MPEG-4's BIFS-Anim Protocol: Using MPEG-4 for streaming of 3D animations.

A thesis submitted as a requirement for the degree of Masters of Engineering in Electronic Engineering.

Author: Tony Walsh

Supervisor: Tommy Curran

Dublin City University School of Electronic Engineering

09-September-2000

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Master of Engineering is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: [illegible] ID No: 98971220

Date: 09-Sptember-2000

Acknowledgements

Dedicated to my Parents.

Thanks to Tommy Curran, Sean Murphy, Liam Ward, Noel O'Connor and Eoin Connely.

Abstract

This thesis explores issues related to the generation and animation of synthetic objects within the context of MPEG-4. MPEG-4 was designed to provide a standard that will deliver rich multimedia content on many different platforms and networks. MPEG-4 should be viewed as a toolbox rather than as a monolithic standard as each implementer of the standard will pick the necessary tools adequate to their needs, likely to be a small subset of the available tools. The subset of MPEG-4 that will be examined here are the tools relating to the generation of 3D scenes and to the animation of those scenes. A comparison with the most popular 3D standard, Virtual Reality Modeling Language (VRML) will be included.

An overview of the MPEG-4 standard will be given, describing the basic concepts. MPEG-4 uses a scene description language called Binary Format for Scene (BIFS) for the composition of scenes, this description language will be described. The potential for the technology used in BIFS to provide low bitrate streaming 3D animations will be analysed and some examples of the possible uses of this technology will be given. A tool for the encoding of streaming 3D animations will be described and results will be shown that MPEG-4 provides a more efficient way of encoding 3D data when compared to VRML. Finally a look will be taken at the future of 3D content on the Internet.

Table of Contents

Table of Figures

1 Introduction

1.1 History of Cyberspace

In 1994 one of the most influential science fiction novels of recent times was published. That novel was William Gibson's Neuromancer [10]. The novel Neuromancer portrayed a world in which all the computers of the world were connected together and communicated with each other and the individuals using them. Gibson called this matrix of computers "cyberspace".

At the time Gibson wrote Neuromancer the Internet consisted of a few thousand computers mostly located at universities or defence installations. The Internet at this stage was difficult to use, cryptic command sets were needed and important information was difficult to find. The Internet grew steadily and soon its size was beginning to overwhelm those using it. There was a significant amount of information available via the Internet, but it was often very difficult to find. All this information was stored in separate files: the information was isolated from the rest of the information on the Internet. Consequently, one could spend some time searching for a document on a particular topic. This was a problem especially in the sciences where each new work draws upon the prior work of others.

To help solve this problem Tim Berners-Lee, a software engineer at the CERN institute in Geneva, developed a set of protocols that made it possible to show the connections between documents. These protocols became the foundation for the World Wide Web. Then Marc Andreesen created a web browser that could mix different types of media. Now for example a document could contain both text and images. This browser was given the name NCSA Mosaic. There are really two ages of the Internet, before Mosaic and after Mosaic. Mosaic made it easy to browse the World Wide Web, which today spans the entire Internet.

For some the Mosaic interface was not enough, it did not represent the real world very well. The idea of a three dimensional interface to the web was born. This interface should allow its users to move about and work in 3D and attract many more people to the Internet, as they would be using an environment closer to what they are familiar with. In February 1994 the world's first 3D Web browser was created by Tony Parisi and Mark Pesce. They could use their browser to go to link to a Web site in 3D or click on a link in a Web document and enter a 3D world. Later in 1994 Silicon Graphics released the file format of their Open Inventor scene description language into the public domain. This formed the basis of the VRML 1.0 specification [3]. VRML 1.0 was important as it was the first standard that allowed 3D content to be displayed over the Internet.

While version 1.0 proved the concept of 3Dweb content, Version 2.0 of the VRML [4] standard released in 1997 over comes many of the limitations of the original effort [20]. Version 2 added new functionality: most importantly the ability to move through the scene and to add animations to the scene.

It is possible that we are at the beginning of another revolution in computing as the Internet changes to 3D. Slowly this change is happening as more and more people are using and creating VRML worlds. The type of applications to benefit from VRML are many, they include Games and entertainment, Multi-User Interactive 3D worlds, E-Commerce product and technology demonstrations. No single "killer app" has emerged, Instead most people indicate that a steady growth in the use of VRML continues across a wide variety of application areas. Right now the tools and technology are being released that will allow the web to change to a 3D medium.

1.2 MPEG-4 and Multimedia

The Motion Picture Experts Group (MPEG) have recently developed an international standard that supports the encoding of 3D information. This standard - MPEG-4 - has much more functionality, but the functionality that is of primary concern here is the 3D encoding functionality. This enables 3D content to be stored and replayed at some later date, but, more importantly, it enables 3D content to be distributed over the web. MPEG-4 was developed to provide a standard that will deliver rich multimedia content on many different platforms and networks. The types of applications that will benefit from MPEG-4 are many and found in very different environments [22] and as such MPEG-4 is a very large standard. MPEG-4 should be viewed as a toolbox rather than as a monolithic standard, as each implementer of the standard will pick the necessary tools adequate to their needs, which is likely to be a small subset of the available tools. Anyone who chooses to use MPEG-4 for a particular application has a set of standardised tools at their disposal and can choose those that are most appropriate. Most applications will only require that a small subset of the tools available in the standard be implemented.

As MPEG-4 is a large toolbox, we will investigate those parts of the toolbox that will add extra functionality to 3D applications on the Internet. MPEG-4 defines a scene description language called Binary Format for Scene (BIFS). This scene description language borrows heavily from the existing VRML 2.0 specification and in fact includes all the functionality of VRML. BIFS allows a 3D scene to be defined, such that the user can move around the scene and interact with the scene. The BIFS-Anim protocol allows animations in the scene and the BIFS-Command protocol allows the scene to be updated. The two main advantages in using BIFS are that it is a much more efficient coding scheme and that it supports streaming of 3D animations. We will examine why these two improvements are a significant advance over VRML.

1.3 Research objectives

There are 4 main aims to this research work, they are:

- To develop a good understanding of the technology used in MPEG-4 systems.
- To investigate and compare the scene description approaches taken by VRML and MPEG-4.
- To design and develop a tool for encoding BIFS-Anim bitstreams. BIFS-Anim is a mechanism in MPEG-4 that allows objects in the scene to be animated.
- To take part in the development and verification of the MPEG-4 systems reference software.

1.4 Thesis St: :cture

Chapter 2 examines a number of possible applications that would benefit from the use of BIFS-Anim and MPEG-4 technology.

Chapter 3 will give an overview of MPEG-4. The basic concepts on which the standard are based are first introduced. Particular emphasis is given to the way in which a scene is composed in MPEG-4. Then we will look at the audio, video and systems parts of MPEG-4. With MPEG-4, audio and video compression is changing to encompass the whole concept of multimedia. The rest of this chapter and in fact the rest of the thesis deals with MPEG-4 systems. The systems part of MPEG-4 deals with combining the different media (video, audio) into a complete scene. This chapter will describe the systems part of MPEG-4 up to the scene description.

Chapter 4 describes the MPEG-4 systems scene description language, which is called BIFS (Binary Format for Scenes). We will look at the concepts behind the scene description language such as nodes, fields and events. The compression techniques used by BIFS, context dependency and quantization and the mechanism for updating the scene will also be examined.

The emphasis in chapter 5 is on animation. Here we look at ways of animating objects in the scene. In particular we will examine the BIFS-Anim protocol. BIFS-Anim is a protocol that is used for continuous streaming animation of a scene. We will focus on the configuration of a BIFS-Anim bitstream and the quantization of BIFS-Anim. Finally in this chapter will look at the advantages of BIFS-Anim and some example applications using BIFS-Anim technology.

Chapter 6 starts b, describing the work done by the IM1 MPEG-4 group and the tools they developed. Then we will look at a tool that encodes from a simple description language into BIFS-Anim bitstreams. Finally in this chapter we will look at some results produced by the encoder tool.

Chapter 7 will review all the preceding chapters and offer the conclusions of the thesis. We will also look at possible future areas of research and the possible direction that the technology will follow.

