*acd->high+1;
        acd->value = 2*acd->value + input_bit(acd);
    }

    /*--- Update probability model ---*/
    update_model (acm, sym);

    return sym;
}
```

The bits_plus_follow function mentioned above calls another function, output_bit. They are:

```
static void output_bit (ac_encoder *ace, int bit) {
    ace->buffer <<= 1;
    if (bit)
        ace->buffer |= 0x01;

    ace->bits_to_go -= 1;
    ace->total_bits += 1;
    if (ace->bits_to_go==0)  {

        if (ace->bitstream) {
            if (ace->bitstream_len >= MAX_BUFFER)
                if ((ace->bitstream = (uChar *)realloc(ace->bitstream, sizeof(uChar)*
                    (ace->bitstream_len/MAX_BUFFER+1)*MAX_BUFFER))==NULL) {
                    fprintf(stderr, "Couldn't reallocate memory for ace->bitstream in output_bit.\n");
                    exit(-1);
                }

            ace->bitstream[ace->bitstream_len++] = ace->buffer;
        }
        ace->bits_to_go = 8;
    }

    return;
}


static void bit_plus_follow (ac_encoder *ace, int bit) {
    output_bit (ace, bit);
    while (ace->fbits > 0)  {
        output_bit (ace, !bit);
        ace->fbits -= 1;
    }

    return;
}
```

The update of the probability model used in the decoding of the symbols is shown in the following function:

```
static void update_model (ac_model *acm, int sym)
{
    int i;

    if (acm->cfreq[0]==Max_frequency)  {
        int cum = 0;
        acm->cfreq[acm->nsym] = 0;
        for (i = acm->nsym-1; i>=0; i--)  {
            acm->freq[i] = ((int)acm->freq[i] + 1) / 2;
            cum += acm->freq[i];
            acm->cfreq[i] = cum;
        }
    }

    acm->freq[sym] += 1;
    for (i=sym; i>=0; i--)
        acm->cfreq[i] += 1;

    return;
}
```

This function simply updates the frequency counts based on the symbol just decoded. It also makes sure that the maximum frequency allowed is not exceeded. This is done by rescaling all frequency counts by 2.

11.2.2 Arithmetic decoding for shape decoding

11.2.2.1 Structures and Typedefs

```
typedef void Void;
typedef int Int;
typedef unsigned short int USInt;
#define  CODE_BIT 32
#define  HALF   ((unsigned) 1 << (CODE_BITS-1))
#define  QUARTER  (1 << (CODE_BITS-2))
struct arcodec {
   UInt L; /* lower bound */
   UInt R; /* code range */
   UInt V; /* current code value */
   UInt arpipe;
   Int bits_to_follow; /* follow bit count */
   Int first_bit;
   Int nzeros;
   Int nonzero;
   Int nzerosf;
   Int extrabits;
};
typedef struct arcodec ArCoder;
typedef struct arcodec ArDecoder;
#define MAXHEADING 8
#define MAXMIDDLE 16
#define MAXTRAILING 8
```

11.2.2.2 Decoder Source

```
Void StartArDecoder(ArDecoder *decoder, Bitstream *bitstream) {
   Int i,j;
```

```
      decoder->V = 0;
      decoder->nzerosf = MAXHEADING;
      decoder->extrabits = 0;
      for (i = 1; i<CODE_BITS; i++) {
         j=BitstreamLookBit(bitstream,i+decoder->extrabits);
         decoder->V += decoder->V + j;
         if (j == 0) {
            decoder->nzerosf--;
            if (decoder->nzerosf == 0) {
               decoder->extrabits++;
               decoder->nzerosf = MAXMIDDLE;
            }
         }
         else
            decoder->nzerosf = MAXMIDDLE;
      }
      decoder->L = 0;
      decoder->R = HALF - 1;
      decoder->bits_to_follow = 0;
      decoder->arpipe = decoder->V;
      decoder->nzeros = MAXHEADING;
      decoder->nonzero = 0;
   }
   Void StopArDecoder(ArDecoder *decoder, Bitstream *bitstream) {
      Int a = decoder->L >> (CODE_BITS-3);
      Int b = (decoder->R + decoder->L) >> (CODE_BITS-3);
      Int nbits,i;
      if (b == 0)
         b = 8;
      if (b-a >= 4 || (b-a == 3 && a&1))
         nbits = 2;
      else
         nbits = 3;
      for (i = 1; i <= nbits-1; i++)
         AddNextInputBit(bitstream, decoder);
      if (decoder->nzeros < MAXMIDDLE-MAXTRAILING || decoder->nonzero == 0)
         BitstreamFlushBits(bitstream,1);
   }
   Void AddNextInputBit(Bitstream *bitstream, ArDecoder *decoder) {
      Int i;
      if (((decoder->arpipe >> (CODE_BITS-2))&1) == 0) {
         decoder->nzeros--;
         if (decoder->nzeros == 0) {
            BitstreamFlushBits(bitstream,1);
            decoder->extrabits--;
            decoder->nzeros = MAXMIDDLE;
            decoder->nonzero = 1;
         }
      }
      else {
         decoder->nzeros = MAXMIDDLE;
         decoder->nonzero = 1;
      }
      BitstreamFlushBits(bitstream,1);
      i = (Int)BitstreamLookBit(bitstream, CODE_BITS-1+decoder->extrabits);
      decoder->V += decoder->V + i;
      decoder->arpipe += decoder->arpipe + i;
      if (i == 0) {
         decoder->nzerosf--;
         if (decoder->nzerosf == 0) {
            decoder->nzerosf = MAXMIDDLE;
            decoder->extrabits++;
         }
      }
      else
         decoder->nzerosf = MAXMIDDLE;
```

```
}
Int ArDecodeSymbol(USInt c0, ArDecoder *decoder, Bitstream *bitstream ) {
   Int bit;
   Int c1 = (1<<16) - c0;
   Int LPS = c0 > c1;
   Int cLPS = LPS ? c1 : c0;
   unsigned long rLPS;
   rLPS = ((decoder->R) >> 16) * cLPS;
   if ((decoder->V - decoder->L) >= (decoder->R - rLPS)) {
      bit = LPS;
      decoder->L += decoder->R - rLPS;
      decoder->R = rLPS;
   }
   else {
      bit = (1-LPS);
      decoder->R -= rLPS;
   }
   DECODE_RENORMALISE(decoder,bitstream);
   return(bit);
}
Void DECODE_RENORMALISE(ArDecoder *decoder,  Bitstream *bitstream) {
   while (decoder->R < QUARTER) {
      if (decoder->L >= HALF) {
         decoder->V -= HALF;
         decoder->L -= HALF;
         decoder->bits_to_follow = 0;
      }
      else
         if (decoder->L + decoder->R <= HALF)
            decoder->bits_to_follow = 0;
         else{
            decoder->V -= QUARTER;
            decoder->L -= QUARTER;
            (decoder->bits_to_follow)++;
         }
      decoder->L += decoder->L;
      decoder->R += decoder->R;
      AddNextInputBit(bitstream, decoder);
   }
}
```

- BitstreamLookBit(bitstream,nbits) : Looks nbits ahead in the bitstream beginning from the current position in the bitstream and returns the bit.

- BitstreamFlushBits(bitstream,nbits) : Moves the current bitstream position forward by nbits.

The parameter c0 (used in ArDecodeSymbol()) is taken directly from the probability tables of USint inter_prob or .Usint intra_prob in Table 11-28. That is, for the pixel to be coded / decoded, c0 is the probability than this pixel is equal to zero. The value of c0 depends on the context number of the given pixel to be decoded.

11.2.3 Face Object Decoding

In FAP decoder, a symbol is decoded by using a specific model based on the syntax and by calling the following procedure which is specified in C.

```
static long low, high, code_value, bit, length, sacindex, cum, zerorun=0;

int aa_decode(int cumul_freq[ ])
{
  length = high - low + 1;
  cum = (-1 + (code_value - low + 1) * cumul_freq[0]) / length;
  for (sacindex = 1; cumul_freq[sacindex] > cum; sacindex++);
  high = low - 1 + (length * cumul_freq[sacindex-1]) / cumul_freq[0];
  low += (length * cumul_freq[sacindex]) / cumul_freq[0];

  for ( ; ; ) {
    if (high < q2) ;
    else if (low >= q2) {
      code_value -= q2;
      low -= q2;
      high -= q2;
    }
    else if (low >= q1 && high < q3) {
      code_value -= q1;
      low -= q1;
      high -= q1;
    }
    else {
      break;
    }
    low *= 2;
    high = 2*high + 1;
    bit_out_psc_layer();
    code_value = 2*code_value + bit;
    used_bits++;
  }
  return (sacindex-1);
}

void bit_out_psc_layer()
{
  bit = getbits(1);
}
```

Again the model is specified through `cumul_freq[ ]`. The decoded symbol is returned through its index in the model. The decoder is initialized to start decoding an arithmetic coded bitstream by calling the following procedure.