2 Applications for BIFS-Anim

2.1 Introduction

In this section we will describe a number of real world applications using MPEG-4 and BIFS-Anim. Three separate applications will be described and examined. Then we will take a look at the issue of online communities and examine what technology BIFS-Anim and MPEG-4 can add to their design.

All these applications use the Internet as the network layer. This is for two reasons: The compression techniques used in BIFS-Anim reduce the bandwidth requirements for animations down to what can be transmitted over a normal Internet connection. Also at this time the Internet is the best medium which can use the interactivity features of MPEG-4. In all the examples below good quality 3D models and textures need to be created and then those models animated.

2.1.1 3D Soap Opera

The 3D soap opera would work like a normal soap in that at a prescribed time a new episode would be made available to the public. The user would open an Internet page and the latest episode would be streamed using BIFS-Anim and MPEG-4 audio (TTS could also be used). The viewer would watch the 3D characters move about a 3D world and interact with each other to produce a story line.

The 3D soap provides two features not available to a TV program, the ability to navigate and interactivity. The user has the option to move through the 3D scene and view the action from any angle. It is possible to have two parts of the story line happening at the same time and the viewer chooses which one to watch. This idea was used in the successful play Tamara in the late 1980s. Tamara was performed in a large old house, with the audience actually following whichever character interested them from room to room [21]. With MPEG-4 the user can replay the soap multiple times to gather all the different parts of the story line.

With interactivity the viewer can decide on the outcome of the story and which possible outcome to watch. At predetermined times in the story the user may be presented with a number of options, each option resulting in a different outcome. The distinction between watching the story and playing a character in the story may be blurred.

For the above scenario to work the user needs to have an initial scene ready to be loaded into memory. This initial scene is likely to be large as it contains all the textures and models needed for the scene and how those models are used to create the world. For example, if the soap was to take place in an office building the 3D model of the office building with all its rooms, tables, chairs etc needs to be loaded into memory before the animation of that scene may take place.

The big advantage of the MPEG-4 approach is that this initial scene need only be downloaded once and the future animations streamed using BIFS-Anim will change the values of the nodes in the initial scene. Small updates to the initial scene can be made using BIFS-Command. Compare this to the VRML approach where for every episode the whole scene would need to be downloaded.

2.1.2 3D sports

The technology exists today to convert the flat 2D video of a sports event into an animated 3D scene [32]. This conversion can currently not be done in real time. This technology is used by television stations to show short 3D clips of exciting moments from soccer matches.
Using such technology 3D content can be created, and this content can be transmitted over the web using MPEG-4 and BIFS-Anim. This application now becomes similar to the 3D soap opera except that the 3D content is generated automatically. This is an important distinction as quality 3D content is difficult to create.

The viewer now has the option of watching the event from any angle, some examples include.

- To watch the match from the eyes of their favourite player
- To watch the match from the referee's viewpoint
- When a goal is scored change to the goalie's viewpoint
- To browse the match as if browsing a VRML scene

While it is unlikely that 3D transmissions of a match will replace traditional television transmissions the 3D model has an important advantage in that fans of a sporting event or team will be able to follow the action over the Internet if they live outside of the traditional coverage areas.

2.2 *Virtual Communities*

With the advent of the Internet, broadcasters are losing viewers and revenue to more collaborative media. Studies indicate that children from homes connected to the Internet spend more time online than they do watching television [15]. For the larger Internet service providers such as AoL and CompuServe, almost half their customer's time online is spent in chat rooms. Pavel Curtis of Xerox said that "People are the killer application of the Internet."

Thousands of people meet in 3D worlds every day over the Internet. Many play First Person Shooter type games where the goal is to kill the other players before they kill you. Others play online role playing games such as Everquest where the people playing will assume a virtual persona and interact with the other players, often acting out their own story line or even getting virtually married to each other. Everquest has up to 50,000 people on-line at the once. Others spend time in interacting in multi-user VRML worlds [8], where one of the attractions is the ability to add to the world, often in the form of extra buildings or features. For the release of the film Star Trek: First Contact, Paramount included in its web-site interactive VRML recreations of the bridge, engineering room and other well know Enterprise sets. The site received nearly 6 million hits per day making the site one of the most popular film web-sites in the webs short history [12].

One of the big problems with multi-user VRML worlds is the problem of updating the scene, when a user interacts with the scene [6]. Take the simple interaction of a user moving about the world or a user adding a new object to the world. To update the position of the user or to add the new object, every user would need to reload the complete scene again. If BIFS is used then the server containing the world can update the position of the user's avatars by a BIFS-Anim stream or add the new object by a BIFS-Command stream sent to every person online in the world. The other problem with virtual worlds is how to update the user with information that is relevant to them and to ignore any noise. For example imagine a virtual world with 10,000 people online at the one time, updating every user with what each other user is doing all the time would be crazy. Everquest solves this problem by dividing its world up into zones. Each user gets updated with the information relevant to the zone they are in. Changing zones takes a little time as the information for that zone is downloaded.

3 Description of MPEG-4

MPEG-4 is a multimedia standard developed by the Moving Pictures Experts Group (MPEG) of the International Organisation for Standardisation (ISO). MPEG-4 is the third standard in the series (there was no MPEG-3). MPEG-4 is ambitious in scope as it aims to be a complete multimedia standard, where natural recorded images can coexist with synthetic 3D models in the same scene.

MPEG-1 [23] was approved in November 1992 and targeted the storage and retrieval of video and audio on compact disk (bitrates of 1.5 Mbit/s). It is used for DAB in Europe and Canada and an MPEG-1 software decoder is included in the windows 95/NT operating systems. MPEG-2 [24] was standardised in November 1994 and was aimed at digital television. MPEG-2 is also the standard used for DVD players. MPEG-4 was started in July 1993 and has attracted a very large group of experts working to create new multimedia standards. For example at the Dublin meeting in July 1998 over 380 people attended representing 23 nationalities and more than 200 companies. The investment in R&D that the participating companies have made to MPEG-4 is huge.

The scope of MPEG-4 is much larger than its predecessors. MPEG-4 aims at being a complete multimedia standard supporting a large number of possible applications and supporting content delivery over a wide range of bitstreams (Internet, mobile phones to Digital TV).

Within MPEG-4 there are many working groups, these include video coding [25], audio coding [26] and systems [27]. The video coding and audio coding groups look at ways of encoding video and audio signals, both continuing the work done on MPEG-1 and MPEG-2 and also providing new coding methods that fit into the new media object framework. The MPEG-4 systems group greatly expanded its scope. It is the work that the systems group did that sets MPEG-4 apart from the other MPEG standards. MPEG-4 Systems provides the glue that allows all the objects to be composed into a meaningful scene.

In this chapter the idea of MPEG-4 media objects and how they are composed to make a scene will be examined. Then a brief overview of MPEG-4 audio and video will be given. The new features that are in MPEG-4 video and audio and how they add to the object model will be described. Finally an in depth look at MPEG-4 Systems will be taken, stoppi· g at the scene description language which is a topic for another chapter.

3.1 Media Objects

The audio and video components of MPEG-4 are know as media objects [28]. A typical MPEG-4 scene is usually composed of several media objects organised in a hierarchical fashion. As the hierarchical tree is descended the media objects become more primitive. It is important to realise that any media object can exist independently, and that a single media object could represent a whole scene or multiple media objects combined may represent the scene depending on the wishes of the scene creator. Examples of objects include,

- Video objects (e.g. a live newscast)
- Shaped video objects (e.g. a person jogging without a background)
- Audio objects (e.g. the sound of the footsteps of the runner)
- Still images (e.g. a background)
- 2D objects (e.g. scrolling text)
- 3D objects (e.g. a 3D model of a jogger)

Visual objects are given a position within a two-dimensional or three-dimensional space. When defined in a three-dimensional the position of the viewer relative to the objects in the scene needs to be defined. The viewer can change their location and orientation in the scene. The calculations to update the graphics and sound are done locally at the viewers terminal. MPEG-4 has a language called BIFS to define the scene description. BIFS will be examined in chapter 3.

3.2 Natural Audio Coding

MPEG-1 Layer-3 (mp3) audio coding scheme has found its way into many applications including widespread acceptance on the Internet. MPEG-4 audio is designed to be the successor of traditional audio coding schemes such as mp3. The tools produced by the audio group provide scalability and the notion of audio objects.

No single paradigm was found to cover the complete range of bitrate requirements for the MPEG-4, so a number of different algorithms were developed to establish optimum coding efficiency for the broad range of anticipated applications [2]. Figure 1 [2] shows the basic audio framework.

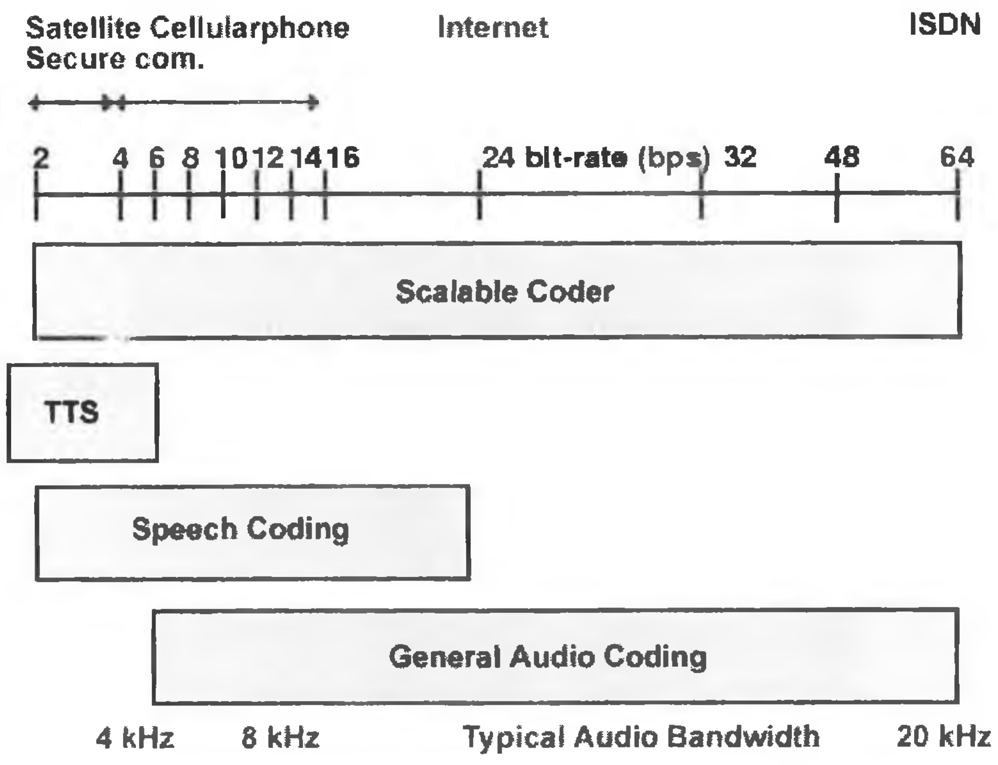


Figure 1 - Assignment of codecs to bitrate ranges

3.2.1 Basic Algorithms

There are 4 main algorithms for MPEG-4 natural audio coding,

- General Audio Coding (AAC based)
  General Audio Coding covers bitrates from 16 kbit/s per channel up to bitrates higher than 64 kbit/s per channel. Using MPEG-4 General Audio quality levels between *better than AM* up to *transparent audio quality* can be achieved. Since MPEG-4 Audio is defined in a way that it remains backwards compatible to MPEG-2 AAC, it supports all tools defined in MPEG-2 AAC. Additionally MPEG-4 Audio defines ways for bitrate scalability.

- TwinVQ
  To increase coding efficiency for coding of musical signals at very low bitrates, the TwinVQ coding tools are defined by MPEG-4 audio.

- HVXC (Harmonic Vector eXcitation Coding)
  HVXC is a speech coding framework supporting low bitrates of 2 kbit/s or 4 kbit/s. HVXC can operate down to an average of 1.2 kbit/s in its variable bitrate mode.

- CELP (Code Excited Linear Predictive coding)
  CELP is a speech coding framework designed for use with bitrates higher than 3.8 kbit/s. CELP supports 2 different sampling frequencies 8KHz and 16KHz. The algorithmic delay by HVXC and CELP is comparable to that of other standards for two-way communications, therefore, MPEG-4 Natural Speech Coding Tool Set is also applicable to such applications.

3.2.2 Scalability

MPEG-4 audio supports bitrate scalability. Bitstream scalability is the ability of an audio codec to support an ordered set of bit streams, which can produce a reconstructed sequence. Moreover, the codec can output useful audio when certain subsets of the bit stream are decoded. The minimum subset that can be decoded is called the base layer. The remaining bit streams in the set are called enhancement or extension layers. Depending on the size of the extension layers we talk about large step or small step (granularity) scalability.

3.3 Synthesised Audio

In addition to its sophisticated audio-compression capabilities, MPEG-4 provides compression, representation, and transmission of synthetic sound and the combination of synthetic and natural sound into hybrid soundtracks [17]. Through these tools, MPEG-4 provides advanced capabilities for ultra-low-bitrate sound transmission, interactive sound scenes, and flexible, re-purposable delivery of sound content.

MPEG-4 provides systems for generating speech sound according to a given text and the capability to generate more general forms of sound including music also according to a given text. These are know a Text-To-Speech (TTS) and Score Driven Synthesis.

Text To Speech:
Text-to-speech (TTS) systems generate speech sound according to given text. This technology enables the translation of text information into speech so that the text can be transferred through speech channels such as telephone lines. Today, TTS systems are used for many applications, including automatic voice-response systems e-mail reading, and information services for the visually handicapped.

TTS systems typically accept text as input and generate a corresponding phoneme sequence. Phonemes are the smallest units of human language; each phoneme corresponds to one sound used in speech. A surprisingly small set of phonemes, about 120, is sufficient to describe all human languages.

The basic MPEG-4 TTS format requires a very low bitrate. A bitstream can be sent containing only the text to be spoken at a bitrate of 200 bits per second. TTS allows a bitstream to be sent that not only contains the basic text but also the detailed prosody of the original speech, i.e. phoneme sequence, duration of each phoneme, base frequency (pitch) of each phoneme, and energy of each phoneme. The synthesised speech in this case will be very similar to the original speech.

One important feature of TTS is the ability to synchronise the lip movements of a facial animation with the synthesised speech. Using this feature of MPEG-4 it is possible to have a synthetic 3D animation of a face, to hear the synthetic speech and to see the lips of the 3D face move in synchronisation with the audio.

Score Driven Synthesis:
MPEG-4 provides structured audio tools that decode input data and produce sounds. A new audio synthesis language called Structured Audio Orchestra Language (SOAL) is defined as part of MPEG-4. This language is used to define an “orchestra” made up of instruments which create and process control data. An instrument is a small network of signal processing primitives that might enable some specific sounds such as those of a natural acoustic instrument.

Careful control in conjunction with customised instrument definition, allows the generation of sounds ranging from simple audio effects, such as footsteps or door closures, to the simulation of natural sounds such as rainfall or music played on conventional instruments, to fully synthetic sounds for complex audio effects of futuristic music.

3.4 Coding of Natural Video Objects

3.4.1 Introduction

Past video compression standards, MPEG-1 and MPEG-2 all work well for the applications for which they were designed (Digital Television, DVD, DAB) but are not flexible enough to address the requirements of a multimedia standard. The MPEG-4 standard provides video coding algorithms designed to allow the user to view, access, and manipulate the content in completely new ways. MPEG-4 video provides tools for Shape coding, motion estimation and compensation, texture coding, error resilience, sprite coding and scalability [9].

3.4.2 Features and Functionality

The most important features provided by MPEG-4 video can be grouped into three main categories.

1. Compression efficiency has been the leading principle for MPEG-1 and MPEG-2, and in itself has enabled applications such as Digital TV and DVD. Improved coding efficiency and coding of multiple concurrent data streams will increase acceptance of applications based on the MPEG-4 standard.

2. Content-based interactivity: Coding and representing video objects rather than video frames enables content-based applications. It is one of the most important novelties offered by MPEG-4. Based on efficient representation of objects, object manipulation, bitstream editing, and object-based scalability allow new levels of content interactivity

3. Universal access: Robustness in error-prone environments allows MPEG-4 encoded content to be accessible over a wide range of media, such as mobile networks as well as wired connections. In addition, object-based temporal and spatial scalability allow the user to decide where to use sparse resources, which

can be the available bandwidth, but also the computing capacity or power consumption.