```
void decoder_reset( )
{
  int i;
  zerorun = 0;        /* clear consecutive zero's counter */
  code_value = 0;
  low = 0;
  high = top;
  for (i = 1;   i <= 16;   i++) {
    bit_out_psc_layer();
    code_value = 2 * code_value + bit;
  }
  used_bits = 0;
}
```

12. Annex C

Face object decoding tables and definitions

(This annex forms an integral part of this International Standard)

FAPs names may contain letters with the following meaning: l = left, r = right, t = top, b = bottom, i = inner, o = outer, m = middle. The sum of two corresponding top and bottom eyelid FAPs must equal 1024 when the eyelids are closed. Inner lips are closed when the sum of two corresponding top and bottom lip FAPs equals zero. For example: (lower_t_midlip + raise_b_midlip) = 0 when the lips are closed. All directions are defined with respect to the face and not the image of the face.

Table 12-1 FAP definitions, group assignments and step sizes

#	FAP name	FAP description	units	Uni- or Bidir	Pos motion	Grp	FDP subgrp num	Quant step size
1	viseme	Set of values determining the mixture of two visemes for this frame (e.g. pbm, fv, th)	na	na	na	1	na	1
2	expression	A set of values determining the mixture of two facial expression	na	na	na	1	na	1
3	open_jaw	Vertical jaw displacement (does not affect mouth opening)	MNS	U	down	2	1	4
4	lower_t_midlip	Vertical top middle inner lip displacement	MNS	B	down	2	2	2
5	raise_b_midlip	Vertical bottom middle inner lip displacement	MNS	B	up	2	3	2
6	stretch_l_cornerlip	Horizontal displacement of left inner lip corner	MW	B	left	2	4	2
7	stretch_r_cornerlip	Horizontal displacement of right inner lip corner	MW	B	right	2	5	2
8	lower_t_lip_lm	Vertical displacement of midpoint between left corner and middle of top inner lip	MNS	B	down	2	6	2
9	lower_t_lip_rm	Vertical displacement of midpoint between right corner and middle of top inner lip	MNS	B	down	2	7	2
10	raise_b_lip_lm	Vertical displacement of midpoint between left corner and middle of bottom inner lip	MNS	B	up	2	8	2
11	raise_b_lip_rm	Vertical displacement of midpoint between right corner and middle of bottom inner lip	MNS	B	up	2	9	2
12	raise_l_cornerlip	Vertical displacement of left inner lip corner	MNS	B	up	2	4	2
13	raise_r_cornerlip	Vertical displacement of right inner lip corner	MNS	B	up	2	5	2
14	thrust_jaw	Depth displacement of jaw	MNS	U	forward	2	1	1
15	shift_jaw	Side to side displacement of jaw	MNS	B	right	2	1	1
16	push_b_lip	Depth displacement of bottom middle lip	MNS	B	forward	2	3	1
17	push_t_lip	Depth displacement of top middle lip	MNS	B	forward	2	2	1
18	depress_chin	Upward and compressing movement of the chin (like in sadness)	MNS	B	up	2	10	1
19	close_t_l_eyelid	Vertical displacement of top left eyelid	IRISD	B	down	3	1	1
20	close_t_r_eyelid	Vertical displacement of top right eyelid	IRISD	B	down	3	2	1

21	close_b_l_eyelid	Vertical displacement of bottom left eyelid	IRISD	B	up	3	3	1
22	close_b_r_eyelid	Vertical displacement of bottom right eyelid	IRISD	B	up	3	4	1
23	yaw_l_eyeball	Horizontal orientation of left eyeball	AU	B	left	3	na	128
24	yaw_r_eyeball	Horizontal orientation of right eyeball	AU	B	left	3	na	128
25	pitch_l_eyeball	Vertical orientation of left eyeball	AU	B	down	3	na	128
26	pitch_r_eyeball	Vertical orientation of right eyeball	AU	B	down	3	na	128
27	thrust_l_eyeball	Depth displacement of left eyeball	IRISD	B	forward	3	na	1
28	thrust_r_eyeball	Depth displacement of right eyeball	IRISD	B	forward	3	na	1
29	dilate_l_pupil	Dilation of left pupil	IRISD	U	growing	3	5	1
30	dilate_r_pupil	Dilation of right pupil	IRISD	U	growing	3	6	1
31	raise_l_i_eyebrow	Vertical displacement of left inner eyebrow	ENS	B	up	4	1	2
32	raise_r_i_eyebrow	Vertical displacement of right inner eyebrow	ENS	B	up	4	2	2
33	raise_l_m_eyebrow	Vertical displacement of left middle eyebrow	ENS	B	up	4	3	2
34	raise_r_m_eyebrow	Vertical displacement of right middle eyebrow	ENS	B	up	4	4	2
35	raise_l_o_eyebrow	Vertical displacement of left outer eyebrow	ENS	B	up	4	5	2
36	raise_r_o_eyebrow	Vertical displacement of right outer eyebrow	ENS	B	up	4	6	2
37	squeeze_l_eyebrow	Horizontal displacement of left eyebrow	ES	B	right	4	1	1
38	squeeze_r_eyebrow	Horizontal displacement of right eyebrow	ES	B	left	4	2	1
39	puff_l_cheek	Horizontal displacement of left cheeck	ES	B	left	5	1	2
40	puff_r_cheek	Horizontal displacement of right cheeck	ES	B	right	5	2	2
41	lift_l_cheek	Vertical displacement of left cheek	ENS	U	up	5	3	2
42	lift_r_cheek	Vertical displacement of right cheek	ENS	U	up	5	4	2
43	shift_tongue_tip	Horizontal displacement of tongue tip	MW	B	right	6	1	1
44	raise_tongue_tip	Vertical displacement of tongue tip	MW	B	up	6	1	1
45	thrust_tongue_tip	Depth displacement of tongue tip	MW	B	forward	6	1	1
46	raise_tongue	Vertical displacement of tongue	MW	B	up	6	2	1

47	tongue_roll	Rolling of the tongue into U shape	AU	U	concave upward	6	3, 4	512
48	head_pitch	Head pitch angle from top of spine	AU	B	down	7	na	128
49	head_yaw	Head yaw angle from top of spine	AU	B	left	7	na	128
50	head_roll	Head roll angle from top of spine	AU	B	right	7	na	128
51	lower_t_midlip _o	Vertical top middle outer lip displacement	MNS	B	down	8	1	2
52	raise_b_midlip_o	Vertical bottom middle outer lip displacement	MNS	B	up	8	2	2
53	stretch_l_cornerlip_o	Horizontal displacement of left outer lip corner	MW	B	left	8	3	2
54	stretch_r_cornerlip_o	Horizontal displacement of right outer lip corner	MW	B	right	8	4	2
55	lower_t_lip_lm _o	Vertical displacement of midpoint between left corner and middle of top outer lip	MNS	B	down	8	5	2
56	lower_t_lip_rm _o	Vertical displacement of midpoint between right corner and middle of top outer lip	MNS	B	down	8	6	2
57	raise_b_lip_lm_o	Vertical displacement of midpoint between left corner and middle of bottom outer lip	MNS	B	up	8	7	2
58	raise_b_lip_rm_o	Vertical displacement of midpoint between right corner and middle of bottom outer lip	MNS	B	up	8	8	2
59	raise_l_cornerlip_o	Vertical displacement of left outer lip corner	MNS	B	up	8	3	2
60	raise_r_cornerlip _o	Vertical displacement of right outer lip corner	MNS	B	up	8	4	2
61	stretch_l_nose	Horizontal displacement of left side of nose	ENS	B	left	9	1	1
62	stretch_r_nose	Horizontal displacement of right side of nose	ENS	B	right	9	2	1
63	raise_nose	Vertical displacement of nose tip	ENS	B	up	9	3	1
64	bend_nose	Horizontal displacement of nose tip	ENS	B	right	9	3	1
65	raise_l_ear	Vertical displacement of left ear	ENS	B	up	10	1	1
66	raise_r_ear	Vertical displacement of right ear	ENS	B	up	10	2	1
67	pull_l_ear	Horizontal displacement of left ear	ENS	B	left	10	3	1
68	pull_r_ear	Horizontal displacement of right ear	ENS	B	right	10	4	1

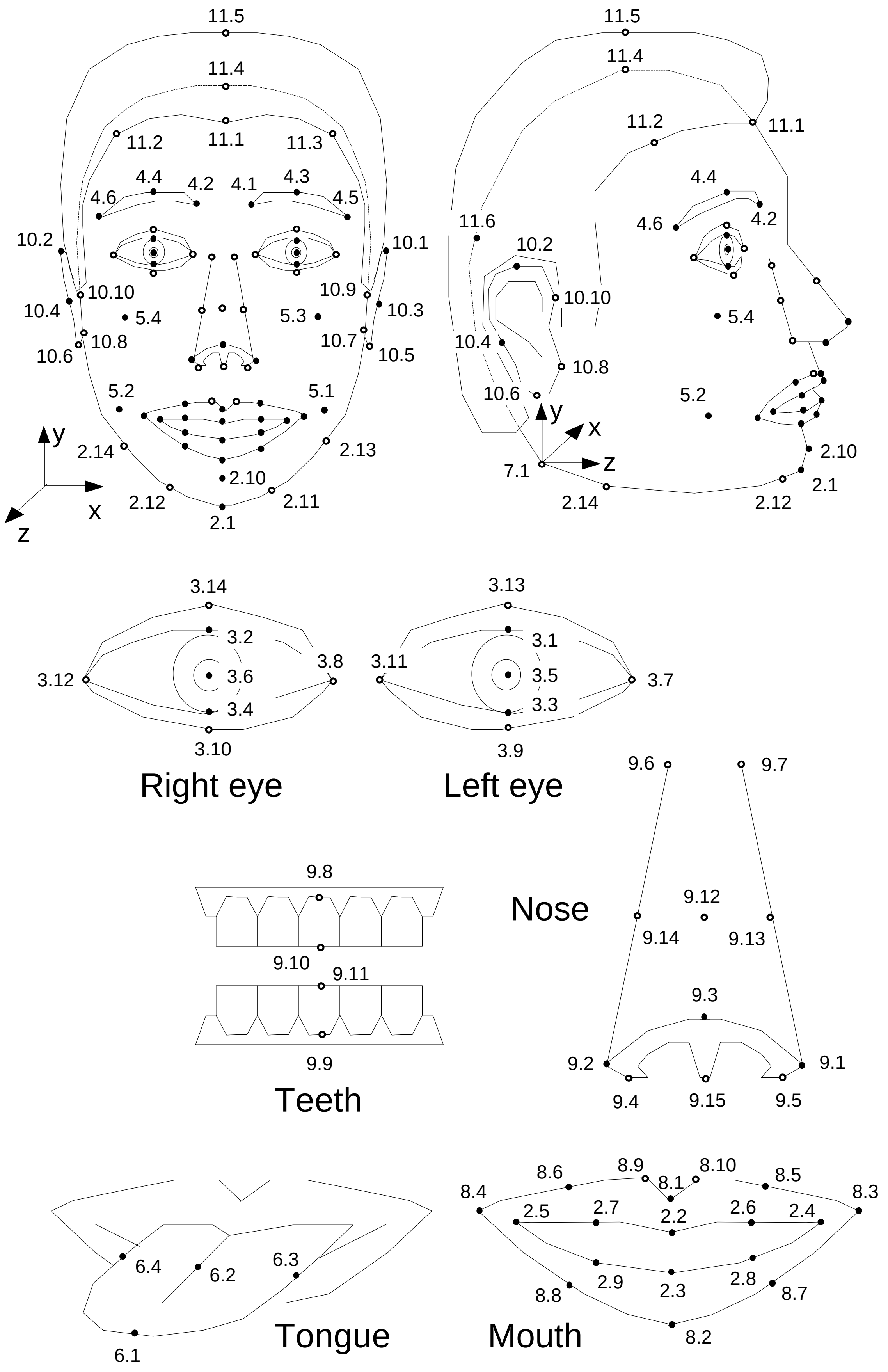
Table 12-2 FAP grouping

Group	Number of FAPs
1: visemes and expressions	2
2: jaw, chin, inner lowerlip, cornerlips, midlip	16
3: eyeballs, pupils, eyelids	12
4: eyebrow	8
5: cheeks	4
6: tongue	5
7: head rotation	3
8: outer lip positions	10
9: nose	4
10: ears	4

In the following, each facial expression is defined by a textual description and a pictorial example. (reference [10], page 114.) This reference was also used for the characteristics of the described expressions.

Table 12-3 Values for expression_select

expression_select	expression name	textual description
0	na	na
1	joy	The eyebrows are relaxed. The mouth is open and the mouth corners pulled back toward the ears.
2	sadness	The inner eyebrows are bent upward. The eyes are slightly closed. The mouth is relaxed.
3	anger	The inner eyebrows are pulled downward and together. The eyes are wide open. The lips are pressed against each other or opened to expose the teeth.
4	fear	The eyebrows are raised and pulled together. The inner eyebrows are bent upward. The eyes are tense and alert.
5	disgust	The eyebrows and eyelids are relaxed. The upper lip is raised and curled, often asymmetrically.
6	surprise	The eyebrows are raised. The upper eyelids are wide open, the lower relaxed. The jaw is opened.



• Feature points affected by FAPs

∘ Other feature points

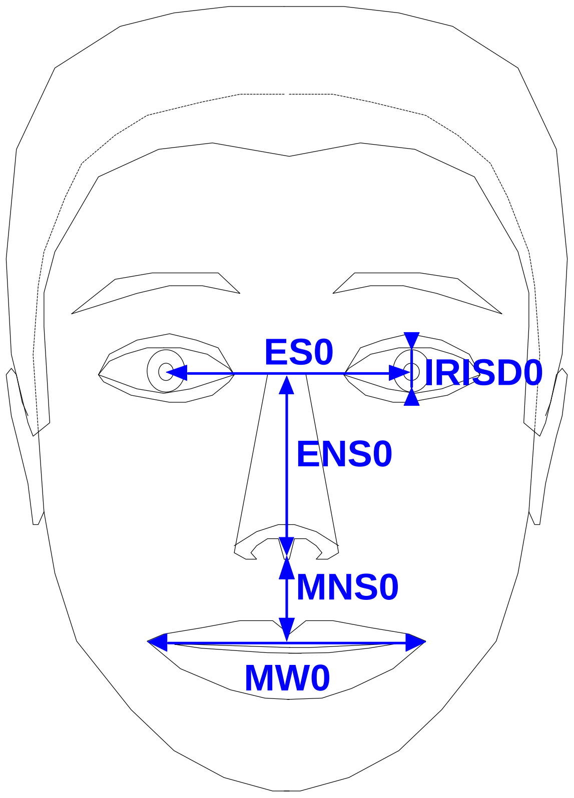


Figure 12-1 FDP feature point set

In the following, the notation 2.1.x indicates the x coordinate of feature point 2.1.

Feature points		Recommended location constraints		
#	Text description	x	y	z
2.1	Bottom of the chin	7.1.x		
2.2	Middle point of inner upper lip contour	7.1.x		
2.3	Middle point of inner lower lip contour	7.1.x		
2.4	Left corner of inner lip contour			
2.5	Right corner of inner lip contour			
2.6	Midpoint between f.p. 2.2 and 2.4 in the inner upper lip contour	(2.2.x+2.4.x)/2		
2.7	Midpoint between f.p. 2.2 and 2.5 in the inner upper lip contour	(2.2.x+2.5.x)/2		
2.8	Midpoint between f.p. 2.3 and 2.4 in the inner lower lip contour	(2.3.x+2.4.x)/2		
2.9	Midpoint between f.p. 2.3 and 2.5 in the inner lower lip contour	(2.3.x+2.5.x)/2		
2.10	Chin boss	7.1.x		
2.11	Chin left corner	> 8.7.x and < 8.3.x		
2.12	Chin right corner	> 8.4.x and < 8.8.x		
2.13	Left corner of jaw bone			
2.14	Right corner of jaw bone			
3.1	Center of upper inner left eyelid	(3.7.x+3.11.x)/2		
3.2	Center of upper inner right eyelid	(3.8.x+3.12.x)/2		
3.3	Center of lower inner left eyelid	(3.7.x+3.11.x)/2		
3.4	Center of lower inner right eyelid	(3.8.x+3.12.x)/2		
3.5	Center of the pupil of left eye			
3.6	Center of the pupil of right eye			
3.7	Left corner of left eye			
3.8	Left corner of right eye			
3.9	Center of lower outer left eyelid	(3.7.x+3.11.x)/2		
3.10	Center of lower outer right eyelid	(3.7.x+3.11.x)/2		
3.11	Right corner of left eye			
3.12	Right corner of right eye			
3.13	Center of upper outer left eyelid	(3.8.x+3.12.x)/2		
3.14	Center of upper outer right eyelid	(3.8.x+3.12.x)/2		
4.1	Right corner of left eyebrow			
4.2	Left corner of right eyebrow			
4.3	Uppermost point of the left eyebrow	(4.1.x+4.5.x)/2 or x coord of the uppermost point of the contour		
4.4	Uppermost point of the right eyebrow	(4.2.x+4.6.x)/2 or x coord of the uppermost point		

		of the contour		
4.5	Left corner of left eyebrow			
4.6	Right corner of right eyebrow			
5.1	Center of the left cheek		8.3.y	
5.2	Center of the right cheek		8.4.y	
5.3	Left cheek bone	> 3.5.x and < 3.7.x	> 9.15.y and < 9.12.y	
5.4	Right cheek bone	> 3.6.x and < 3.12.x	> 9.15.y and < 9.12.y	
6.1	Tip of the tongue	7.1.x		
6.2	Center of the tongue body	7.1.x		
6.3	Left border of the tongue			6.2.z
6.4	Right border of the tongue			6.2.z
7.1	top of spine (center of head rotation)			
8.1	Middle point of outer upper lip contour	7.1.x		
8.2	Middle point of outer lower lip contour	7.1.x		
8.3	Left corner of outer lip contour			
8.4	Right corner of outer lip contour			
8.5	Midpoint between f.p. 8.3 and 8.1 in outer upper lip contour	(8.3.x+8.1.x)/2		
8.6	Midpoint between f.p. 8.4 and 8.1 in outer upper lip contour	(8.4.x+8.1.x)/2		
8.7	Midpoint between f.p. 8.3 and 8.2 in outer lower lip contour	(8.3.x+8.2.x)/2		
8.8	Midpoint between f.p. 8.4 and 8.2 in outer lower lip contour	(8.4.x+8.2.x)/2		
8.9				
8.10				
9.1	Left nostril border			
9.2	Right nostril border			
9.3	Nose tip	7.1.x		
9.4	Bottom right edge of nose			
9.5	Bottom left edge of nose			
9.6	Left upper edge of nose bone			
9.7	Right upper edge of nose bone			
9.8	Top of the upper teeth	7.1.x		
9.9	Bottom of the lower teeth	7.1.x		
9.10	Bottom of the upper teeth	7.1.x		
9.11	Top of the lower teeth	7.1.x		
9.12	Middle lower edge of nose bone (or nose bump)	7.1.x	(9.6.y + 9.3.y)/2 or nose bump	
9.13	Left lower edge of nose bone		(9.6.y +9.3.y)/2	
9.14	Right lower edge of nose bone		(9.6.y +9.3.y)/2	
9.15	Bottom middle edge of nose	7.1.x		

10.1	Top of left ear			
10.2	Top of right ear			
10.3	Back of left ear		(10.1.y+10.5.y)/2	
10.4	Back of right ear		(10.2.y+10.6.y)/2	
10.5	Bottom of right ear lobe			
10.6	Bottom of left ear lobe			
10.7	Lower contact point between left lobe and face			
10.8	Lower contact point between right lobe and face			
10.9	Upper contact point between left ear and face			
10.10	Upper contact point between right ear and face			
11.1	Middle border between hair and forehead	7.1.x		
11.2	Right border between hair and forehead	< 4.4.x		
11.3	Left border between hair and forehead	> 4.3.x		
11.4	Top of skull	7.1.x		> 10.4.z and < 10.2.z
11.5	Hair thickness over f.p. 11.4	11.4.x		11.4.z
11.6	Back of skull	7.1.x	3.5.y	

Table 12-4 FDP fields

FDP field	**Description**
featurePointsCoord	contains a Coordinate node. Specifies feature points for the calibration of the proprietary face. The coordinates are listed in the ‘point’ field in the Coordinate node in the prescribed order, that a feature point with a lower label is listed before a feature point with at higher label (e.g. Figure 12-1 feature point 3.14 before feature point 4.1).
textureCoord4FeaturePoints	contains a TextureCoordinate node. Specifies the texture coordinates for the feature points.
calibrationMesh	contains an IndexedFaceSet node. Specifies a 3D mesh for the calibration of the proprietary face model. All fields in the IndexedFaceSet node can be used as calibration information.
faceTexture	contains an ImageTexture or PixelTexture node. Specifies texture to be applied on the proprietary face model.
animationDefinitionTables	contains AnimationDefinitionTable nodes. If a face model is downloaded, the behavior of FAPs is defined in this field.
faceSceneGraph	contains a Group node. Grouping node for face model rendered in the compositor. Can also be used to download a face model: in this case the effect of Facial Animation Parameters is defined in the ‘animationDefinitionTables’ field.

Table 12-5 Values for viseme_select

viseme_select	**phonemes**	**example**
0	none	na
1	p, b, m	put, bed, mill
2	f, v	far, voice
3	T,D	think, that
4	t, d	tip, doll
5	k, g	call, gas
6	tS, dZ, S	chair, join, she
7	s, z	sir, zeal
8	n, l	lot, not
9	r	red
10	A:	car
11	e	bed
12	I	tip
13	Q	top
14	U	book

13. Annex D
Video buffering verifier

(This annex forms an integral part of the International Standard)

Coded video bitstreams shall meet constraints imposed through a Video Buffering Verifier (VBV) defined in this clause. Each bitstream in a scalable hierarchy shall not violate the VBV constraints defined in this annex.

The VBV is a hypothetical decoder, which is conceptually connected to the output of an encoder. It has an input buffer known as the VBV buffer.

14. Annex E
Features supported by the algorithm

(This annex does not form an integral part of the International Standard)

14.1 Error resilience

14.1.1 Resynchronization

Resynchronization tools, as the name implies, attempt to enable resynchronization between the decoder and the bitstream after a residual error or errors have been detected. Generally, the data between the synchronization point prior to the error and the first point where synchronization is reestablished, is discarded. If the resynchronization approach is effective at localizing the amount of data discarded by the decoder, then the ability of other types of tools that recover data and/or conceal the effects of errors is greatly enhanced.

The resynchronization approach adopted by MPEG-4, referred to as a packet approach, is similar to the Group of Blocks (GOBs) structure utilized by the ITU-T standards H.261 and H.263. In these standards a GOB is defined as one or more rows of macroblocks (MB). At the start of a new GOB, information called a GOB header is placed within the bitstream. This header information contains a GOB start code, which is different from a picture start code, and allows the decoder to locate this GOB. Furthermore, the GOB header contains information which allows the decoding process to be restarted (i.e., resynchronize the decoder to the bitstream and reset all coded data that is predicted).

The GOB approach to resynchronization is based on spatial resynchronization. That is, once a particular macroblock location is reached in the encoding process, a resynchronization marker is inserted into the bitstream. A potential problem with this approach is that since the encoding process is variable rate, these resynchronization markers will most likely be unevenly spaced throughout the bitstream. Therefore, certain portions of the scene, such as high motion areas, will be more susceptible to errors, which will also be more difficult to conceal.

The video packet approach adopted by MPEG-4, is based on providing periodic resynchronization markers throughout the bitstream. In other words, the length of the video packets are not based on the number of macroblocks, but instead on the number of bits contained in that packet. If the number of bits contained in the current video packet exceeds a predetermined threshold, then a new video packet is created at the start of the next macroblock.

Resync Marker	**macroblock_number**	**quant_scale**	**HEC**	**Macroblock Data**	**Resync Marker**

Figure 14-1 Error Resilient Video Packet