To support some of this functionality, MPEG-4 provides the capability to represent arbitrarily shaped video objects. Each object can be encoded with different parameters, and at different qualities. The shape of a video object can be represented in MPEG-4 by a binary or a gray-level (alpha) plane. The texture is coded separately from its shape. For low-bitrate applications, frame based coding of texture can be used, similar to MPEG-1 and MPEG-2. To increase robustness to errors, special provisions are made at the bitstream level to allow fast resynchronization, and efficient error recovery.

The MPEG-4 visual standard has been explicitly optimized for three bitrate ranges:
1) Below 64 kbit/sec
2) 64 - 384 kbit/sec
3) 384- 4 Mbit/sec

3.4.3 Structure and syntax

The central concept defined by the MPEG-4 standard is the audio-visual object, which forms the foundation of the object-based representation. A video object may consist of one or more layers to support scalable coding. The scalable syntax allows the reconstruction of video in a layered fashion starting from a standalone base layer, and adding a number of enhancement layers. This allows applications to generate a single MPEG-4 video bitstream for a variety of bandwidth and/or computational complexity requirements. A special case where a high degree of scalability is needed, is when static image data is mapped onto two or three dimensional objects. To address this requirements, MPEG-4 provides a special mode for encoding static textures using a wavelet transform.

An MPEG-4 visual scene may consist of one or more video objects. Each video object is characterized by temporal and spatial information in the form of shape, motion, and texture. For certain applications video objects may not be desirable, because of either the associated overhead or the difficulty of generating video objects. For those applications, MPEG-4 video allows coding of rectangular frames which represent a degenerate case of an arbitrarily shaped object.

An MPEG-4 visual bitstream provides a hierarchical description of a visual scene as shown in Figure 2 [9]. Each level of the hierarchy can be accessed in the bitstream by special code values called start codes. The hierarchical levels that describe the scene most directly are:

- Visual Object Sequence (VS): The complete MPEG-4 scene which may contain any 2-D or 3-D natural or synthetic objects and their enhancement layers.

- Video Object (VO): A video object corresponds to a particular (2-D) object in the scene. In the most simple case this can be a rectangular frame, or it can be an arbitrarily shaped object corresponding to an object or background of the scene.

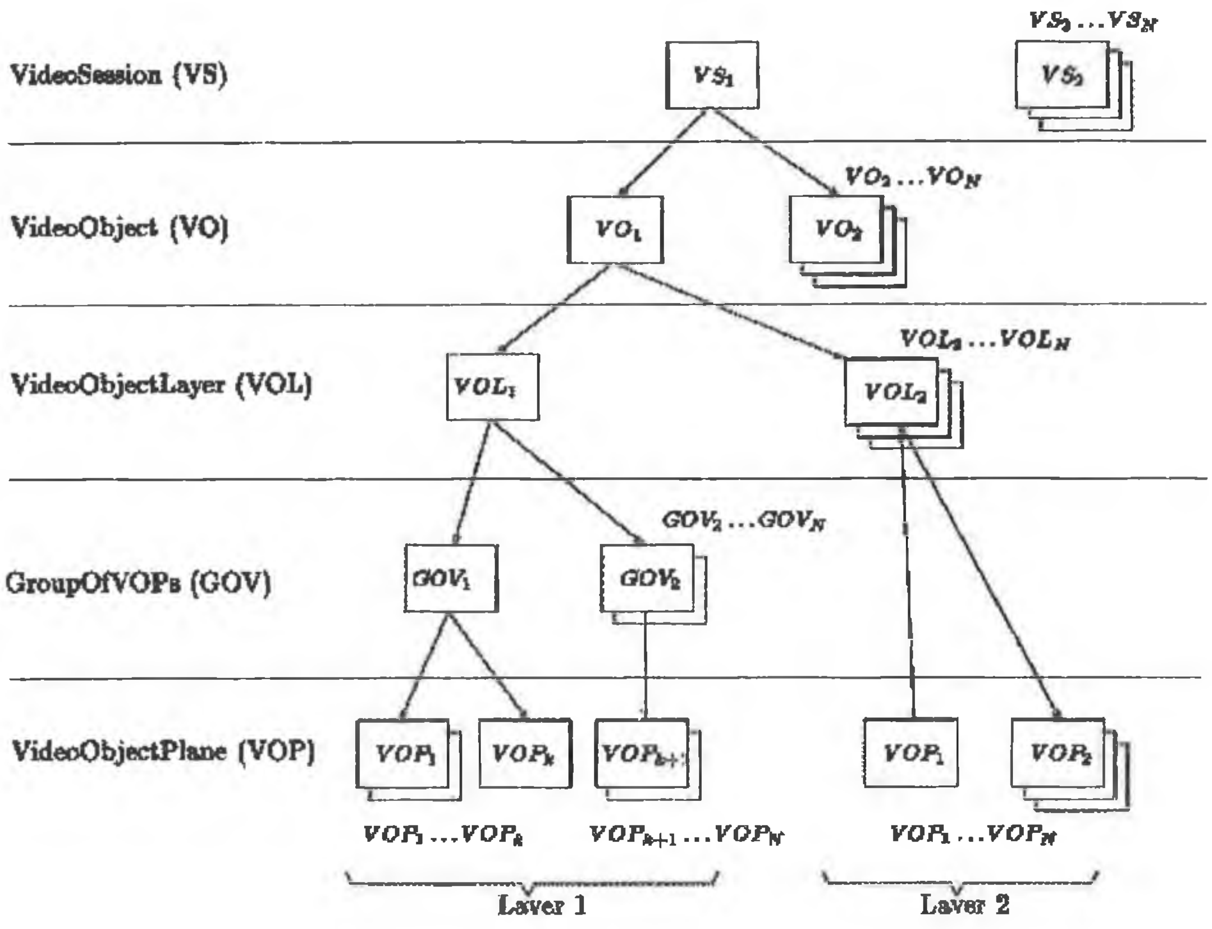


Figure 2: Example of an MPEG-4 video bitstream logical structure

- Video Object Layer (VOL): Each video object can be encoded in scalable (multi-layer) or non-scalable form (single layer), depending on the application, represented by the video object layer (VOL). The VOL provides support for scalabl. coding. A video object can be encoded using spatial or temporal scalability, going from coarse to fine resolution. Depending on parameters such as available bandwidth, computational power, and user preferences, the desired resolution can be made available to the decoder.

Each video object is sampled in time, each time sample of a video object is a video object plane. Video object planes can be grouped together to form a group of video object planes:

- Group of Video Object Planes (GOV): The GOV groups together video object planes. GOVs can provide points in the bitstream where video object planes are encoded independently of each other, and can thus provide random access points into the bitstream. GOVs are optional.

- Video Object Plane (VOP): A VOP is a time sample of a video object. VOPs can be encoded independently of each other, or dependent on each other by using motion compensation. A conventional video frame can be represented by a VOP with rectangular shape.

A video object plane can be used in several different ways. In the most common way the VOP contains the encoded video data of a time sample of a video object. In that case it contains motion parameters, shape information and texture data. These are encoded using macroblocks. It can also be used to code a sprite. A sprite is a video object that is usually larger than the displayed video, and is persistent over time. There are ways to slightly modify a sprite, by changing its brightness, or by warping it to take into account spatial deformation. It is used to represent large, more or less static areas, such as backgrounds. Sprites are encoded using macroblocks.

A macroblock contains a section of the luminance component and the spatially subsampled chrominance components. In the MPEG-4 visual standard there is support for only one chrominance format for a macroblock, the 4:2:0 format. In this format, each macroblock contains 4 luminance blocks, and 2 chrominance blocks. Each block contains 8x8 pixels, and is encoded using the DCT transform. A macroblock carries the shape information, motion information, and texture information. Figure 3 [9] illustrates the general block diagram of MPEG-4 encoding and decoding based on the notion of video objects. Each video object is coded separately.