In Figure 14-1, a typical video packet is described. A resynchronization marker is used to distinguish the start of a new video packet. This marker is distinguishable from all possible VLC code words as well as the VOP start code. Header information is also provided at the start of a video packet. Contained in this header is the information necessary to restart the decoding process and includes: the macroblock address (number) of the first macroblock contained in this packet and the quantization parameter (quant_scale) necessary to decode that first macroblock. The macroblock number provides the necessary spatial resynchronization while the quantization parameter allows the differential decoding process to be resynchronized. Following the quant_scale is the Header Extension Code (HEC). As the name implies, HEC is a single bit used to indicate whether additional information will be available in this header. If the HEC is equal to one then the following additional information is available in this packet header: modulo time base, VOP_time_increment, VOP_coding_type, intra_dc_vlc_thr, VOP_fcode_forward, VOP_fcode_backward.

The Header Extension Code makes each video packet (VP) possible to be decoded independently, when its value is equal to 1. The necessary information to decode the VP is included in the header extension code field, if the HEC is equal to 1.

If the VOP header information is corrupted by the transmission error, they can be corrected by the HEC information. The decoder can detect the error in the VOP header, if the decoded information is inconsistent with its semantics. For example, because it is prohibited that the values of the VOP_fcode_forward and VOP_fcode_backward are set to "0", if they are 0, the decoder can detect the error in the fcode information. In such a case, the decoder can correct the value by using the HEC information of the next VP.

When utilizing the error resilience tools within MPEG-4, some of the compression efficiency tools are modified. For example, all predictively encoded information must be confined within a video packet so as to prevent the propagation of errors. In other words, when predicting (i.e., AC/DC prediction and motion vector prediction) a video packet boundary is treated like a VOP boundary.

In conjunction with the video packet approach to resynchronization, a second method called fixed interval synchronization has also been adopted by MPEG-4. This method requires that VOP start codes and resynchronization markers (i.e., the start of a video packet) appear only at legal fixed interval locations in the bitstream. This helps to avoid the problems associated with start codes emulations. That is, when errors are present in a bitstream it is possible for these errors to emulate a VOP start code. In this case, when fixed interval synchronization is utilized the decoder is only required to search for a VOP start code at the beginning of each fixed interval. The fixed interval synchronization method extends this approach to be any predetermined interval.

The fixed interval synchronization is achieved by first inserting a bit with the value 0 and then, if necessary, inserting bits with the value 1 before the start code and the resync marker. The video decoder can determine if errors are injured in a video packet by detecting the incorrect number of the stuffing bits. (e.g. eight or more 1's are followed after 0 at the last part of a video packet, or the remaining bit pattern is not "011...")

14.1.2 Data Partitioning

Error concealment is an extremely important component of any error robust video codec. Similar to the error resilience tools discussed above, the effectiveness of an error concealment strategy is highly dependent on the performance of the resynchronization scheme. Basically, if the resynchronization method can effectively localize the error then the error concealment problem becomes much more tractable. For low bitrate, low delay applications the current resynchronization scheme provides very acceptable results with a simple concealment strategy, such as copying blocks from the previous frame.

In recognizing the need to provide enhanced concealment capabilities, the Video Group has developed an additional error resilient mode that further improves the ability of the decoder to localize an error. Specifically, this approach utilizes data partitioning. This data partitioning is achieved by separating the motion and macroblock header information away from the texture information. This approach requires that a second resynchronization marker be inserted between motion and texture information. Data partitioning, like the use of RVLCs, is signaled to the decoder in the VOL. Figure 14-2 illustrates the syntactic structure of the data partitioning mode. If the texture information is lost, this approach utilizes the motion information to conceal these errors. That is, due to the errors the texture information is discarded, while the motion is used to motion compensate the previously decoded VOP.

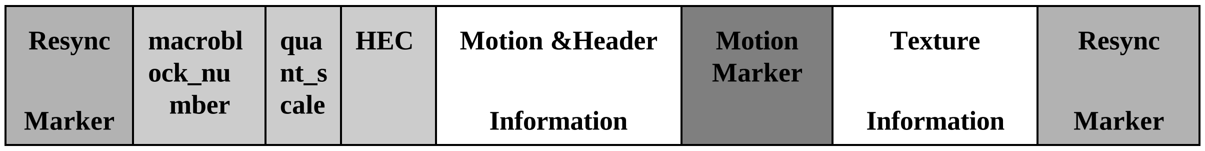


Figure 14-2 Data Partitioning

14.1.3 Reversible VLC

Reversible Variable Length Codes (RVLC) are designed such that they can be instantaneously decoded both in forward and reverse directions. A part of a bitstream which cannot be decoded in the forward direction due to the presence of errors can often be decoded in the backward direction. This is illustrated in Figure 14-3. Therefore number of discarded bits can be reduced. RVLC is applied only to TCOEF coding

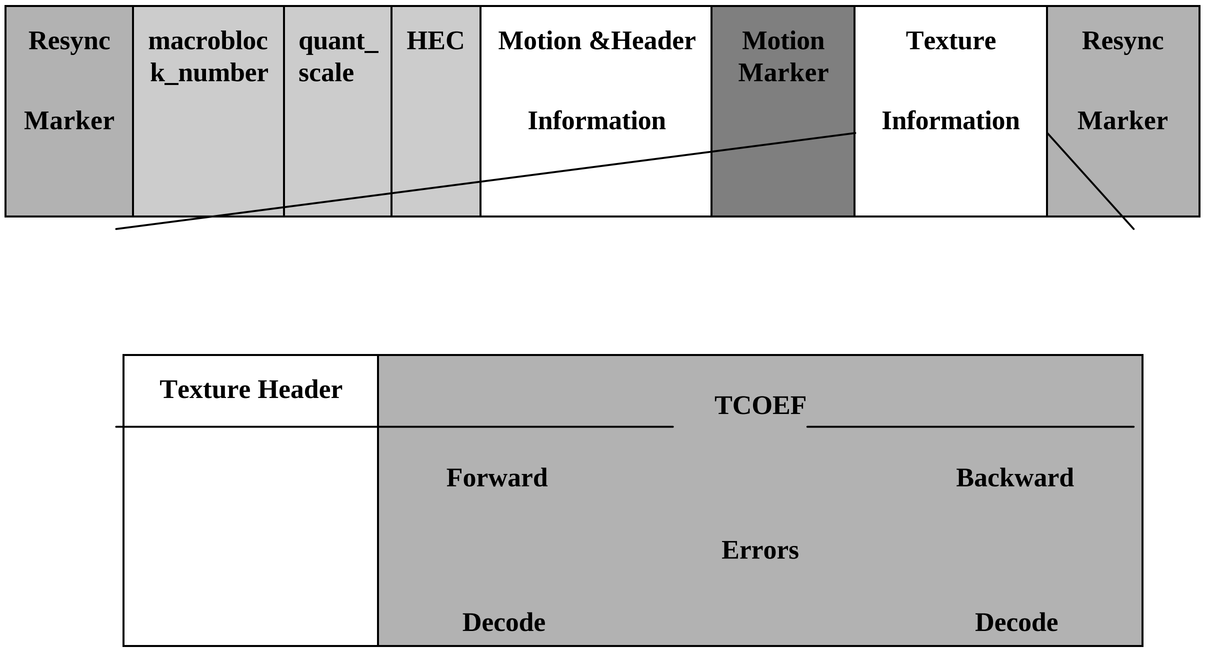


Figure 14-3 Reversible VLC

14.1.4 Decoder Operation

14.1.4.1 General Error Detection

1. An illegal VLC is received.

2. A semantic error is detected.

- More than 64 DCT coefficients are decoded in a block.

- Inconsistent resyncronization header information (i.e., QP out of range, MBA(k)<MBA(k-1),etc.)

14.1.4.2 Resynchronization

When an error is detected in the bitstream, the decoder should resynchronize at the next suitable resynchronization point(VOP_start_code or resync_marker).

After that, it can be determined by detecting the incorrect number of the stuffing bits whether or not errors are injured in a video packet. If eight or more 1's are followed after 0 at the last part of a video packet, or the remaining bit pattern is not "011…", it means there is any error in this video packet.

If the VOP start code is corrupted by the transmission error and the frame synchronization is lost, the decoder may establish the resynchronization by using the HEC information. The decoder compares the VOP_time_increment in the VOP header with one in the HEC field. If they are not same, the decoder may find that the current VOP start code is corrupted by the error. In this case, there must not be the error in the both VOP_time_increments. The simple method is to check whether the VOP_time_increment is mutilple of frame interval of the original source format (NTSC, PAL and so on). Therefore, it is expected that the number of the VOP_time_increment is as many as possible. As this check method does not always detect the error, this is the auxiliary technique.

Missing blocks may be replaced with the same block from the previous frame.

14.1.4.3 Data Partitioning

14.1.4.4 Reversible VLC

This section describes a decoding methodology for Reversible Variable Length Codes (RVLC) when errors in the video bitstream are detected during the decoding process. This particular decoding methodology was developed during the RVLC core experiment process.

14.1.4.4.1 Process for detecting errors for both forward and backward decoding

Errors are present in the following cases:

(1) An illegal RVLC is found, where an illegal RVLC is defined as follows:

- A codeword whose pattern is not listed in the RVLC table (e.g. 169 codeword patterns and escape codes).

- Escape coding is used (i.e., a legal codeword is not available in the RVLC table) and the decoded value for LEVEL is zero.

- The second escape code is incorrect (e.g. codeword is not “00000” or “00001” for forward decoding, and/or is not “00001” for backward decoding).

- There is a decoded value of FLC part using escape codes (e.g. LAST, RUN, LEVEL) in the RVLC table.

- An incorrect number of stuffing bits for byte alignment (e.g. eight or more 1s follow 0 at the last part of a Video packet (VP), or the remaining bit pattern is not “0111...” after decoding process is finished).

(2) More than 64 DCT coefficients are decoded in a block.

14.1.4.4.2 Decoding information

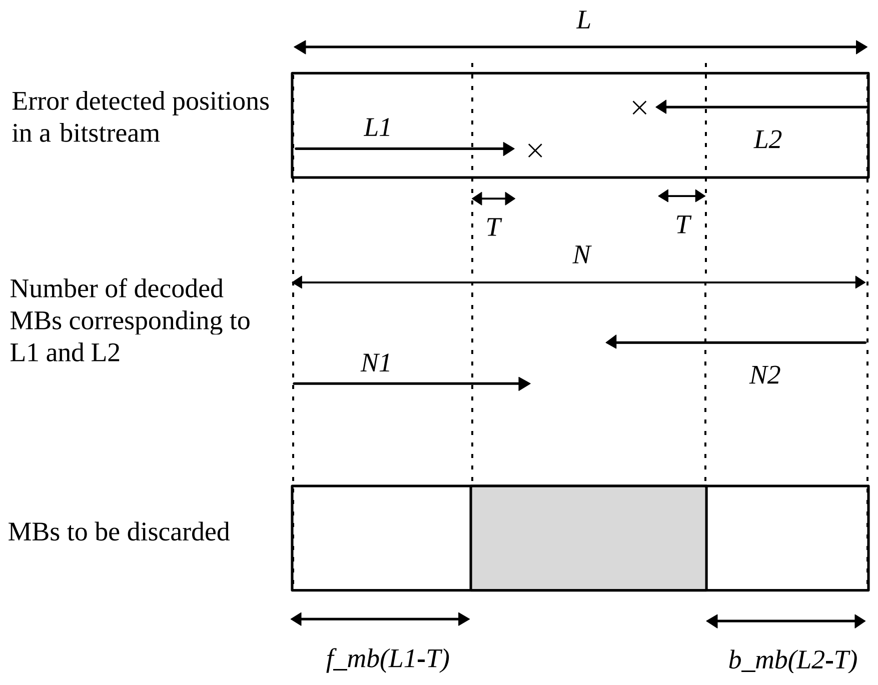
The bitstream is decoded in a forward direction first. If no errors are detected, the bitstream is assumed to be valid and the decoding process is finished for that video packet. If an error is detected however, two-way decoding is applied. The following strategies for determining which bits to discard are used. These strategies are described using the figures given below along with the following definitions:

L	: Total number of bits for DCT coefficients part in a VP.
N	: Total number of macroblocks (MBs) in a VP.
L1	: Number of bits which can be decoded in a forward decoding.
L2	: Number of bits which can be decoded in a backward decoding.
N1	: Number of MBs which can be completely decoded in a forward decoding.
N2	: Number of MBs which can be completely decoded in a backward decoding.
f_mb(S)	: Number of decoded MBs when S bits can be decoded in a forward direction. (Equal to or more than one bit can be decoded in a MB, f_mb(S) counter is up.)
b_mb(S)	: Number of decoded MBs when S bits can be decoded in a backward direction.
T	: Threshold (90 is used now).

14.1.4.4.2.1 Strategies for decoding RVLC

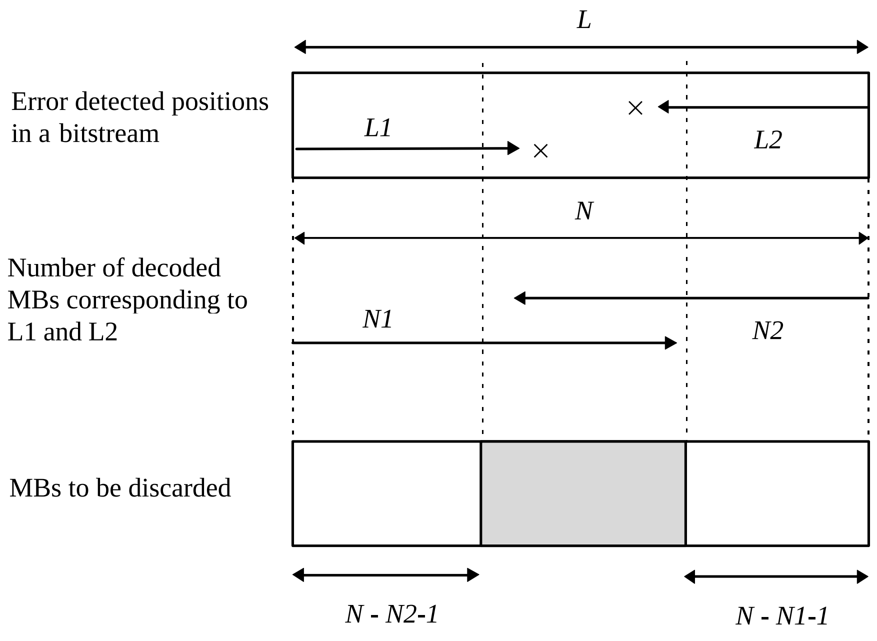
(1) Strategy 1 : *L1+L2 < L* and *N1+N2 < N*

MBs of *f_mb(L1-T)* from the beginning and MBs of *b_mb(L2-T)* from the end are used. In the figure below, the MBs of the dark part are discarded.


L
Error detected positions in a bitstream
L1
L2
T
T
N
Number of decoded MBs corresponding to L1 and L2
N1
N2
MBs to be discarded
f_mb(L1-T)
b_mb(L2-T)

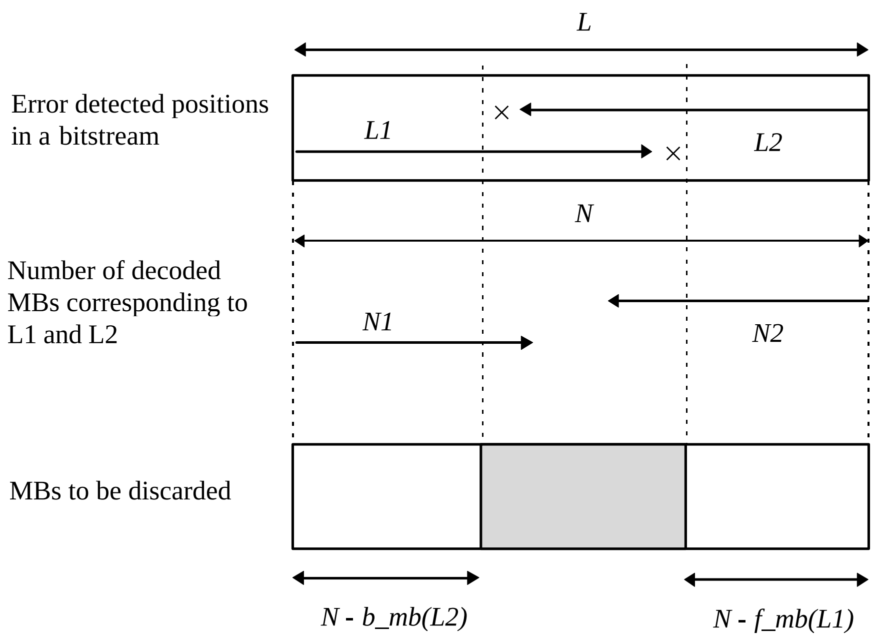
(2) Strategy 2 : *L1+L2 < L* and *N1+N2 >= N*

MBs of *N-N2-1* from the beginning and MBs of *N-N1-1* from the end are used. MBs of the dark part are discarded.


L
Error detected positions in a bitstream
L1
L2
N
Number of decoded MBs corresponding to L1 and L2
N1
N2
MBs to be discarded
N - N2-1
N - N1-1

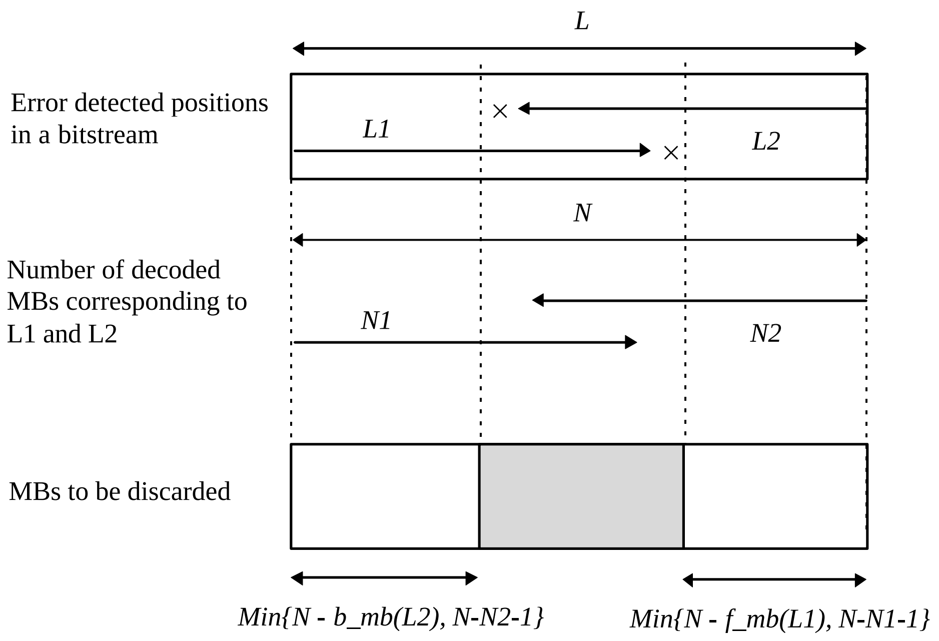
(3) Strategy 3 : *L1+L2 >= L* and *N1+N2 < N*

MBs of *N-b_mb(L2)* from the beginning and MBs of *N-f_mb(L1)* from the end are used. MBs of the dark part are discarded.


L
Error detected positions in a bitstream
L1
L2
N
Number of decoded MBs corresponding to L1 and L2
N1
N2
MBs to be discarded
N - b_mb(L2)
N - f_mb(L1)

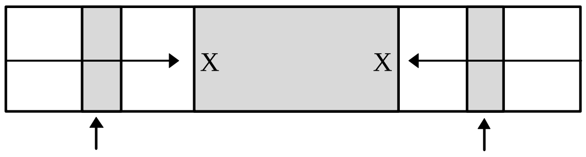
(4) Strategy 4 : *L1+L2 >= L* and *N1+N2 >= N*

MBs of *min{N-b_mb(L2), N-N2-1}* from the beginning and MBs of *min{N-f_mb(L1), N-N1-1}* from the end are used. MBs of the dark part are discarded.


L
Error detected positions in a bitstream
L1
L2
N
Number of decoded MBs corresponding to L1 and L2
N1
N2
MBs to be discarded
Min{N - b_mb(L2), N-N2-1}
Min{N - f_mb(L1), N-N1-1}

14.1.4.4.2.2 INTRA MBs within a bitstream

In the above strategies (**Strategy 1 - Strategy 4**), INTRA MBs are discarded even though they could have been decoded. An example of such a case is shown below.



Although these intra MBs are thought to be correct, the result of displaying an Intra MB that does contain an error can substantially degrade the quality of the video. Therefore, when a video packet is determined to contain errors, all Intra MBs are not displayed, but instead concealed.

14.1.5 Adaptive Intra Refresh (AIR) Method

The AIR is the technique of the intra refresh method for the error resilience. In the AIR, motion area is encoded frequently in Intra mode. Therefore, it is possible to recover the corrupted motion area quickly.

The method of the "AIR"

The number of Intra MBs in a VOP is fixed and pre-determined. It depends on bitrates and frame rate and so on.

The encoder estimates motion of each MB and the only motion area is encoded in Intra mode. The results of the estimation are recorded to the **Refresh Map** MB by MB. The encoder refers to the Refresh Map and decides to encode current MB in Intra mode or not. The estimation of motion is performed by the comparison between SAD and SAD_th. SAD is the Sum of the Absolute Differential value between the current MB and the MB in same location of the previous VOP. The SAD has been already calculated in the Motion Estimation part. Therefore, additional calculation for the AIR is not needed. SAD_th is the threshold value. If the SAD of the current MB is larger than the SAD_th, this MB is regarded as motion area. Once the MB is regarded as motion area, it is regarded as motion area until it is encoded in Intra mode predetermined times. The predetermined value is recorded to the Refresh Map. (*See figure 14-4. In this figure, predetermined value is "1" as an example*)

The holizontal scan is used to determine the MBs to be encoded in Intra mode within the moving area (*see figure 14-5*).

0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0	0	0
0	0	1	1	1	1	0	0	0	0	0
0	0	0	1	1	1	1	0	1	0	0
0	0	0	1	1	1	1	1	1	1	0
0	0	0	0	0	1	1	1	1	1	0
0	0	0	0	0	0	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0	0

Figure 14-4 Refresh Map for QCIF

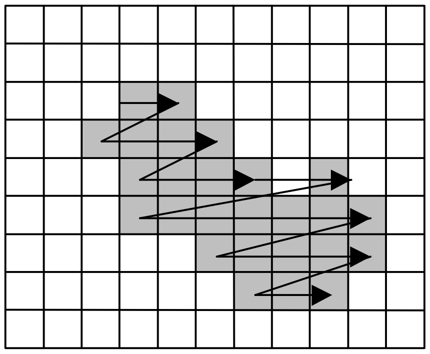


Figure 14-5 Scan order for the Adaptive Refresh

The processing of the "AIR"

The following is the explanation of the processing of AIR (*see figure 3*). The fixed number of the Intra MB in a VOP should be determined in advance. Here, it is set to "2" as an example.

[1] 1st VOP ([a]~[b] in figure 3)

The all MBs in the 1st VOP are encoded in Intra mode [a]. The Refresh Map is set to "0", because there is no previous VOP [b].

[2] 2nd VOP ([c] ~ [f])
The 2nd VOP is encoded as P-VOP. Intra refresh is not performed in this VOP, because all values in the Refresh Map is zero yet ([c] and [d]). The encoder estimates motion of each MB. If the SAD for current MB is larger than the SAD_th, it is regarded as motion area (hatched area in figure 3 [e]). And the Refresh Map is updated [f].

[3] 3rd VOP ([g] ~ [k])
When the 3rd VOP is encoded, the encoder refers to the Refresh Map [g]. If the current MB is the target of the Intra refresh, it is encoded in Intra mode [h]. The value of the MB in Refresh Map is decreased by 1 [i]. If the decreased value is 0, this MB is not regarded as motion area. After this, the processing is as same as the 2nd VOP [j]~[k].

[4] 4th VOP ([l]~[p])

It is as same as 3rd VOP...

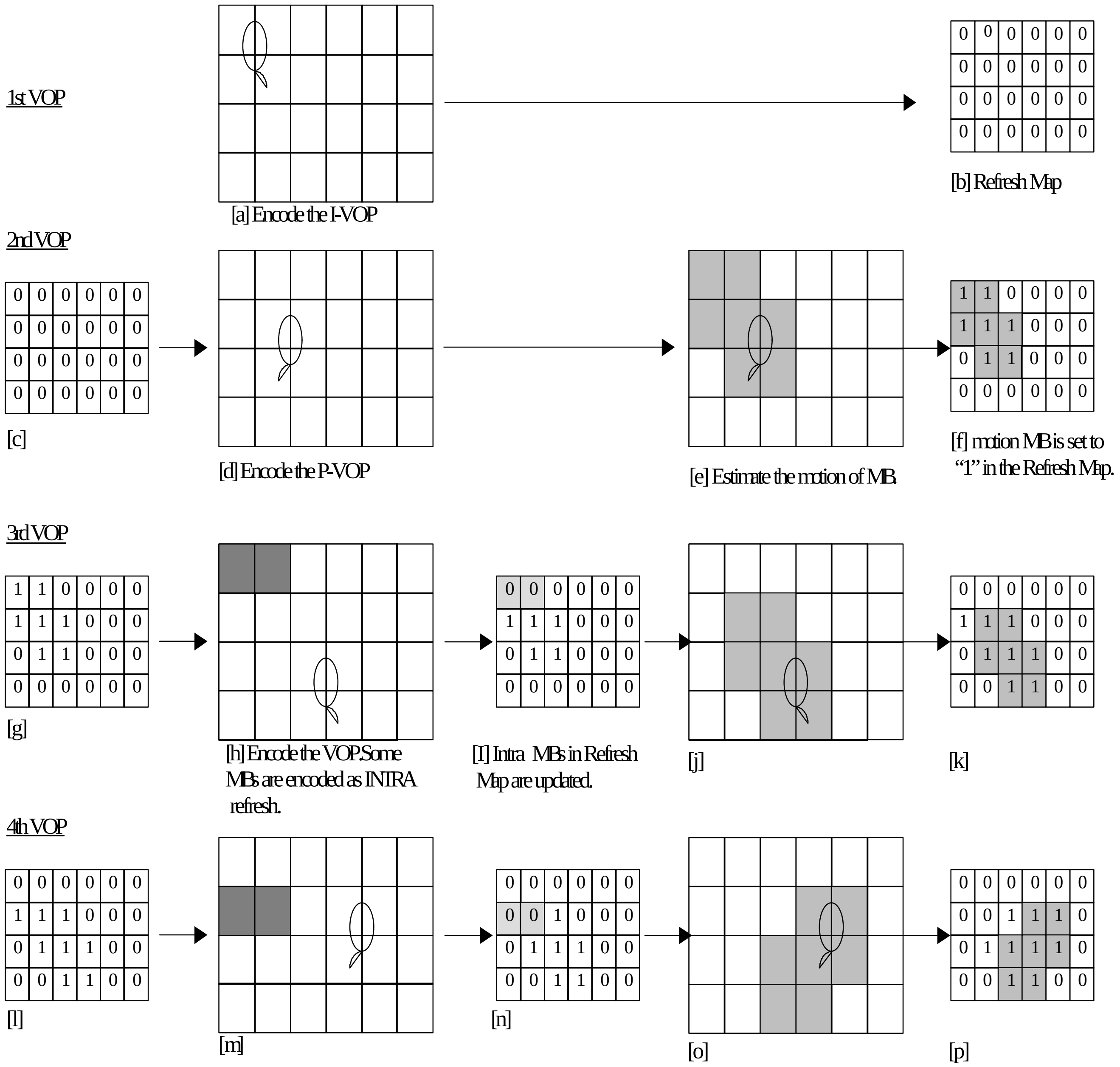

1st VOP
[a] Encode the I-VOP
[b] Refresh Map
2nd VOP
[c]
[d] Encode the P-VOP
[e] Estimate the motion of MB.
[f] motion MB is set to "1" in the Refresh Map.
3rd VOP
[g]
[h] Encode the VOP.Some MBs are encoded as INTRA refresh.
[I] Intra MBs in Refresh Map are updated.
[j]
[k]
4th VOP
[l]
[m]
[n]
[o]
[p]

Figure 14-6 the Explanation of AIR

14.2 Complexity Estimation

The Complexity Estimation Tool enables the estimation of decoding complexity without the need of the actual decoding of the incoming VOPs. The tool is based on the trasmission of the statistic of the actual encoding algorithms, modes and parameters used to encode the incoming VOP. The 'cost' in complexity for the execution of each algorithm is measured or eastimated on each decoder platform. The actual statistic of the decoding algorithms is transmitted in the video bitstream and can be converted by means of the mentioned 'costs' into the VOP actual decoding cost for the specific decoder.
The tool is flexible since it enables, for each VO, the definition of the set of used statistics. Such definition is done in the VO header. The actual values of the defined statistics is then inserted into each VOP header according to the 'complexity estimation syntax'.

For the implementation of the Complexity Estimation Tool, the following modifications to the video syntax, indicated by the gray background, are necessary :

14.2.1 Video Object Layer Class

14.2.1.1 Video Object Layer

Syntax	No. of bits	Mnemonic
VideoObjectLayer() {		
video_object_layer_start_code	sc+4=28	
video_object_layer_id	4	
video_object_layer_shape	2	
if (video_object_layer_shape == '00') {		
video_object_layer_width	13	
video_object_layer_height	13	
}		

…………………………..

…………………………..

…………………………..

…………………………..

if(load_gray_nonintra_quant_mat)		
Gray_nonintra_quant_mat[64]	8*64	
}		
}		
Complexity_estimation_disable	1	
if (!Complexity_estimation_disable){		
Parameter_list	8	
Estimation_method	2	
if (Estimation_method =='00'){		
Shape_complexity_estimation_disable	**1**	
if (Shape_complexity_estimation_disable) {		
Shape_Binary	**1**	
Shape_Gray	**1**	
}		
Texture_complexity_estimation_set_1_disable	**1**	
if (!Texture_complexity_estimation_set_1_disable) {		
Intra		
Inter	**1**	
Inter4v	**1**	
Not_Coded	**1**	
}		
Texture_complexity_estimation_set_2_disable	**1**	
if (!Texture_complexity_ estimation_set_2_disable) {		
DctCoef	**1**	
DctLine	**1**	
VlcSymbols	**1**	
VlcBits	**1**	
}		
Motion_compensation_complexity_disable	**1**	
if (!Motion_compensation_complexity_disable) {		
APM (Advanced Prediction Mode)	**1**	
NPM (Normal Prediction Mode)	**1**	

```
                **InterpolateMC+Q**                    **1**
                **Forw+Back+MC+Q**                     **1**
                **HalfPel2**                           **1**
                **HalfPel4**                           **1**
              **}**
          }
      }
      **Error_resilient_disable**                     1
      if (!error_resilient_disable) {
          **data_partitioning**                       1
          **Reversible_VLC**                          1
      }
```

……………………….

……………………….

………………………..

14.2.1.2 Parameter definition of Complexity Estimation Syntax

Complexity_estimation_disable: flag for disabling complexity estimation header in each VOP

Parameter_list: number of complexity estimation parameters.

Estimation_method: definition of the estimation method.

Shape_complexity_estimation_disable: flag to disable setting of shape parameters

Shape_Binary: flag enabling transmission of the number of luminance and chrominance blocks coded using binary alpha block shape coding information in % of the total number of blocks (bounding box).

Shape_Gray: flag enabling transmission of the number of luminance and chrominance blocks coded using gray scale shape coding information in % of the total number of blocks (bounding box).

Texture_complexity_estimation_set_1_disable: flag to disable parameter set 1.

Intra: flag enabling transmission of the number of luminance and chrominance Intra or Intra+Q coded blocks in % of the total number of blocks (bounding box).

Inter: flag enabling transmission of the number of luminance and chrominance Inter and Inter+Q coded blocks in % of the total number of blocks (bounding box).

Inter4v: flag enabling transmission of the number of luminance and chrominance Inter4V coded blocks in % of the total number of blocks (bounding box).

Not-Coded: flag enabling transmission of the number of luminance and chrominance Non Coded blocks in % of the total number of blocks (bounding box).

Texture_complexity_estimation_set_2_disable: flag to disable parameter set 2.

DctCoef: flag enabling transmission of the number of DCT coefficients % of the maximum number of coefficients (coded blocks).

DctLine: flag enabling transmission of the number of DCT8x1 in % of the maximum number of DCT8x1 (coded blocks).

VlcSymbols: flag enabling transmission of the average number of VLC symbols for macroblock.

VlcBits: flag enabling transmission of the average number of bits for each symbol.

Motion_compensation_complexity_disable: flag to disable motion compensation parameter set.

APM (Advanced Prediction Mode): flag enabling transmission of the number of luminance block predicted using APM in % of the total number of blocks for VOP (bounding box).

NPM (Normal Prediction Mode): flag enabling transmission of the number of luminance and chrominance blocks predicted using NPM in % of the total number of luminance and chrominance for VOP (bounding box).

InterpolateMC+Q: flag enabling transmission of the number of luminance and chrominance interpolated blocks in % of the total number of blocks for VOP (bounding box).

Forw+Back+MC+Q: flag enabling transmission of the number of luminance and chrominance predicted blocks in % of the total number of blocks for VOP (bounding box).

HalfPel2: flag enabling transmission of the number of luminance and chrominance block predicted by a half-pel vector on one dimension (horizontal or vertical) in % of the total number of blocks (bounding box).

HalfPel4: flag enabling transmission of the number of luminance and chrominance block predicted by a half-pel vector on two dimensions (horizontal and vertical) in % of the total number of blocks (bounding box).

14.2.2 Video Object Plane Class

14.2.3 Video Object Plane

Syntax	**No. of bits**	**Mnemonic**
VideoObjectPlane() {		
VOP_start_code	Sc+8=32	
do {		
modulo_time_base	1	
} while (modulo_time_base != "0")		
VOP_time_increment	10	
VOP_prediction_type	2	
if (video_object_layer_sprite_usage != SPRITE_NOT_USED)		

………………….

………………….

………………….

………………….

Interlaced	1	
if (interlaced)		
Top_field_first	1	
if (VOP_prediction_type=='10')		
VOP_dbquant	2	
else {		
VOP_quant	Quant_precision	
If(video_object_layer_shape == "10")		
VOP_gray_quant	6	
}		
if ((video_object_layer_shape_effects == '0010') ||		
(video_object_layer_shape_effects == '0011') ||		
(video_object_layer_shape_effects == '0101')) {		
VOP_constant_alpha	1	