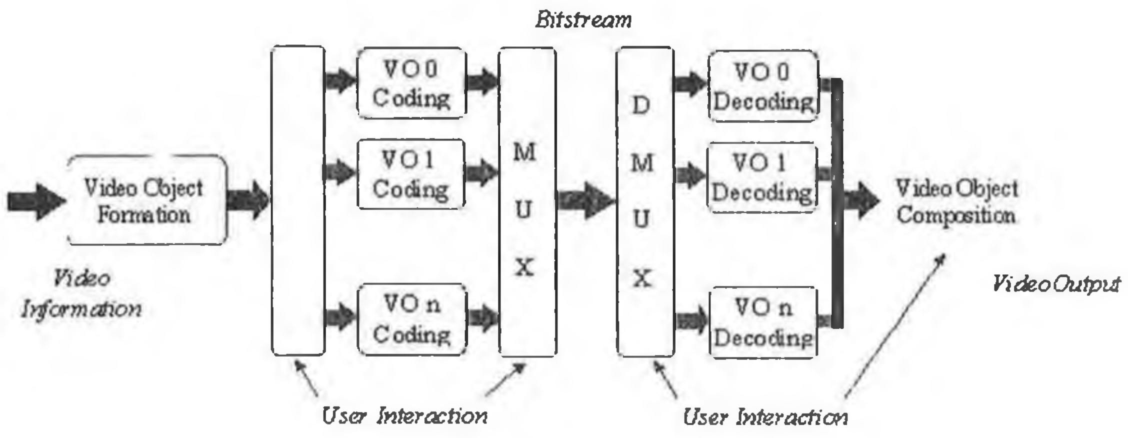


Figure 3: General block diagram of MPEG-4 video

3.5 Systems Architecture

3.5.1 Introduction

MPEG-4 Systems defines a new approach to multimedia. It allows different types of media to be mixed in the same scene and for the viewer to interact with that scene.

A scene consists of audio-visual objects. Examples of audio-visual objects include, video (H263, AVI etc), an audio track, a 3D animation, scrolling text or an animated 3D face. MPEG-4 Systems specify how the objects are composed to make the scene and how the user can interact with the scene. These audio-visual objects are known as Elementary Streams in MPEG-4.

MPEG-4 Systems deal with the coding and streaming of elementary audio-visual sources, and provides a description of how they are combined to form a scene. All the information that MPEG-4 Systems deals with is coded into a binary form, this is to increase bandwidth efficiency. As the audio-visual sources are streamed means that not all of the audio-visual object needs to be downloaded to be displayed. MPEG-4 Systems does not deal with the encoding of audio or visual information but only with the information related to the combinations of streams: combination of audio-visual objects to create an interactive audio visual scene, synchronization of streams, multiplexing of streams for storage or transport.
MPEG-4 Systems is based on a 3 layer architecture, see figure 4 [27] for an overview.

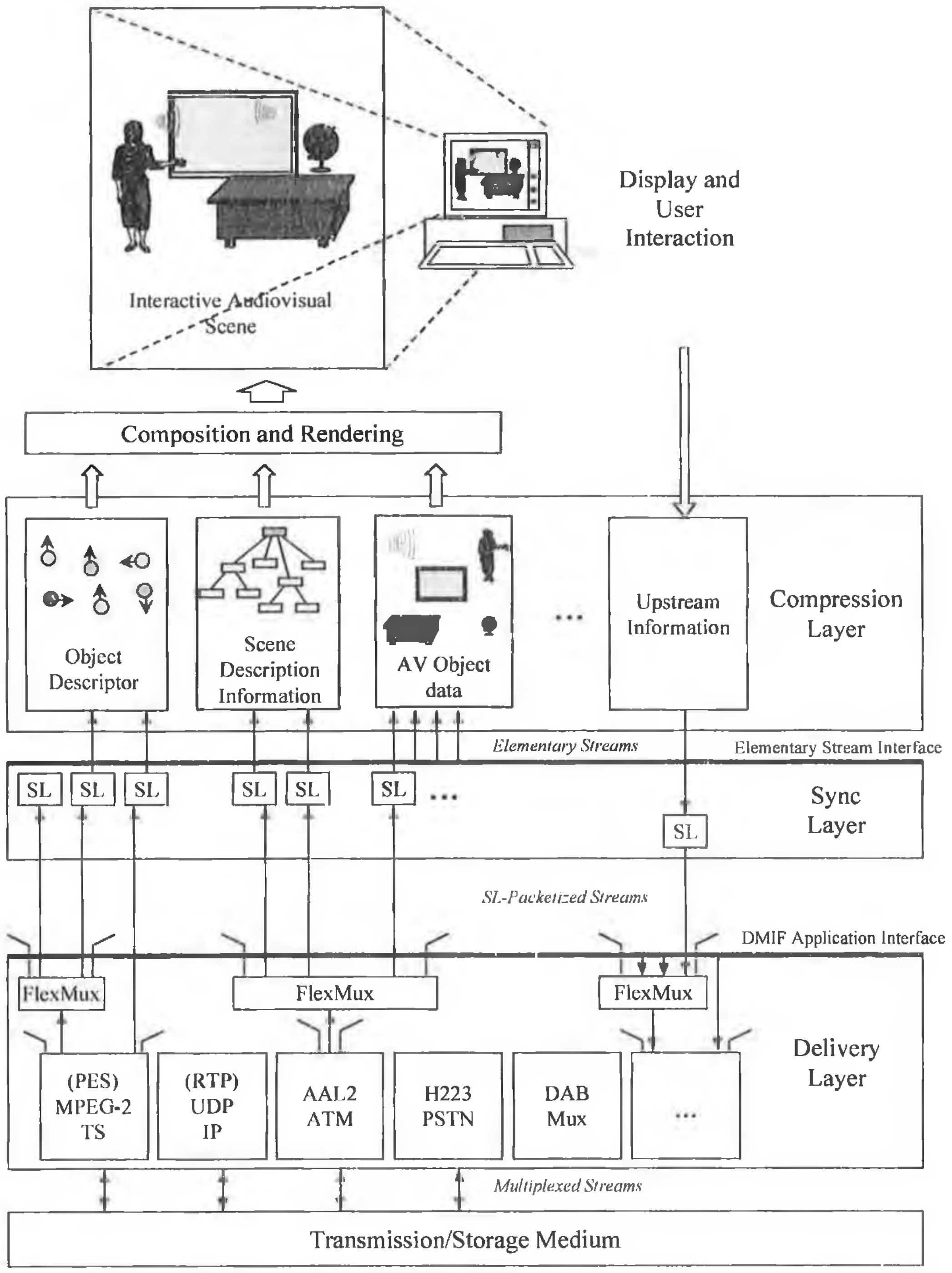


Figure 4: MPEG-4 Systems Architecture

3.5.2 Delivery Layer

The Delivery Layer consists of a transport multiplex and the MPEG-4 multiplex (FlexMux). There are many existing network protocols that provide ways for packetization and transport of data (RTP, MPEG-2, ATM). MPEG-4 allows for the content of a scene to originate from many different locations, also MPEG-4 makes no assumption about the type of underlying communications structure and hence the transport multiplex is not defined by MPEG-4.

The FlexMux is a tool that provides a flexible, low overhead way of interleaving SL-packetized streams. It is not designed to be robust to errors as it can be transported within a robust transport multiplex. The use of the FlexMux is optional.

3.5.3 The Systems Decoder Model

The Systems Decoder Model (SDM) provides an abstract model of the behaviour of an MPEG-4 receiving terminal. The SDM describes the idealised decoder architecture and defines how the receiving terminal will behave in terms of buffer management and synchronisation of elementary streams with each other. The SDM expects the simultaneous delivery of the demultiplexed elementary streams to the decoding buffers of their respective decoders.

The systems decoder model specifies

- The interface for accessing the already demultiplexed elementary streams,
- Required buffer resources for each elementary stream,
- The timing of the elementary streams decoders,
- Composition memory for the Access Units from the decoders,
- The resources needed by the compositor to display the data in the composition memory.

Every elementary stream is made up of a sequence of Access Units (AU). An Access unit is the smallest data entity to which timing information can be attributed (example a frame of video). This is done in a uniform manner for all different stream types in order to ease identification and processing of the access units in each stream. The semantic meaning of an AU is determined by the individual media encoder and is not relevant to the SDM.

Figure 5 [27] gives a view of the Systems Decoder model. Each Access Unit (or part of) is sent to a decoding buffer where it is decoded at its decoding time and then composed on the screen of the receiving terminal. The SDM assumes instantaneous decoding of the access unit, removal from the decoding buffer and insertion of the decoded Access Unit into composition memory. With this model it is possible for the encoding terminal to know the buffer space available at the receiving terminal for a specific elementary stream at any point in time.

The contents of the decoding buffer is used by the decoders to produce Composition Units. The relationship between Access Units and composition units may not be one-to-one, but will be consistent for each specific type. Each composition unit must be ready for composition at a specified time, the two time stamps involved are,

- Decoding Time Stamp (DTS),

  This is the time that the Access Unit must be removed from the decoding buffer and placed in the composition memory.

- Composition Time Stamp (CTS).

  This is the time the Access Unit must be available in the composition memory for composition

All the time stamps are based on an object time base (OTB). The object time base defines the notation of time for a given elementary stream. The OTB is set by the sending side. Since the OTB is not a universal clock, object clock references (OCR) must be sent periodically to the receiver. The value of the OCR corresponds to the value of the OTB at the time the sending terminal generates the object clock reference time stamp.

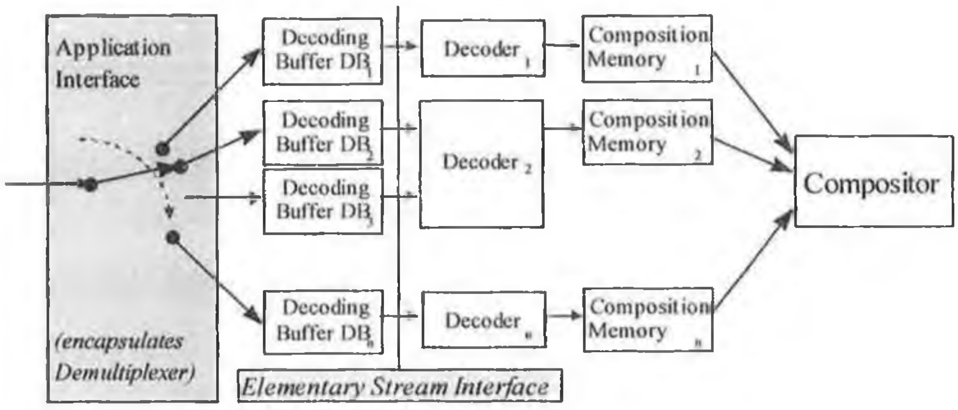


Figure 5: Systems Decoder Model

Figure 6 [27] shows 2 Access Units from the demultiplexer to the decoding buffers. Once the time specified by the Decoding Time stamp is reached, the Access Unit is decoded and the Composition Units are placed in the composition memory. They become available for composition when the time specified by the Composition Time stamp is reached.

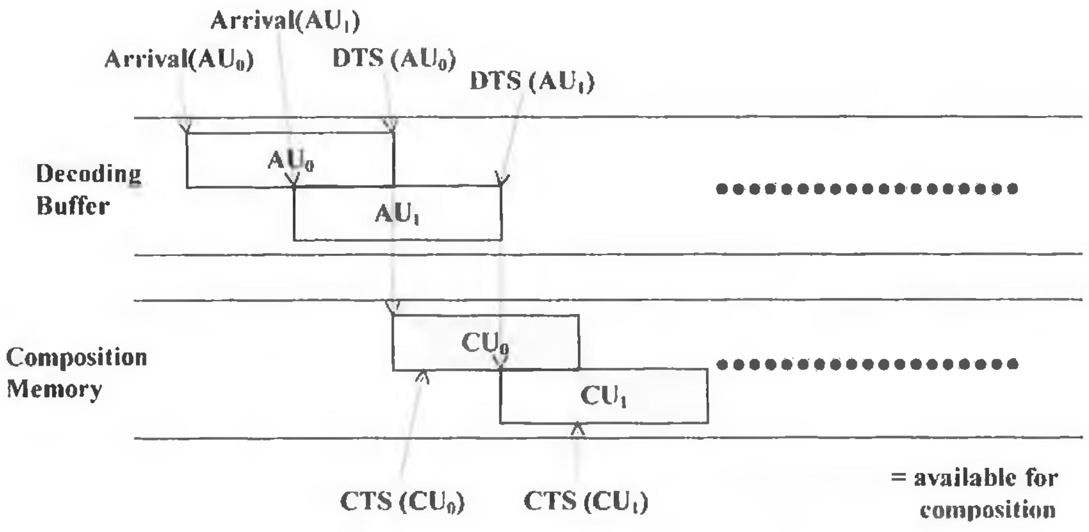


Figure 6: Composition unit availability

3.5.4 Sync Layer

This sync layer defines the tools to maintain synchronisation within and among elementary streams. It supplies information needed by both the delivery layer and the compression layer. On the sync layer, an elementary stream is mapped into a sequence of packets, called an SL-packetized stream (SPS). The sync layer (SL) specifies a syntax for the packetization of elementary streams into access units or parts thereof. See figure 7 [27] for a view of the sync layer.

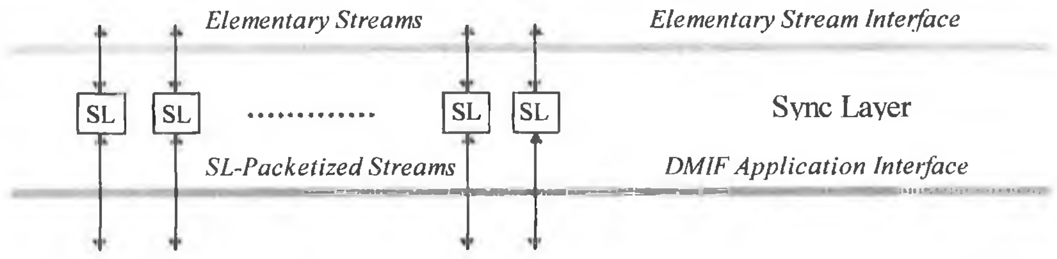


Figure 7: The sync layer

The sync layer can be configured individually for each elementary stream. This is done via the SLConfig descriptor, which is required for the elementary stream descriptor. The SLConfig descriptor defines parameters such as the resolution and accuracy of time stamps and clock references. The length of the parameters can be defined according to the requirements of the elementary stream. Lower bitrate streams may be defined to have timestamps that require fewer bits, thus improving efficiency. The SLComfig may also contain 2 sequence numbers to track lost SL packets and access units.

3.5.5 Compression Layer

Basically the Compression Layer performs all the necessary operations needed to reconstruct the original information and display the scene. The compression layer consists of,

- Scene Description,
  The scene description declares the spatio-temporal relationship of audio-visual objects in the scene. The Scene Description information is sent as an Elementary Stream. The scene description language is called BIFS and will be examined in detail in chapter 3.

- Elementary Streams,
  The data for each audio-visual object is fully or partially coded in a separate stream called an Elementary Stream. Scene description and control information is also coded as elementary streams. Each Elementary Stream contains only one type of data.

- Object Descriptors,
  The Object Descriptor framework provides the link between the Scene Description and the Elementary Streams. The object descriptor stream contains information that the receiving terminal needs to interpret the rest of the elementary streams. This information includes the format of the data as well as an indication of the resources necessary for decoding of the stream.

The scene description information is separated from the stream description information [11]. The scene description contains no information about how to reconstruct a particular audio-visual object. The stream description contains no information about how an audio-visual object is used in the scene. This separation improves content manageability as content providers may relocate streams without affecting the scene description. Figure 8 [1] gives an overview of the compression layer.