```
        If (VOP_constant_alpha)
            VOP_constant_alpha_value                               8
    }
    if (!Complexity_estimation_disable){
        if (estimation_method==’00’){
            If (VOP_prediction_type==‘00’){
                If(Shape_Gray)    DCECS_Shape_Binary               8
                If (Shape_Binary) DCECS_Shape_Gray                 8
                    If (Intra)          DCECS_Intra                8
                    If (Not_Coded)      DCECS_Not_Coded            8
                    If (DctCoef)        DCECS_DctCoef              8
                    If (DctLine)        DCECS_DctLine              8
                    If (VlcSymbols)     DCECS_VlcSymbols           8
                    If (VlcBits)        DCECS_VlcBits              4
            }
            If (VOP_prediction_type==‘01’){
                If(Shape_Gray)    DCECS_Shape_Binary               8
                If (Shape_Binary) DCECS_Shape_Gray                 8
                    If (Intra)          DCECS_Intra                8
                    If (Not_Coded)      DCECS_Not_Coded            8
                    If (DctCoef)        DCECS_DctCoef              8
                    If (DctLine)        DCECS_DctLine              8
                    If (VlcSymbols)     DCECS_VlcSymbols           8
                    If (VlcBits)        DCECS_VlcBits              4
                    If (Inter)          DCECS_Inter                8
                    If (Inter4v)        DCECS_Inter4v              8
                    If(APM)             DCECS_APM                  8
                    If(NPM)             DCECS_NPM                  8
                    If(Forw+Back+MC+Q)  DCECS_Forw+Back+MC+Q       8
                    If(HalfPel2)        DCECS_HalfPel2             8
                    If(HalfPel4)        DCECS_HalfPel4             8
            }
            If (VOP_prediction_type==‘10’){
                If(Shape_Gray)    DCECS_Shape_Binary               8
                If (Shape_Binary) DCECS_Shape_Gray                 8
                    If (Intra)          DCECS_Intra                8
                    If (Not_Coded)      DCECS_Not_Coded            8
                    If (DctCoef)        DCECS_DctCoef              8
                    If (DctLine)        DCECS_DctLine              8
                    If (VlcSymbols)     DCECS_VlcSymbols           8
                    If (VlcBits)        DCECS_VlcBits              4
                    If (Inter)          DCECS_Inter                8
                    If (Inter4v)        DCECS_Inter4v              8
                    If(APM)             DCECS_APM                  8
                    If(NPM)             DCECS_NPM                  8
                    If(Forw+Back+MC+Q) DCECS_ Forw+Back+MC+Q         8
                    If(HalfPel2)       DCECS_ HalfPel2               8
                    If(HalfPel4)       DCECS_ HalfPel4               8
                    If(InterpolateMC+Q)  DCECS_InterpolateMC+Q       8
                }
            If (VOP_prediction_type=='11'){
                    If (Intra)         DCECS_Intra                   8
                    If (Not_Coded)     DCECS_Not_Coded               8
                    If (DctCoef)       DCECS_DctCoef                 8
                    If (DctLine)       DCECS_DctLine                 8
                    If (VlcSymbols)    DCECS_VlcSymbols              8
                    If (VlcBits)       DCECS_VlcBits                 4
                    If (Inter)         DCECS_Inter                   8
                    If (Inter4v)       DCECS_Inter4v                 8
                    If(APM)            DCECS_APM                     8
                    If(NPM)            DCECS_NPM                     8
                    If(Forw+Back+Q)    DCECS_ Forw+Back+Q            8
                    If(HalfPel2)       DCECS_ HalfPel2               8
                    If(HalfPel4)       DCECS_ HalfPel4               8
                    If(InterpolateMC+Q)  DCECS_InterpolateMC+Q       8
            }
        }
    }