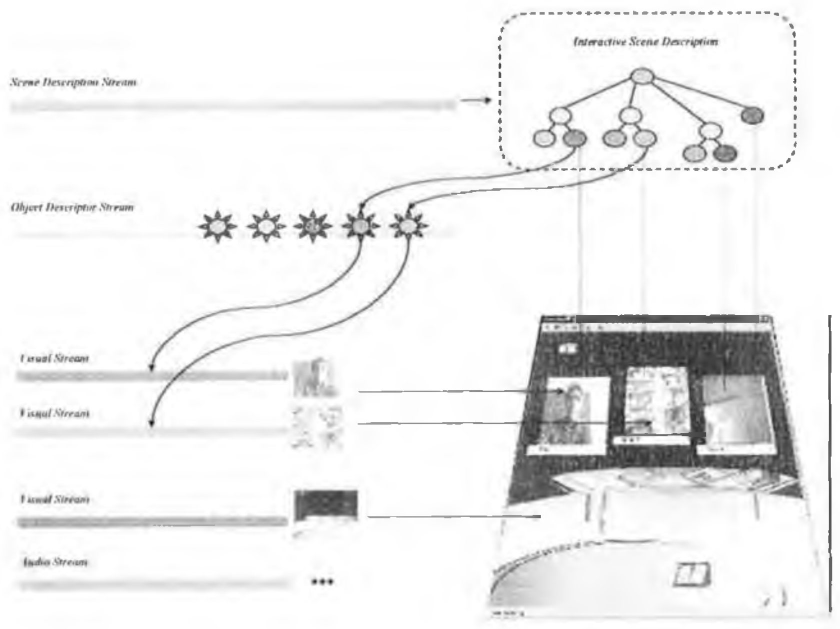


Figure 8: The Compression Layer

The link between the scene description and the stream description is a numeric object descriptor identifier that the scene description uses to point to the object descriptors. These links provide the receiving terminal with the necessary information for assembling the elementary data in order to decode and reconstruct the object at hand.

3.5.6 Object Descriptor Framework

The object descriptor framework is composed of hierarchical structured set of descriptors. The highest level descriptor is the object descriptor itself. The object descriptor is a container that contains a number of other descriptors. Other descriptors provide auxiliary information, to describe textual information about the content of the streams (Object Content Information, OCI) and information about the intellectual property management and protection of the streams (IPMP). The object descriptor elementary stream contains all the object descriptors for the current scene.

The linking of the scene to the elementary streams is a two step process.
Every object descriptor contains a numeric identifier called the object descriptor ID (OD_ID), that is a reference to that object descriptor. This reference is used by the scene description stream to link the object descriptor to the stream.

Each elementary stream is assigned an identifier called the elementary stream ID (ES_ID). The ES_ID is contained within the object descriptor and is used to reference the stream.

In the most simple case, an object descriptor contains one elementary stream descriptor, for example a video stream. An object descriptor can also contain many elementary stream descriptors, possibly to identify a low bitrate version of the stream and a high bitrate one, or to identify different language audio streams. An object descriptor can contain the elementary streams that make up a scalable encoding of audio-visual data.

A special object descriptor called the initial object descriptor is defined to contain the scene description and associated object descriptors necessary for the initial decoding of the scene. The initial object descriptor also contains content complexity information expressed in profile and level indicators and usually contains two elementary stream descriptors one points to the scene description stream and the other to the object descriptor stream (figure 9 [1]).

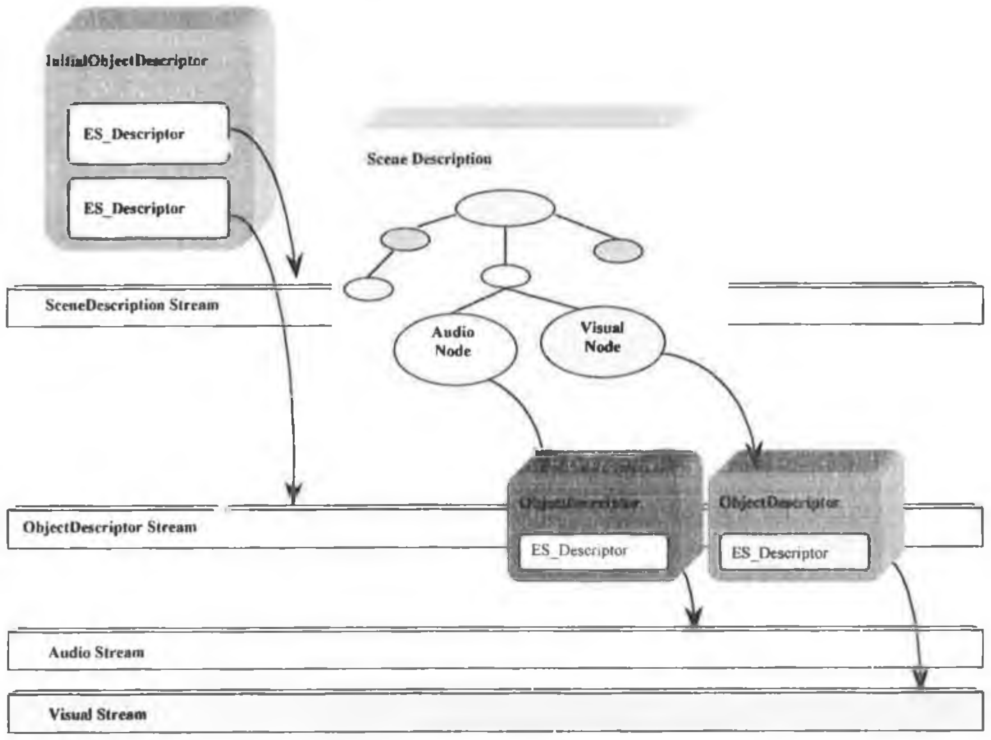


Figure 9: The initial object descriptor

3.5.7 Intellectual property management and protection (IPMP)

MPEG-4 provides mechanisms that allow the owners of content the right to exclude others (with certain limited exceptions) from the use or re-use of their intellectual property without a licence from the IPR owner.

An object descriptor may contain an Intellectual Property Identification (IPI) which is used to contain information about the type of content and standardised identifiers for content, examples are ISBN or ISMN.

While MPEG-4 does not standardize IPMP systems themselves, it does standardize the MPEG-4 IPMP interface. This interface consists of IPMP Descriptors (IPMP-Ds) and IPMP Elementary Streams (IPMP-ES). The IPMP descriptors convey proprietary information that aids in decrypting the elementary streams or conveying authorisation or entitlement information to be used by a proprietary IPMP subsystem at the receiving end.

There will be no standard MPEG-4 IPMP system, this is because the requirements for the management and protection of intellectual property are diverse. All IPMP subsystems will use the standardised IPMP interface, but there is likely to be different IPMP subsystems for a variety of applications. It is left up to the implementers of IPMP subsystems to decide what type of encryption or authentication techniques are used.

3.5.8 Object Content Information

Object content information (OCI) descriptors attach descriptive information to audio-visual objects. This information includes a keyword description (possible used by search engines), context classification descriptors, textual description of the content, language information, content rating descriptions and descriptions about the author of the content and creation date of the content.

All elementary streams contained within the same object descriptor are meant to refer to the same content item, there is no object content information for individual elementary streams. It is also possible to stream object content information to the object descriptor. This allows the object content information to change over time. Object content descriptors and streams may be attached to audio-visual streams and also to scene description streams.

3.5.9 Syntax Description Language

MPEG-4 provides a syntax description language, which is a documentation tool to describe the exact binary syntax of both visual-audio bitstreams, and scene description information. The need for a standard way to represent bitstreams arose from the fact that past MPEG standardisation works used a number of ah-hoc techniques to describe the syntax of their data. SDL was designed to be a natural extension of the typing system of C++.

SDL has been designed to follow a declarative approach to bitstream specification. Developers specify how the data is laid out on the bitstream, and do not detail a step-by-step procedure that parses it. SDL makes it easy and efficient to convert the representation of the bitstream into running code. For a description of SDL see Appendix E [27]. A translator is available that converts SDL into C++ or Java code [29].

4 BIFS and Scene description

4.1 Introduction

BIFS stands for Binary Format for Scenes, that is the language that MPEG-4 uses to describe scenes and also for dynamically changing the scene. BIFS describes the spatial and temporal locations of objects in scenes, along with their attributes and behaviours. BIFS is more than just a scene description language, in that it integrates both natural and synthetic objects in the same composition space. Some objects are fully described within the scene description itself.

4.2 Requirements

We have seen that in MPEG-4 media are separately coded. Also MPEG-4 can transmit 2D, 3D, natural and synthetic media together. MPEG-4 needed a format that allowed both 3D and 2D media to be composed in a 2D or 3D space. MPEG wanted a scene description language that allowed interactivity between these objects. Rather that start from scratch MPEG-4 used VRML version 2.0 as the basis for BIFS.