    if (!scalability) {
        if (!separate_motion_shape_texture)
```

Definition of DCECS Parameter Values

The semantic of all DCECS parameters is defined at the VO syntax level. They represents % values. The actual values of all 8 bit words are normalized to 256 plus the addition of a binary 1 to prevent start code emulation (i.e 0% = '00000001', 99.5% = '11111111' and 100% = '11111111'). The binary '00000000' string is a forbidden value. The only parameter expressed in absolute value is the DCEDS_VlcBits parameter expressed in absolute value in a 4 bit word.

14.2.4 Resynchronization in Case of Unknown Video Header Format

Two video object layer starting indicators are supported:

1. video_object_layer_start_code, and
2. short_video_start_marker

The automatic detection of which of the these byte aligned codes is present is unambiguous. The short_video_start_marker will never emulate a video_object_layer_start_code, since 23 byte-aligned zeros cannot occur in any video stream using the short_video_start_marker. The video_object_layer_start_code will also never emulate a short_video_start_marker, because its first non-zero bit is in a different location (provided byte alignment is not lost).

However, special attention needs to be paid if some application requires starting at any arbitrary point in a bitstream for which there is no prior knowledge of the format type. Although unlikely, a resync_marker can emulate a short_video_start_marker (for certain macroblock_number field lengths and macroblock_number values and vop_fcode_forward values).

Although the behavior of the decoder in these circumstances is not specified, it is suggested to perform validity checks on the first few bits beyond the short_video_start_marker if the video_object_layer_start_code is not the first starting indicator found. Numerous checks are possible, for example, checking the values of the bits 9, 10, 18-21 and 27 beyond the short_video_start_marker. The decoder may also choose to delay acquisition until an "I" vop-type is indicated in bit 17. Even simply discarding some data while searching for a video_object_layer_start_code prior to "timing out" with a decision to seek the short_video_start_marker may be acceptable for some applications.

15. Annex F
Preprocessing and Postprocessing

(This annex does not form an integral part of the International Standard)

15.1 Segmentation for VOP Generation

15.1.1 Introduction

The video coding scheme defined by this standard offers several content-based functionalities, demanding the description of the scene in terms of so-called video-objects (VOs). The separate coding of the video objects may enrich the user interaction in several multimedia services due to flexible access to the bit-stream and an easy manipulation of the video information. In this framework, the coder may perform a locally defined pre-processing, aimed at the automatic identification of the objects appearing in the sequence. Hence, segmentation is a key issue in efficiently applying the MPEG-4 coding scheme, although not affecting at all the bit-stream syntax and thus not being a normative part of the standard.

Usually, the term segmentation denotes the operation aimed at partitioning an image or a video sequence into regions extracted according to a given criterion. In the case of video sequences, this partition should achieve the temporal coherence of the resulting sequence of object masks representing the video object. In the recent literature, different methods have been proposed for segmentation of video sequences, based on either a spatial homogeneity, a motion coherence criterion [[8]] or on joint processing of spatio-temporal information [4][[9]][[11]][[15]]. These algorithms are expected to identify classes of moving objects, according to some luminance homogeneity and motion coherence criterion.

In this annex, a framework aiming at an appropriate combination of temporal and spatial segmentation strategies, developed throughout the standardisation phase of MPEG-4 Version 1, is described. The description is given only for informative purposes as the technique to extract objects from the scene is not standardised. The classification of the pels in a video sequence is performed into two classes, namely moving objects (foreground) and background. This framework will continue to be investigated throughout the standardisation phase of MPEG-4 Version 2, leading to improved segmentation results. Only the general principles are shortly described, however, if more detail is required a number of references containing much more detailed descriptions are given.

15.1.2 Description of a combined temporal and spatial segmentation framework

Throughout the work on automatic segmentation of moving objects, different proposals for temporal and spatial segmentation algorithms have been proposed and investigated. This resulted at the end in a combined temporal and spatial segmentation framework [[6]] which is shown in a high level block diagram in Figure 15-1.

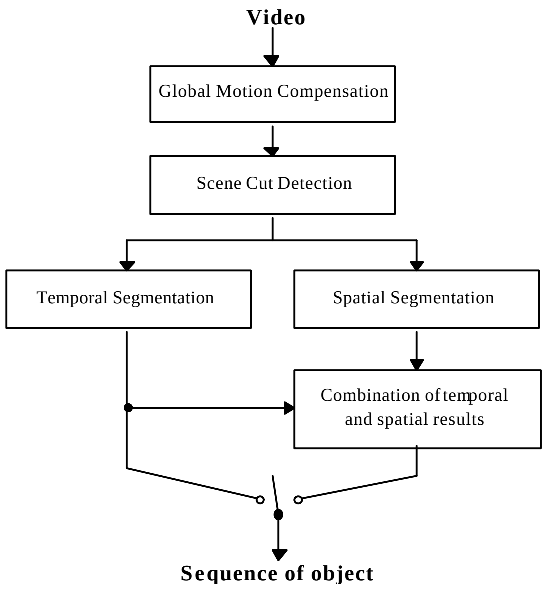


Figure 15-1 Block diagram of combined temporal and spatial segmentation framework

The combined scheme applies in a first step the general blocks of camera motion estimation and compensation [[17]][[18]] and scene cut detection [[13]] which can be seen as a kind of pre-processing in order to eliminate the influence of a moving camera.

In a second step, either temporal or combined spatio-temporal segmentation of each image are carried out, depending on the requirements. The reason for this is, that in general only performing temporal segmentation requires less computational complexity. On the other hand, taking into account also spatial segmentation leads to more accurate segmentation results, but increases the computational complexity of the segmentation.

For temporal segmentation, two possible algorithms are under consideration, both having been verified by extensive cross-checking. It will be one main task for the group which will be working on segmentation for MPEG-4 Version 2, to decide which of these algorithms performs better. For spatial segmentation, only one algorithm is considered, which however has not been cross-checked by the group.

Finally, if temporal and spatial segmentation is performed, both temporal and spatial segmentation results are combined. It will be the second main task of the group to work out an appropriate algorithm for combining the temporal and spatial segmentation results.

The three algorithms for temporal and spatial segmentation will be shortly described in the following. For more details on them as well as on the possible combination approaches [[15]][[10]][[7]], the reader is referred to the given references, where more detailed descriptions can be found.

Temporal segmentation based on change detection*:* this segmentation algorithm [[16]][[17]][[18]], which is mainly based on a change detection, can be subdivided into two main steps, assuming that a possible camera motion has already been compensated: by the first step, a change detection mask between two successive frames is estimated. In this mask, pels for which the image luminance has changed due to a moving object are labelled as changed. For that, first an initial change detection mask between the two successive frames is generated by global thresholding the frame difference. After that, boundaries of changed image areas are smoothed by a relaxation technique using local adaptive thresholds [[1]][[2]]. Thereby, the algorithm adapts frame-wise automatically to camera noise. In order to finally get temporal stable object regions, an object mask memory with scene adaptive memory length is applied. Finally, the mask is simplified and small regions are eliminated, resulting in the final change detection mask.

In the second step, an object mask is calculated by eliminating the uncovered background areas from the change detection mask as in [[12]]. Therefore, displacement information for pels within the changed regions is used. The displacement is estimated by a hierarchical blockmatcher (HBM) [[3]]. For a higher accuracy of the calculated displacement vector field (DVF), the change detection mask from the first step is considered by the HBM. Pels are set to foreground in the object mask, if foot- and top-point of the corresponding displacement vector are both inside the changed area in the current CDM. If not, these pels are set to background. Results for the described method can be found in [[14]][[16]][[18]].

Temporal segmentation using higher order moments and motion tracking*:* The algorithm [8][[9]][[19]][[20]] produces the segmentation map of each frame f_k of the sequence by processing a group of frames $\{f_{k-i},\ i=0,..n\}$. The number of frames n varies on the basis of the estimated object speed [8]. For each frame f_k, the algorithm splits in three steps. First, the differences $\{\ d_{k-j}\ (x,y)=f_{k-j}(x,y)-f_{k-n}(x,y),\ j=0,..n-1\ \}$ of each frame of the group with respect to the first frame f_{k-n} are evaluated in order to *detect the changed areas*, due to object motion, uncovered background and noise. In order to reject the luminance variations due to noise, an Higher Order Statistic test is performed. Namely, for each pixel (x,y) the fourth-order moment $\hat{m}_d^{(4)}(x,y)$ of each inter-frame difference $d(x,y)$ is estimated on a 3x3 window, it is compared with a threshold adaptively set on the basis of the estimated background activity [8], and set to zero if it is below the threshold. Then, on the sequence of the thresholded fourth-order moment maps $\tilde{m}_{d_{k-j}}^{(4)}(x,y)$, a *motion detection* procedure is performed. This step aims at distinguish changed areas representing uncovered background (which stands still in the HOS maps) and moving objects (moving in the HOS maps). At the *j*-th iteration, the pair of thresholded HOS maps $\tilde{m}_{d_{k-j}}^{(4)}(x,y), \tilde{m}_{d_{k-j-1}}^{(4)}(x,y)$ is examined. For each pixel (x,y) the displacement of is evaluated on a 3x3 window, adopting a SAD criterion, and if the displacement is not null the pixel is classified as moving. Then, the lag j is increased (i.e. the pair of maps slides) and the motion analysis is repeated, until $j=n-2$. Pixels presenting null displacements on all the observed pairs are classified as still. Finally, a *regularization* algorithm re-assigns still regions, internal to moving regions, to foreground and refines the segmentation results imposing a priori topological constraints on the size of objects irregularities such as holes, isthmi, gulfs and isles by morphological filtering. A post-processing operation refines the results on the basis of spatial edges.

Spatial segmentation based on watershed algorithm: In the spatial segmentation, images are first simplified to make easier the image segmentation [[21]]. Morphological filters are used for the purpose of image simplification. These filters remove regions that are smaller than a given size but preserve the contours of the remaining objects. By the second step, the spatial gradient of the simplified image is approximated by the use of a morphological gradient operator [[21]]. The spatial gradient can be used as an input of watershed algorithm to partition an image into homogeneous intensity regions. For the problem of ambiguous boundaries by spatial gradient, we incorporate color information into gradient computation in which the largest values among the weighed gradients obtained in aY, bC_r, gC_b are chosen [[5]]. In the boundary decision step, the boundary decision is taken through the use of a watershed algorithm that assigns pixels in the uncertainty area to the most similar region with some segmentation criterion such as difference of intensity values [[22]]. To merge into semantic regions the genetically over-segmented regions from watershedding, a region merging algorithm is then incorporated [[5]]. The final output of the spatial segmentation is the images that are composed of semantically meaningful regions with precise boundaries. Moving objects are therefore represented with semantic regions with precise boundaries and can be segmented in conjunction with temporal information that localizes the moving objects.

15.1.3 References

[1] T. Aach, A. Kaup, R. Mester, „Statistical model-based change detection in moving video“, *Signal Processing,* Vol. 31, No. 2, pp. 165-180, March 1993.

[2] T. Aach, A. Kaup, R. Mester, „Change detection in image sequences using Gibbs random fields: a Bayesian approach“, *Proceedings Int. Workshop on Intelligent Signal Processing and Communication Systems,* Sendai, Japan, pp. 56-61, October 1993.

[3] M. Bierling, „Displacement estimation by hierarchical blockmatching“, *3rd SPIE Symposium on Visual Communications and Image Processing*, Cambridge, USA, pp. 942-951, November 1988.

[4] P. Bouthemy, E. François, „Motion segmentation and qualitative dynamic scene analysis from an image sequence” in Int. Journal of Computer Vision vol.10, no.2, pp157-182, 1993.

[5] J. G. Choi, M. Kim, M. H. Lee, C. Ahn, **„**Automatic segmentation based on spatio-temporal information“, Doc. ISO/IEC JTC1/SC29/WG11 MPEG97/2091, April 1997.

[6] J. G. Choi, M. Kim, M. H. Lee, C. Ahn (ETRI); S. Colonnese, U. Mascia, G. Russo, P. Talone (FUB); Roland Mech, Michael Wollborn (UH), **„**Merging of temporal and spatial segmentation“, Doc. ISO/IEC JTC1/SC29/WG11 MPEG97/2383, July 1997.

[7] J. G. Choi, M. Kim, M. H. Lee, C. Ahn, „New ETRI results on core experiment N2 on automatic segmentation techniques“, Doc. ISO/IEC JTC1/SC29/WG11 MPEG97/2641, October 1997.

[8] S. Colonnese, A. Neri, G. Russo, C. Tabacco, „Adaptive Segmentation of Moving Object versus Background for Video Coding”, Proceedings of SPIE Annual Symposium, Vol. 3164, San Diego, August 1997.

[9] S. Colonnese, U. Mascia, G. Russo, P. Talone, „Core Experiment N2: Preliminary FUB results on combination of automatic segmentation techniques“, Doc. ISO/IEC JTC1/SC29/WG11 MPEG97/2365, July 1997.

[10] S. Colonnese, U. Mascia, G. Russo, „Automatic segmentation techniques: updated FUB results on core experiment N2“, Doc. ISO/IEC JTC1/SC29/WG11 MPEG97/2664, October 1997.

[11] J. Dugelay, H. Sanson, „Differential methods for the identification of 2D and 3D motion models in image sequences“, *Signal Processing*, Vol.7, pp. 105-127, Sept. 1995.

[12] M. Hötter, R. Thoma, „Image Segmentation based on object oriented mapping parameter estimation“, *Signal Processing*, Vol. 15, No. 3, pp. 315-334, October 1988.

[13] M. Kim, J. G. Choi, M. H. Lee, C. Ahn; **„**Performance analysis of an ETRI’s global motion compensation and scene cut detection algorithms for automatic segmentation“, Doc. ISO/IEC JTC1/SC29/WG11 MPEG97/2387, July 1997.

[14] R. Mech, P. Gerken, „Automatic segmentation of moving objects (Partial results of core experiment N2), Doc. ISO/IEC JTC1/SC29/WG11 MPEG97/1949, April 1997.

[15] R. Mech, M. Wollborn, „Automatic segmentation of moving objects (Partial results of core experiment N2)“, Doc. ISO/IEC JTC1/SC29/WG11 MPEG97/2703, October 1997.

[16] R. Mech, M. Wollborn, „A Noise Robust Method for Segmentation of Moving Objects in Video Sequences“, *International Conference on Acoustic, Speech and Signal*, Munich, Germany, April 1997.

[17] R. Mech, M. Wollborn, „A Noise Robust Method for 2D Shape Estimation of Moving Objects in Video Sequences Considering a Moving Camera“, *Workshop on Image Analysis for Multimedia Interactive Services*, Louvain-la-Neuve, Belgium, June 1997.

[18] R. Mech, M. Wollborn, „A Noise Robust Method for 2D Shape Estimation of Moving Objects in Video Sequences Considering a Moving Camera“, accepted for publication in *Signal Processing: Special Issue on Video Sequence Segmentation for Content-based Processing and Manipulation*, to be published in the beginning of 1998.

[19] A. Neri, S. Colonnese, G. Russo, „Video Sequence Segmentation for Object-based Coders using Higher Order Statistics“, ISCAS ‘97, Hongkong, June 1997.

[20] A. Neri, S. Colonnese, G. Russo, „Automatic Moving Objects and Background Segmentation by means of Higher Order Statistics“, IS&T Electronic Imaging ‘97 Conference: Visual Communication and Image Processing, San Jose’, 8-14 February 1997, SPIE Vol. 3024.

[21] P. Salembier, M. Pardàs, „Hierarchical Morphological Segmentation for Image Sequence Coding”, IEEE Transactions on Image Processing, Vol.3, No.5, pp. 639-651, September 1994.

[22] Luc Vincent, Pierre Soille, „Watersheds in digital spaces: an efficient algorithm based on immersion simulations“, *IEEE Transactions on PAMI*, Vol.13, No. 6, pp. 583-598, June 1991.

15.2 Bounding Rectangle of VOP Formation

This section describes the bounding rectangle of VOP formation process. The formation of the bounding rectangle of VOP is based on the segmented shape information. The following explains the process to achieve the bounding rectanle of a VOP in such a way that the minimum number of macroblocks containing the object will be attained in order to achieve a higher coding efficiency.

1. Generate the tightest rectangle whose top left poition is an even number.

2. If the top left position of this rectangle is not the same as the origin of the image, the following steps have to be performed. Otherwise no further processing is necessary.

3. Form a control macroblock at the top left corner of the tightest rectangle as shown in Figure 15-2.

4. Count the number of macroblocks that completely contain the VOP for all even numbered point of the control macroblock using the following procedure.
   - Generate a bounding rectangle from the control point to the right bottom side of the VOP which consists of multiples of 16x16 blocks.
   - Count the number of macroblocks in this bounding rectangle, which contain at least one object pel. To do so, it would suffice to take into account the boundary pels of a macroblock only.

5. Select the control point that results in the smallest number of non transparent macroblocks for the given object.

6. Extend the top left coordinate of the tightest rectangle generated in Figure 15-2. to the selected control coordinate. This will create a rectangle that completely contains the object but with the minimum number of non transparent macroblocks in it. The VOP horizontal and vertical spatial references are taken directly from the modified top-left coordinate.

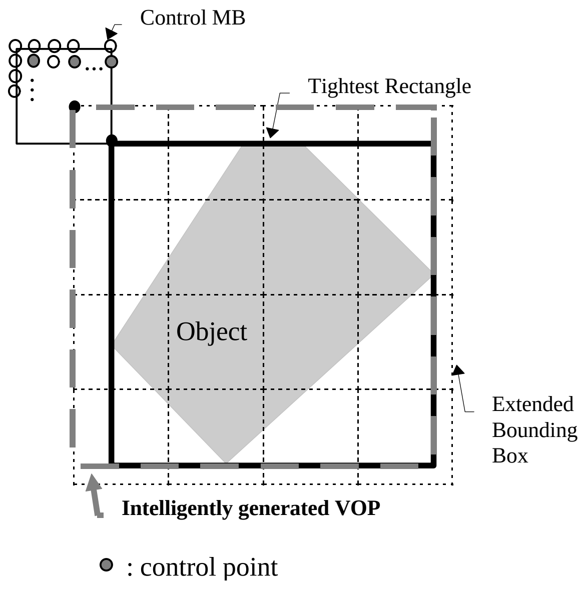


Figure 15-2 Intelligent VOP Formation

15.3 Postprocessing for Coding Noise Reduction

The post-filter consists of deblocking filter and deringing filter. Either one or both of them can be turned on as needed.

15.3.1 Deblocking filter

The filter operations are performed along the 8x8 block edges at the decoder as a post-processing operation. Luminance as well as chrominace data is filtered. Figure 15-3 shows the block boundaries.

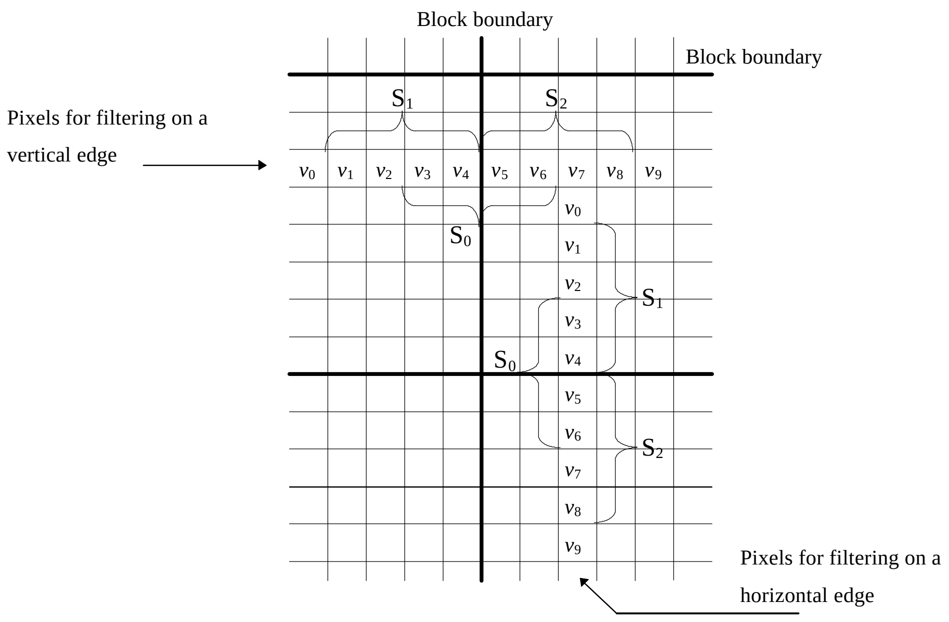

Block boundary
Block boundary
Pixels for filtering on a vertical edge
S1
S2
v0 v1 v2 v3 v4 v5 v6 v7 v8 v9
S0
v0 v1 v2 v3 v4 v5 v6 v7 v8 v9
S1
S0
S2
Pixels for filtering on a horizontal edge

Figure 15-3 Boundary area around block of interest

In the filter operations, two modes are used separately depending on the pixel conditions around a boundary. The following procedure is used to find a very smooth region with blocking artifacts due to small dc offset and to assign it a DC offset mode. In the other case, default mode operations are applied.

```
eq_cnt = $\phi(v0–v1) + \phi(v1–v2) + \phi(v2–v3) + \phi(v3–v4) + \phi(v4–v5) + \phi(v5–v6) + \phi(v6–v7)$
              $+ \phi(v7–v8) + \phi(v8–v9)$,
where     $\phi(\gamma) = 1$  if $|\gamma| \leq$ THR1 and 0 otherwise.
If (eq_cnt $\geq$ THR2)
          DC offset mode is applied,
else
          Default mode is applied.
```

For the simulation, threshold values of THR1 = 2 and THR2 = 6 are used.

In the default mode, a signal adaptive smoothing scheme is applied by differentiating image details at the block discontinuities using the frequency information of neighbor pixel arrays, S_0, S_1, and S_2,. The filtering scheme in default mode is executed by replacing the boundary pixel values v_4 and v_5 with v_4' and v_5' as follows:

$v_4' = v_4 - d$,

$v_5' = v_5 + d$,

and $d = \mathrm{CLIP}(5\cdot(a_{3,0}' - a_{3,0})//8, 0, (v_4 - v_5)/2) \cdot \delta(|a_{3,0}| < \mathrm{QP})$

where $a_{3,0}' = \mathrm{SIGN}(a_{3,0}) \cdot \mathrm{MIN}(|a_{3,0}|, |a_{3,1}|, |a_{3,2}|)$.

Frequency components $a_{3,0}$, $a_{3,1}$, and $a_{3,2}$ can be evaluated from the simple inner product of the approximated DCT kernel [2 -5 5 -2] with the pixel vectors, i.e.,

a3,0 = ([2 -5 5 -2] • [v3 v4 v5 v6]T) // 8,
a3,1 = ([2 -5 5 -2] • [v1 v2 v3 v4]T) // 8,
a3,2 = ([2 -5 5 -2] • [v5 v6 v7 v8]T) // 8.

Here CLIP(x,p,q) clips x to a value between p and q; and QP denotes the quantization parameter of the macroblock where pixel v_5 belongs. δ(*condition*)=1 if the "*condition*" is true and 0 otherwise..

In very smooth region, the filtering in the default mode is not good enough to reduce the blocking artifact due to dc offset. So we treat this case in the DC offset mode and apply a stronger smoothing filter as follows :

```
max = MAX (v1, v2, v3, v4, v5, v6, v7, v8),
    min = MIN (v1, v2, v3, v4, v5, v6, v7, v8),
if ( |max–min| < 2·QP ) {
    $v_n' = \sum_{k=-4}^{4} b_k \cdot p_{n+k}, 1 \le n \le 8$
    $p_m = \begin{cases} (|v_1 - v_0| < QP) ? v_0 : v_1, if \quad m < 1 \\ v_m, \quad if 1 \le m \le 8 \\ (|v_8 - v_9| < QP) ? v_9 : v_8, if \quad m > 8 \end{cases}$
    $\{b_k : -4 \le k \le 4\} = \{1,1,2,2,4,2,2,1,1\} // 16$
}
else
    No change will be done.
```

The above filter operations are applied for all the block boundaries first along the horizontal edges followed by the vertical edges. If a pixel value is changed by the previous filtering operation, the updated pixel value is used for the next filtering.

15.3.2 Deringing filter

This filter comprises three subprocesses; threshold determination, index acquisition and adaptive smoothing. This filter is applied to the pixels on 8x8 block basis. More specifically 8x8 pixels are processed by referencing 10x10 pixels at each block. The following notation is used to specify the six blocks in a macroblock. For instance, block[5] corresponds to the *Cb* block whereas block[*k*] is used as a general representation in the following sections.

15.3.2.1 Threshold determination

Firstly, calculate maximum and minimum gray value within a block in the decoded image. Secondary, the threshold denoted by *thr[k]* and the dynamic range of gray scale denoted by *range[k]* are set:

$$thr[k] = (\max imum[k] + \min imum[k] + 1) / 2$$
$$range[k] = \max imum[k] - \min imum[k]$$

An additional process is done only for the luminance blocks. Let *max_range* be the maximum value of the dynamic range among four luminance blocks.

$$\max\_range = range[k_{\max}]$$

Then apply the rearrangement as follows.

```
for( k=1 ; k<5 ; k++ ){
    if( range[k] < 32 && max_range > =64 )
        thr[k] = thr[kmax];
    if( max_range<16 )
        thr[k] = 0;
}
```

15.3.2.2 Index acquisition

Once the threshold value is determined, the remaining operations are purely 8x8 block basis. Let *rec(h,v)* and *bin(h,v)* be the gray value at coordinates *(h,v)* where *h,v=0,1,2,...,7,* and the corresponding binary index, respectively. Then *bin(h,v)* can be obtained by:

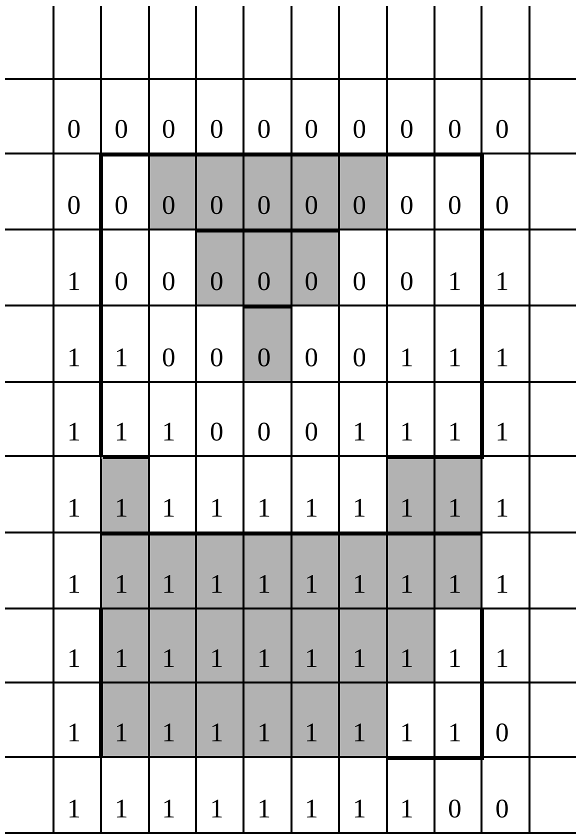
$$bin(h,v) = \begin{cases} 1 & if\ rec(h,v) \geq thr \\ 0 & otherwise \end{cases}$$

Note that *(h,v)* is use to address a pixel in a block, while *(i,j)* is for accessing a pixel in a 3x3 window.

15.3.2.3 Adaptive smoothing

15.3.2.3.1 Adaptive filtering

The figure below is the binary indices in 8x8 block level, whereas practically 10x10 binary indices are calculated to process one 8x8 block.



0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1	1
1	1	1	0	0	0	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	1	0	0

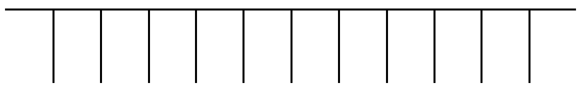


Figure 15-4: Example of adaptive filtering and binary index

The filter is applied only if the binary indices in a 3x3 window are all the same, i.e., all “0” indices or all “1” indices. Note 10x10 binary indices are obtained with a single threshold which corresponds to the 8x8 block shown in the above figure, where the shaded region represents the pixels to be filtered.

The filter coefficients used for both intra and non-intra blocks denoted by *coef(i,j)*, where *i,j=-1,0,1*, are:

1	2	1
2	4	2
1	2	1

Figure 15-5: Filter mask for adaptive smoothing

Here the coefficient at the center pixel, i.e., *coef(0,0)*, corresponds to the pixel to be filtered. The filter output *flt’(i,j)* is obtained by:

$$flt'(h,v) = \left\{8 + \sum_{i=-1}^{1}\sum_{j=-1}^{1} coef(i,j)\cdot rec(h+i,v+j)\ \right\}//16$$

15.3.2.3.2 Clipping

The maximum gray level change between the reconstructed pixel and the filtered one is limited according to the quantization parameter, i.e., QP. Let *flt(h,v)* and *flt’(h,v)* be the filtered pixel value and the pixel value before limitation, respectively.

```
if( flt’(h,v) - rec(h,v) > max_diff )
      flt(h,v) = rec(h,v) + max_diff
else if( flt’(h,v) - rec(h,v) < -max_diff )
      flt(h,v) = rec(h,v) - max_diff
else
      flt(h,v) = flt’(h,v)
where max_diff=QP/2 for both intra and inetr macroblocks.
```

15.3.3 Further issues

In order to reduce the number of computations in post-filtering, two kinds of semaphores can be defined: the blocking semaphores and the ringing semaphore. Depending on the blocking semaphores, the horizontal and the vertical deblocking filtering is applied strongly and weakly on the block boundary. If the ringing semaphore (RS) of a current block is “1”, deringing filtering is applied. For extracting the semaphores in intra-frame, when only a DC component in the 8x8 inverse quantized coefficients (IQC), the DCT coefficients after inverse quantization, has a non-zero value, both the horizontal blocking semaphore (HBS) and the vertical blocking semaphore (VBS) of the block are set to “1”. When only the coefficients in the top row of the IQC have non-zero values, the VBS is set to “1”. When only the coefficients in the far left column of the IQC have non-zero values, the HBS is set to “1”. The RS is set to “1” if any non-zero coefficient exists in positions other than a DC component, the first horizontal AC component, and the first vertical AC comonent. Also the semaphores of the inter-frame are calculated from both the residual signal and the semaphores of the reference frame by using the motion vector.

[1] M2723 Y.L. Lee et al.

16. Annex G
Profile and level restrictions

(This annex does not form an integral part of the International Standard)

This annex specifies the syntax element restrictions and permissible layer combinations.

17. Annex H
Visual Bitstream Syntax in MSDL-S

(This annex forms an integral part of the International Standard)

*{**Note:** This annex needs to be updated to be conformant to the bitstream specification of this part.}*

18. Annex I

Patent statements

(This annex does not form an integral part of this International Standard)

19. Annex J
Bibliography

(This annex does not form an integral part of this International Standard)

1 Arun N. Netravali and Barry G. Haskell “Digital Pictures, representation and compression” Plenum Press, 1988

2 See the Normative Reference for Recommendation ITU-R BT.601

3 See the Normative Reference for IEC Standard Publication 461

4 See the Normative Reference for Recommendation ITU-T H.263

5 See the Normative reference for IEEE Standard Specification P1180-1990

6 ISO/IEC 10918-1 | ITU-T T.81 (JPEG)

7 Barry G. Haskell, Atul Puri, Arun N. Netravali, "Digital Video: An Introduction to MPEG-2," Chapman & Hall, ISBN 0-412-08411-2, 1997.

8 A. Puri, R. L. Schmidt and B. G. Haskell, "Improvements in DCT Based Video Coding," Proc. SPIE Visual Communications and Image Processing, San Jose, Feb. 1997.

9 A. Puri, R. L. Schmidt and B. G. Haskell, "Performance Evaluation of the MPEG-4 Visual Coding Standard," to appear in SPIE Visual Communications and Image Processing, San Jose, Jan 1998.

10 F. I. Parke, K. Waters, Computer Facial Animation, A K Peters, Wellesley, MA, USA, 1996

20. Annex K
View Dependent Object Scalability

(This annex does form an integral part of this International Standard)

20.1 Introduction

Coding of View-Dependent Scalability (VDS) parameters for texture can provide for efficient incremental decoding of 3D images (e.g. 2D texture mapped onto a gridded 3D mesh such as terrain). Corresponding tools from ISO/IEC 14496-1 and 14496-2 of this specification are used in conjunction with downstream and upstream channels of a decoding terminal. The combined capabilities provide the means for an encoder to react to a stream of viewpoint information received from a terminal. The encoder transmits a series of coded textures optimized for the viewing conditions which can be applied in the rendering of textured 3D meshes by the receiving terminal. Each encoded view-dependent texture (initial texture and incremental updates) typically corresponds to a specific 3D view in the user's viewpoint that is first transmitted from the receiving terminal.

A tool defined in ISO/IEC 14496-1 transmits 3D viewpoint parameters in the upstream channel back to the encoder. The encoder's response is a frequency-selective, view-dependent update of DCT coefficients for the 2D texture (based upon view-dependent projection of the 2D texture in 3D) back to the receiving terminal, along the downstream channel, for decoding by a Visual DCT tool at the receiving terminal. This bilateral communication supports interactive server-based refinement of texture for low-bandwidth transmissions to a decoding terminal that renders the texture in 3D for a user controlling the viewpoint movement. A gain in texture transmission efficiency is traded for longer closed-loop latency in the rendering of the textures in 3D. The terminal coordinates inbound texture updates with local 3D renderings, accounting for network delays so that texture cached in the terminal matches each rendered 3D view.

A method to obtain an optimal coding of 3D data is to take into account the viewing position in order to transmit only the most visible information. This approach reduces greatly the transmission delay, in comparison to transmitting all scene texture that might be viewable in 3D from the encoding database server to the decoder. At a given time, only the most important information is sent, depending on object geometry and viewpoint displacement. This technique allows the data to be streamed across a network, given that a upstream channel is available for sending the new viewing conditions to the remote database. This principle is applied to the texture data to be mapped on a 3D grid mesh. The mesh is first downloaded into the memory of the decoder using the appropriate BIFS node, and then the DCT coefficients of the texture image are updated by taking into account the viewing parameters, i.e. the field of view, the distance and the direction to the viewpoint.

20.2 Decoding Process of a View-Dependent Object

This subclause explains the process for decoding the texture data using the VDS parameters. In order to determine which of the DCT coefficients are to be updated, a "mask", which is a simple binary image, shall be computed. The first step is to determine the viewing parameters obtained from the texture-mesh composition procedure that drives 3D rendering in the user's decoding terminal. These parameters are used to construct the DCT mask corresponding to the first viewpoint of the session (VD mask). This mask is then updated with differential masks, built with the new viewing parameters that allow the texture image to be streamed. The bitstream syntax for view parameters and incremental transmission of DCT coefficients is given elsewhere in the ISO/IEC 14496-1 and 14496-2 of this standard.

20.2.1 General Decoding Scheme

The following subclauses outline the overall process for the decoder and encoder to accomplish the VDS functionalities.

20.2.1.1 View-dependent parameters computation

The VDS parameters (α and β angles, distance d for each cell) shall be computed using the geometrical parameters (Mesh, Viewpoint, Aimpoint, Rendering window). These parameters shall be computed for each cell of the grid mesh.

20.2.1.2 VD mask computation

For each 8x8 block of texture elements within a 3D mesh cell, the locations of the visible DCT coefficients inside the DCT block shall be computed using α and β angles, and the distance d defined for each cell relative to the viewpoint. The result shall be put in a binary mask image.

20.2.1.3 Differential mask computation

With the knowledge of which DCT coefficients have already been received (Binary mask buffered image) and which DCT coefficients are necessary for the current viewing conditions (Binary VD mask image), the new DCT coefficients shall be determined (Binary Differential mask image) as described in subclause 20.2.4 of this specification.

20.2.1.4 DCT coefficients decoding

The Video Intra bitstream, in the downstream channel, shall be decoded by the receiver terminal to obtain the DCT coefficients (DCT image). The decoding procedure is described in subclause 20.2.5 of this specification.

20.2.1.5 Texture update

The current DCT buffer in the receiver terminal shall be updated according to the Differential mask, using the received DCT image. The new received DCT coefficients shall be added to the buffered DCT image.

20.2.1.6 IDCT

The Inverse DCT of the updated DCT image shall computed, as specified in subclause 20.2.7 of this specification, to obtain the final texture.

20.2.1.7 Rendering

The texture is mapped onto the 3D mesh and the rendering of the scene is done, taking into account the mesh and the viewing conditions. This part of the procedure is outside the scope of this specification.

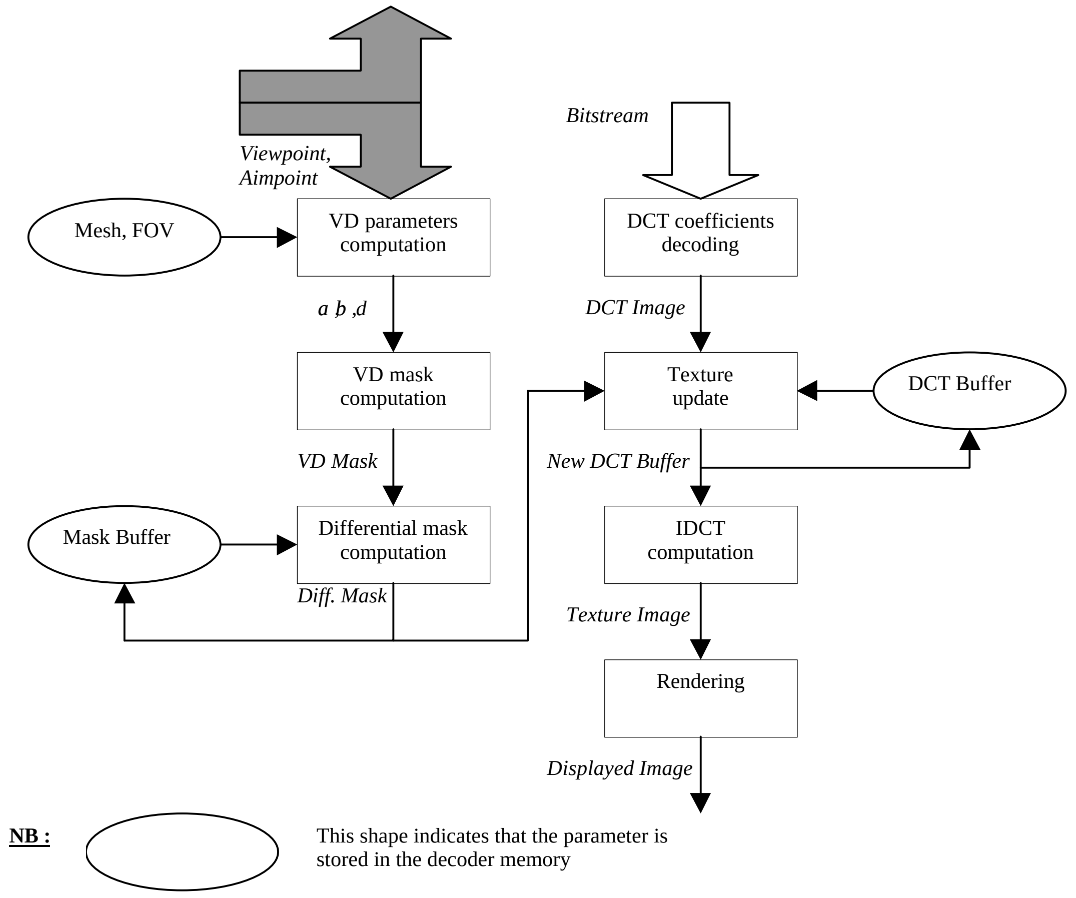
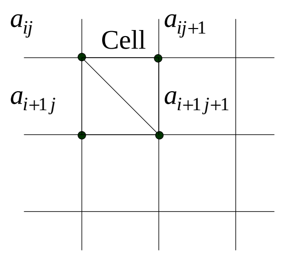


Figure 20-1: General Decoding Scheme of a View-Dependent Object

20.2.2 Computation of the View-Dependent Scalability parameters

The VDS parameters shall be computed for each cell of the grid mesh. The mesh may either be a quadrilateral or a triangular mesh. The number of cells in each dimension shall be equal to the texture size divided by 8.



20.2.2.1 Distance criterion:

$$Rd = \frac{1}{u}$$

u is the distance between viewpoint and Cell center: $u = \|\vec{v} - \vec{c}\|$ with $\vec{c} = \frac{1}{4}(\vec{a}_{ij} + \vec{a}_{i+1j} + \vec{a}_{ij+1} + \vec{a}_{i+1j+1})$ and $\vec{v}$ is the viewpoint vector.

20.2.2.2 Rendering criterion:

$$R_r = \frac{p}{q}$$

p is the distance between viewpoint and projection plane normalized to window width. *p* may be computed using:

$$p = \frac{1}{2\tan(FOV/2)}$$

$$p = \frac{1}{2\tan(FOV/2)}, \text{FOV is the Field Of View}$$

where FOV is the Field of View specified in radians, and q = <TextureWidth>/<WindowWidth> where the texture width is the width of the full texture (1024 for instance) and the WindowWidth is the width of the rendering window.

20.2.2.3 Orientation criteria:

$$Ra = \cos(a)$$
$$Rb = \cos(b)$$

The angle between the aiming direction and the normal of the current cell center shall be projected into two planes. These two planes are spans of normal vector $\vec{n}$ of the cell and the cell edges in x and y directions, respectively. Then the angles (α, β) between projected vectors and the normal $\vec{n}$ shall be calculated, respectively.

The angle *b* is specified as the projection of the angle between $\vec{n}$, the normal of the quad cell, and $\vec{u}$, the aiming direction, onto the plane Π_x that passes through $\vec{g}_x$ and is parallel to $\vec{n}$. Similarly, the angle *a* is specified as the projection of the same angle onto the plane Π_y that passes through $\vec{g}_y$ and its parallel to $\vec{n}$.

This is illustrated in Figure 20-2

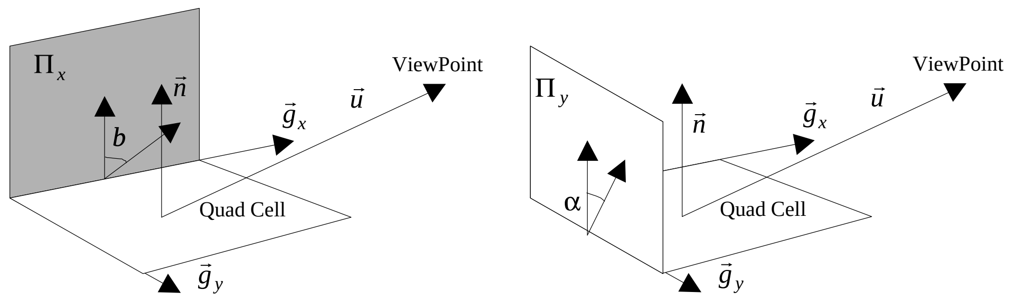


Figure 20-2: Definition of α and β angles

20.2.2.4 Cropping criterion:

Cells that are out of the field of view shall not be transmitted/received: that is, at least one of the 4 vertices which define the cell should all be inside the horizontal and vertical Field Of View (FOV).

The horizontal FOV shall be deduced from the vertical FOV using the screen geometry. The vertical FOV is equal to the FOV. Then the following shall be calculated

$HFOV = A\tan(\frac{w}{h}\cdot\tan(FOV/2))$ where *w* and *h* are the width and height, respectively, of the rendered image.

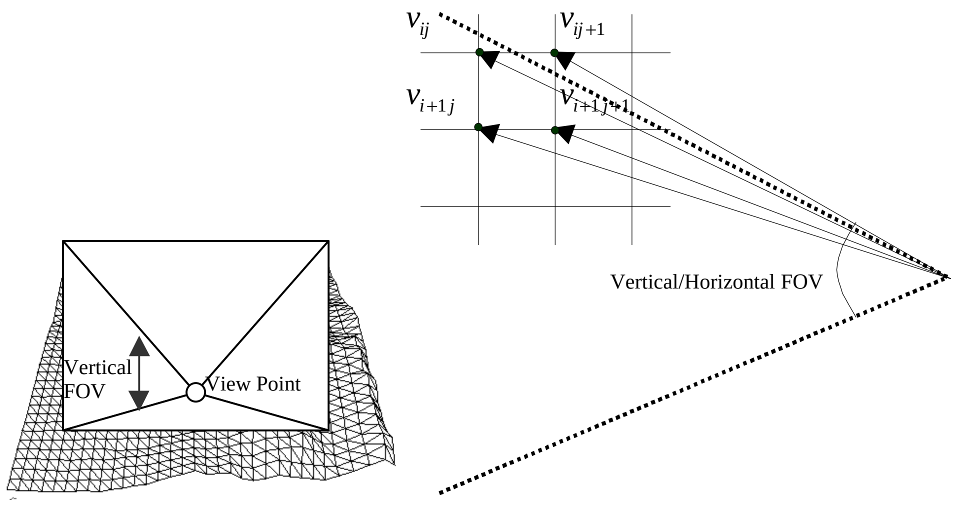


Figure 20-3: Definition of Out of Field of View cells

20.2.3 VD mask computation

The VD mask is a binary image of the same size as the texture image. Each value in the mask shall indicate if the corresponding DCT coefficient is needed (1) or not (0), given the VDS parameters.

For each cell, the following rules shall be applied to fill the corresponding 8x8 block of the VD mask:

- **Use of cropping criterion:** If all the vertices of the cell are out of the field of view, the corresponding 8x8 block of the mask image shall be set to 0.

- **Use of rendering, distance, tilting and rotation criteria:** For each 8x8 block of the mask (corresponding to a quad cell), the 4 criteria mentioned above shall be computed. Two values of the rotation and tilting criteria shall be obtained for a quad cell, but only the higher value of each criterion shall be kept.

Two thresholds, T_X and T_Y, shall be calculated as the product of the three VDS parameters *Rr, Rd, Rb*, and *Rr, Rd, Ra*, respectively, and the value 8. The results shall be bounded to 8. This procedure may be indicated symbolically as follows

$$Tx = Min(8, 8 \cdot R_r \cdot R_d \cdot R_b)$$
$$Ty = Min(8, 8 \cdot R_r \cdot R_d \cdot R_a)$$

The flag (i,j) of the 8x8 block corresponding to the current cell shall be set to 1 if $i < T_X$ and $j < T_Y$. The flag shall be set to 0 in all other cases, as illustrated in the figure below.

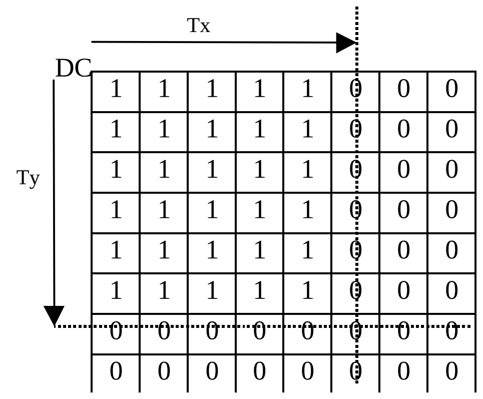


Figure 20-4: VD mask of an 8x8 block using VD parameters

20.2.4 Differential mask computation

Once the first image has been received using the previously described filter, less data is necessary to update the texture data for the following frames. (assuming there is a correlation between viewpoint positions). Since this computation is exactly the same for each flag of each cell, it shall be performed directly on the full mask images and not on a cell by cell basis.

If the coefficient has not been already transmitted (buffer mask set to 0) and is needed according to VDS visibility criteria (VD mask set to 1), then the corresponding pixel of the differential mask shall be set to 1. This implies that the texture shall be updated, according to the procedure described in the subclause 20.2.6 of this specification.

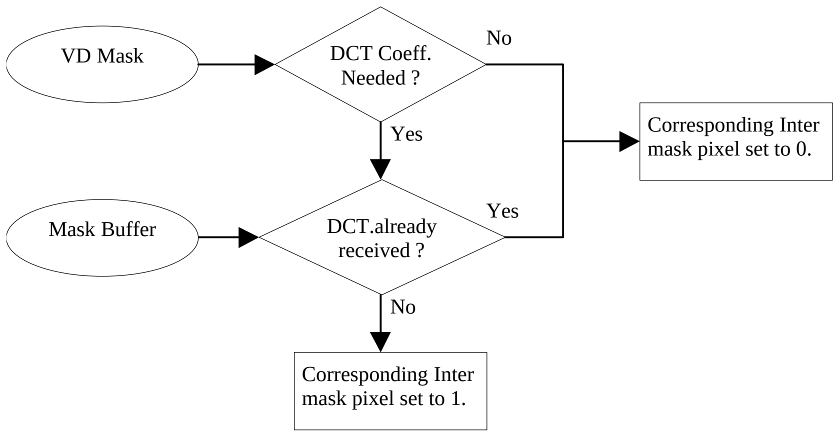

VD Mask
DCT Coeff. Needed ?
No
Yes
Corresponding Inter mask pixel set to 0.
Mask Buffer
DCT.already received ?
Yes
No
Corresponding Inter mask pixel set to 1.

Figure 20-5: Differential mask computation scheme

20.2.5 DCT coefficients decoding

The DCT coefficients shall be decoded using the Video Intra mode, Separated Texture/Motion mode as described in the subclause 7.3 of the ISO/IEC 14496-2.

20.2.6 Texture update

The Differential mask image shall be used to select which DCT coefficients of the buffered texture should be updated using the decoded DCT coefficients.

- Y component

If the Differential mask is set to 0, the corresponding DCT value of the buffer shall be left unchanged, otherwise the value shall be updated with the previously decoded DCT coefficient.

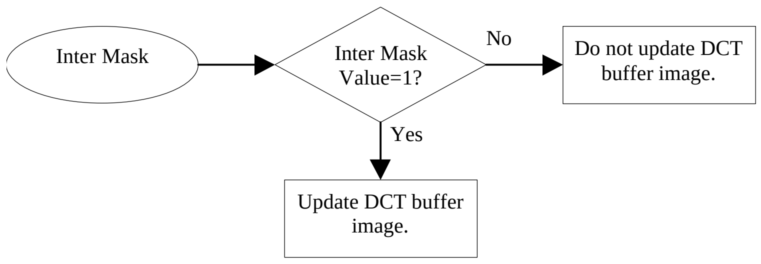


Figure 20-6: Texture update scheme

- U and V component

The texture is coded in 4:2:0 format, as specified in subclause 6.1.1.6 of the ISO/IEC 14496-2, which shall imply that for each chrominance DCT coefficient, 4 Differential mask flags shall be available. The chrominance coefficients shall be received/transmitted if at least 1 of these 4 flags is set to 1.

20.2.7 IDCT

The IDCT and de-quantization shall be performed using the same process as in the Video Intra mode as described in the clause 7.3 of ISO/IEC 14496-2.

21. Annex L

Decoder Configuration Information

(This annex does form an integral part of this International Standard)

21.1 Introduction

This annex identifies the syntax elements defined in the main body of ISO/IEC 14496-2 that describe the configuration of the visual decoder. These elements shall be processed differently if this part of the specification is used jointly with the Systems part, ISO/IEC 14496-1. Instead of conveying the configuration at the beginning of a visual elementary bitstream, it shall be conveyed as part of an Elementary Stream Descriptor that is itself included in an Object Descriptor describing the visual object. This descriptor framework is specified in ISO/IEC 14496-1.

21.2 Description of the set up of a visual decoder (informative)

The process of accessing ISO/IEC 14496 content is specified in ISO/IEC 14496-1. It is summarized here and visualized in Fig. L.1 to outline the processing of decoder configuration information in an ISO/IEC 14496 terminal. This description assumes that all elementary streams are accessible and solely the problem of identifying them is to be solved.

The content access procedure starts from an Initial Object Descriptor that may be made available through means that are not defined in this specification, like a http or ftp link on an HTML page, a descriptor in an ISO/IEC 13818-1 Transport Stream, or some H.245 signaling, etc. Note that this may need standardisation by the responsible group.

This Initial Object Descriptor contains pointers at least to a scene description stream and an object descriptor stream. The scene description stream conveys the time variant spatiotemporal layout of the scene. For each streaming media object incorporated in the scene, an Object Descriptor exists that describes the set of streams associated to this media object. The set of Object Descriptors is conveyed in a separate stream, in order to distinguish scene description from the description of the streaming resources.

Both the scene description stream and the object descriptor stream allow for time stamped updates of the scene description and the object descriptors, respectively. Due to the time stamps it is always known at which point in time, or from which point in time onwards a data item, called Access Unit, is valid.

The Object Descriptor associated to a given visual object is identified by its ObjectDescriptor_ID. Each visual object may require more than one elementary stream (ES) that convey its coded representation, especially if any form of scaleability is used. Each of these streams is described by an ES_Descriptor. This description contains a unique label for the stream, the ES_Id, and, among others, the DecoderSpecificInfo structure that is of concern for the purpose of this annex.

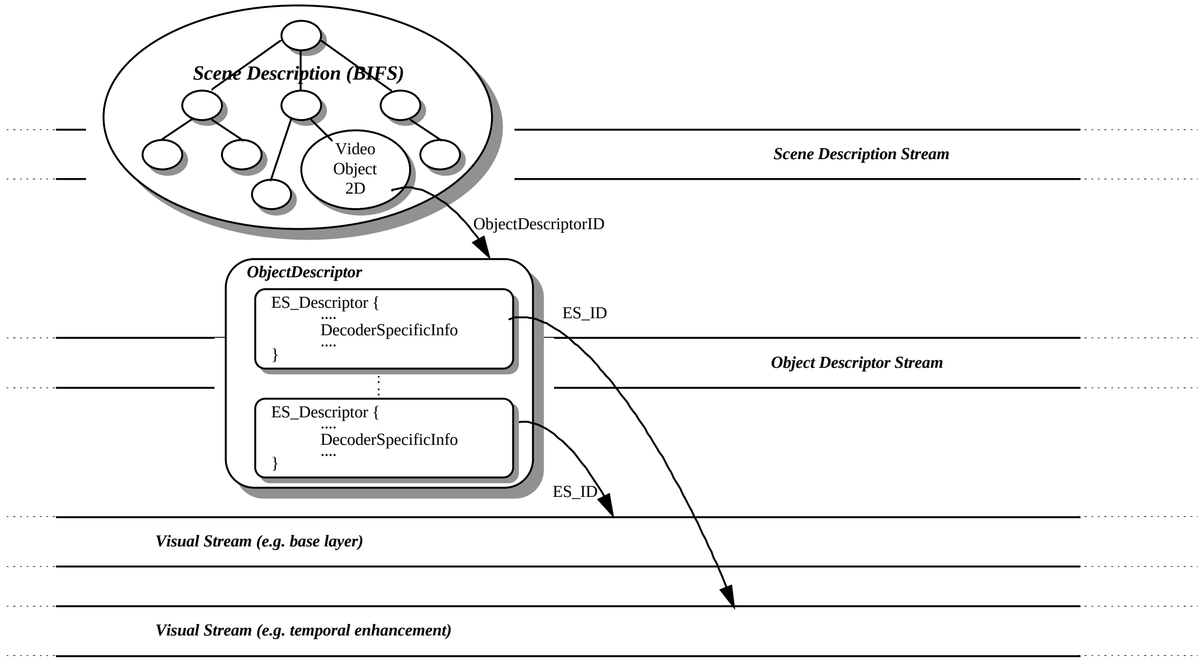


Figure 21-1: Visual decoder setup

21.2.1 Processing of decoder configuration information

After the retrieval of the Object Descriptor for a media object a decoder for the visual stream(s) is instantiated, connected to the stream(s) and initialised with the data found in ES_Descriptor.DecoderSpecificInfo.specificInfo[] for each stream. Subsequently, in a random access scenario, the decoder is expected to search forward to the next random access point, while in a client-server or local storage scenario, data in the visual stream may already be aligned in a way that the first data arriving in the visual stream corresponds to a random access point.

The difference between a visual-only scenario, as specified in the main body of ISO/IEC 14496-2, and an integrated application using both Systems and Visual parts of this specification is visualized in a figure. Figure 21-2 shows the Visual-only approach with configuration information at the beginning of a bit stream and optionally repeated thereafter to enable random access. Figure 21-3 with the integrated Systems and Visual approach shows the plain bit streams with the configuration information extracted into the object descriptors. In this case the object descriptors will be repeated if random access is desired. The Access Units shown in the figure correspond to VOPs in the case of visual media streams.

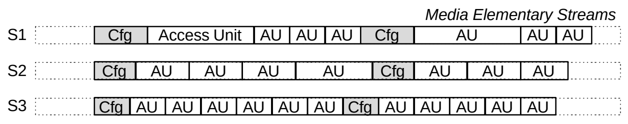


Figure 21-2: Visual-only scenario

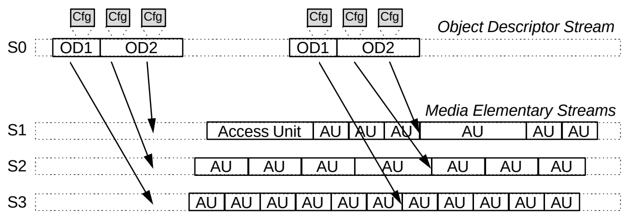


Figure 21-3: Integrated Systems and Visual approach

21.3 Specification of decoder configuration information

The decoder configuration information for a visual elementary stream is given by a concatenation of those syntax elements that precede the actual encoded data, i. e., that form the ‚stream header' according to the syntax specification in clause 6.2. Those syntax elements are identified separately for each type of visual object in the subsequent subclauses. The syntax elements that are conveyed as decoder configuration information shall not be present in the visual elementary stream itself. Furthermore, the generic syntax definition in clause 6.2 is constrained to clarify that an elementary stream may not contain a concatenation of, for example, multiple VisualObject() structures or multipe VideoObjectLayer() structures.

The decoder configuration information shall be conveyed in the DecoderSpecificInfo.specificInfo[] field of the respective ES_Descriptor and passed to the decoder before any data of the visual elementary stream itself.

21.3.1 VideoObject

The decoder configuration information for a visual object of type VideoObject consists of the following elements

- All syntax elements of VisualObjectSequence()
  - including all syntax elements of one VisualObject()
    - including all syntax elements of one VideoObject()
      - including all syntax elements of one VideoObjectLayer() excluding the trailing Group_of_VideoObjectPlane() and VideoObjectPlane() that convey the coded data.

VisualObject.profile_and_level_indication is overridden by the value present in the ES_Descriptor for this stream.

VisualObject.is_visual_object_identifier shall be zero. Note: Identification and priority of objects is signaled generically for all kinds of audiovisual objects in the ES_Descriptor.

VideoObjectLayer.is_object_layer_identifier shall be zero. Note: Identification and priority of objects is signaled generically for all kinds of audiovisual objects in the ES_Descriptor.

21.3.2 StillTextureObject

The decoder configuration information for a visual object of type StillTextureObject consists of the following elements

- All syntax elements of VisualObjectSequence()
  - including all syntax elements of one VisualObject()
    - including all syntax elements of one StillTextureObject() up to and including wavelet_decomposition_levels

VisualObject.profile_and_level_indication is overridden by the value present in the ES_Descriptor for this stream.

VisualObject.is_visual_object_identifier shall be zero. Note: Identification and priority of objects is signaled generically for all kinds of audiovisual objects in the ES_Descriptor.

21.3.3 MeshObject

The decoder configuration information for a visual object of type MeshObject consists of the following elements

- All syntax elements of VisualObjectSequence()
  - including all syntax elements of one VisualObject()
    - including all syntax elements of one MeshObject() excluding the trailing MeshObjectPlane() that convey the coded data.

VisualObject.profile_and_level_indication is overridden by the value present in the ES_Descriptor for this stream.

VisualObject.is_visual_object_identifier shall be zero. Note: Identification and priority of objects is signaled generically for all kinds of audiovisual objects in the ES_Descriptor.

21.3.4 FaceObject

The decoder configuration information for a visual object of type FaceObject consists of the following elements

- All syntax elements of VisualObjectSequence()
  - including all syntax elements of one VisualObject()
    - including all syntax elements of one FaceObject() up to and including face_object_coding_type

VisualObject.profile_and_level_indication is overridden by the value present in the ES_Descriptor for this stream.

VisualObject.is_visual_object_identifier shall be zero. Note: Identification and priority of objects is signaled generically for all kinds of audiovisual objects in the ES_Descriptor.

22. Annex M
Visual Combination Profiles@Levels

(This annex does form an integral part of this International Standard)

The table that describes the visual combination profiles and levels is given below, with the following notes:

1. Enhancement layers are not counted as separate objects.

2. Defined as the combined Object surface in active macroblocks (including memory needed for enhancement layers and B-VOPs). The Macroblocks can be overlapping. The numbers were derived as follows:
   - 99 MBs can fill a QCIF surface; 198 MBs is twice that amount.
   - 396 MBs can fill a CIF surface; 792 MBs is twice that amount; 495 MBs is 1.25 times 396.
   - 1620 MBs can fill a ITU-R BT.601 surface twice.

3. The conformance point for the Simple Scalable CP levels is the Simple CP @L1 when spatial scalability is used and Simple CP @ L2 when temporal scalability is used.

4. The numbers of MB's per second were derived as follows:
   - 1485 MBs/s corresponds to QCIF at 15Hz.
   - 5940 MBs/s corresponds to 15 Hz CIF; 11880 MB/s corresponds to 30 Hz CIF.
   - 7425 MB/s corresponds to 1.25 times the amount of MBs in CIF at 15 Hz.
   - 4860 MB/s corresponds to 1620 MBs at 30 Hz.

5. The maximum Number of Boundary Macroblocks is 50% of the maximum total number of macroblocks

Visual Combination Profile	Level	Typical Visual Session Size (in-dicative)	Max. total number of objects [1]	Max. number per type	Max. number different Quant Tables	Max. total Ref. memory (MB units)[2]	Max. number of MB/sec [4]	Max. number of Boundary MB/sec.[5]	Max sprite size (MB units)	Wavelet restric-tions	Max bitrate	Max. enhancement layers per object
12-Bit	L2	CIF	8	8 x Core or Simple	4	792	23760	11880 (= 50%)	N. A.		2 Mbit/s	1
Main	L3	CCIR 601	16	16 x Main or Core or Simple	4	1620	Progressive: 48600 Interlaced: tbd	24300 (=50%)	t.b.d	1 taps default integer filter	15 Mbit/s	1
Main	L2	CIF	8	8 x Main or Core or Simple	4	792	23760	11880 (= 50%)	t.b.d.	1 taps default integer filter	2 Mbit/s	1
Main	L1	QCIF	*Will*	*not*	*be*	*defined*						
Core	L2	CIF	8	8 x Core or Simple	4	792	23760	11880 (= 50%)	N. A.		2 Mbit/s	1
Core	L1	QCIF	4	4 x Core or Simple	4	198	5940	2970 (= 50%)	N. A.		384 kbits/s	1
Simple Scalable	L2	CIF	4	4 x Simple or Simple Scalable	1	792	23760	N.A.	N.A		256 kbit/s	1 spatial or temporal enhancement layer
Simple Scalable	L1	CIF	4	4 x Simple or Simple Scalable	1	495	7425	N.A.	N.A		128 kbit/s	1 spatial or temporal enhancement layer
Simple	L3	CIF	4	4 x Simple	1	396	11880	N. A.	N. A.		384 kbit/s	N. A.
Simple	L2	CIF	4	4 x Simple	1	396	5940	N. A.	N. A.		128 kbits/s	N. A.
Simple	L1	QCIF	4	4 x Simple	1	99	1485	N. A.	N. A.		64 kbits/s	N. A.

Table 22-1 - Definition of Natural Visual Combination Profiles@Levels.