4.3 Virtual Reality Modelling Language and BIFS

The structure of BIFS is very similar to Virtual Reality Modelling Language (VRML). VRML is a standard for generating 3D content for the Internet. VRML can be looked at as a 3D HTML, as a means of publishing 3D web pages. BIFS and VRML can be viewed as different representations of the same data. In VRML all the object information is transmitted as text like a high level language. BIFS encodes all this information in binary resulting in a much more efficient way of transmitting the same content. Typically BIFS will be 10 to 15 times shorter for the same content [13].

4.3.1 Nodes

BIFS uses a hierarchical scene graph to describe 3D objects and the relationships between them. Entities in the scene graph are called nodes. A node is the fundamental building block of a BIFS scene and nodes can represent a variety of different concepts. BIFS describes about 101 different node types, including basic primitives, lighting properties, sound properties and grouping nodes. Many nodes may contain other nodes (have children) and may be contained in more than one node (parent) but a node must not contain itself. The following are the types of possible nodes:

- Shape node,
  Is the basic container for a geometry object. A shape node contains exactly one geometry node in its geometry field. A shape node also contains nodes that define the geometry's appearance.

- Geometry Nodes,
  These nodes provide basic geometry, examples of this type of node are `Box`, `Sphere` and `IndexFaceSet`.

- Appearance Node,
  This defines surface properties of the object. The types of properties include colour, smoothness of its surface, shininess and texture.

- Grouping Nodes,
  These nodes allow you to group objects together and define a coordinate space for its children.

- Light Source Nodes,
  These nodes define how Shape nodes are illuminated. There are 3 different types of light nodes: `DirectionalLight`, `PointLight` and `SpotLight`.
  - `DirectionalLight` Nodes,
    Are lights considered to be at infinity. A `DirectionaLight` illuminates a scene with parallel rays from the one direction.

    - `PointLight` Nodes,
      A `pointlight` is located at a specific point and illuminates in all directions.
    - `SpotLight` Nodes,
      A spotlight is located at a specific point and illuminates in the shape of a cone in a specific direction.

- Sensor Nodes,
  These nodes keep an eye on the user interactions with the scene and also with the dynamic changes within the scene, examples are `TouchSensor`, `TimeSensor` and `Collision`.

- Interpolator Nodes.
  Interpolator nodes are used to generate linear keyframed animation in the scene. An interpolator node contains 2 fields, `key` and `keyValue`. Key contains a list of keyframe times, each represented as a fraction of the total animation time represented as a floating point number from 0 to 1 inclusive. KeyValue contains a list of values, one for each keyframe to interpolate along. Examples include `PositionInterpolator`, and `ColorInterpolator`.

- Time-dependent nodes
  These nodes activate and deactivate themselves at specified times. Examples include `MovieTexture` and `TimeSensor`.

Each node has a semantic definition that contains the node name, the list of fields in that node with the name of each field, its default values and its event types. For example here is the definition for the Transform node. A Transform node is a grouping node that defines a coordinate system for its children that is relative to the coordinate systems of its parents. As a Transform node is a grouping node it can contain Transform nodes as its children.

```
Transform {
      BboxCenter  0  0  0       #SFVect3f
      BboxSize    -1 -1 -1      #SFVect3f
      Translation 0  0  0       #exposed field SFVect3f
      Rotation    0  1  0  1    #exposed field SFRotation
      Scale       1  1  1             #exposed field SFVect3f
      ScaleOrientation  0  0  1  0  #exposed field SFRotation
      Center            0  0  0     #exposedfield SFVect3f
      Children          [ ]         #exposed field MFNode    }
```

4.3.2 Fields

Every node contains a set of fields that contain the values for that node. There are two different types of fields: fields that can contain a single value, and fields that can contain multiple values. Fields that contain single values are know as "SF" fields and their names begin with "SF" such as `SFColor`. Fields that contain multiple values are know as "MF" fields and their name begin with "MF" such as `MFFloat`. An SF field, even thought it contains only a single value, can contain more than one number, example an `SFColor` node could contain the values 0, 1, 0 representing the colour green. Table 1 gives an overview of the basic field types and the types of values they may contain.

Data Type	**Single Field Type**	**Multiple Field Type**	**Example**
Boolean Value	SFBool	N/A	TRUE
Colour	SFColor	MFColor	(1.0 1.0 0.0)
Floating Point Value	SFFloat	MFFloat	12.98
Image	SFImage	N/A	
Integer Value	SFInt	MFInt	12
A single Node	SFNode	MFNode	Transform{ Box{} }
Rotation around an axis	SFRotation	MFRotation	(1 0 0 3.14) is a 180-degree rotation about x axis
UTF-8 format string	SFString	MFString	"Hello World"
Time value	SFTime	MFTime	0.2
2D vector	SFVect2f	MFVect2f	(2.0 4.0)
3D vector	SFVect3f	MFVect3f	(2.5 4.5 5.5)

Table 1: Basic Field Types

4.3.3 Event Processing

BIFS defines events which provide a message passing mechanism allowing different nodes in the scene graph to communicate with each other. Fields can be labelled of type `eventIn`, `eventOut` or `exposedField`. An event is an indication that something has happened in the scene, for example the user has moved or clicked the mouse, or that a certain amount of time has passed. `eventIn` and `eventOut` define the type of events that each node may receive or generate. Fields that can receive events are labelled `eventIn`, those that send events are labelled `eventOut`, and those that can both receive and send events are labelled `exposedField`.

Most of the nodes in BIFS have at least one `eventIn` definition and hence can receive events. Incoming events are data messages sent by other nodes to the receiving node. These data messages change some state within the receiving node. Many nodes also have an `eventOut` definition. These are used when some state has changed in the source node to send data messages to the destination node.

The link from the node generating the event to the node that is receiving is called a route. Routes are not nodes but a means of creating event paths between nodes.

The event model is an important features in BIFS as it forms the basis for much of the animation and interaction functionality in BIFS.

4.3.4 Reusing Nodes

For reuse of a node the keyword DEF is used. DEF allows a node to be given a name and then to be referenced later by use of the USE or Route statements. Naming a node and referring to that node a number of times is know as *instantiation*. The USE statement will not create a copy of the node, instead the node is placed into the scene graph a second time.

4.3.5 VRML Scene

```
#VRML V2.0 utf8


     DEF LIGHT PointLight {
          on FALSE
          intensity 1.0
          radius 100.0
     }

     Transform {
          translation 2.5 0 0
          children [

          DEF LIGHT_SWITCH TouchSensor { enabled TRUE }
               Shape {
                    appearance Appearance {
                         material Material
                              {diffuseColor 1 0 0
                               ambientIntensity 0.0}}
                    geometry Box {}
               }
          ]
     }

ROUTE LIGHT_SWITCH.isActive TO LIGHT.set_on
```

Figure 10: A sample VRML scene

Figure 10 shows a VRML scene demonstrating many of the concepts explained above. The first node in the example is a `PointLight` node. Note that the `PointLight` will not be active when the scene is first loaded as the `on` field is set to false. The next node in the example is a `Transform` node, which is a grouping node that defines a coordinate system for its children that is relative to the coordinate system of its parents. In this case the transform node has no parents. The next node is a `TouchSensor` node, which interacts with any geometry that's also a child of the same grouping node. A sensor by itself has no geometry or other visible manifestation. If you don't explicitly include some geometry as a sibling of your pointing-device sensor, the user can't click anything and the sensor is useless. A red box is then defined using the `Appearance` and `Box` nodes. Using the DEF statement the `TouchSensor` is named LIGHT_SWITCH and the `PointLight` is named LIGHT. The final statement in the example is a ROUTE command. For the `TouchSensor` the event `isActive` TRUE is generated when the user has the mouse button pressed and the mouse is pointing to the geometry and when the user releases the mouse button (regardless of what the mouse is pointing to) the event `isActive` FALSE is generated. So in our example, when the user presses the mouse button and the mouse is pointing over the box the light is turned on, when the user releases the mouse the light is turned off. Figure 11 shows what the scene looks like using Internet Explorer and Cosmo Player to display the scene.

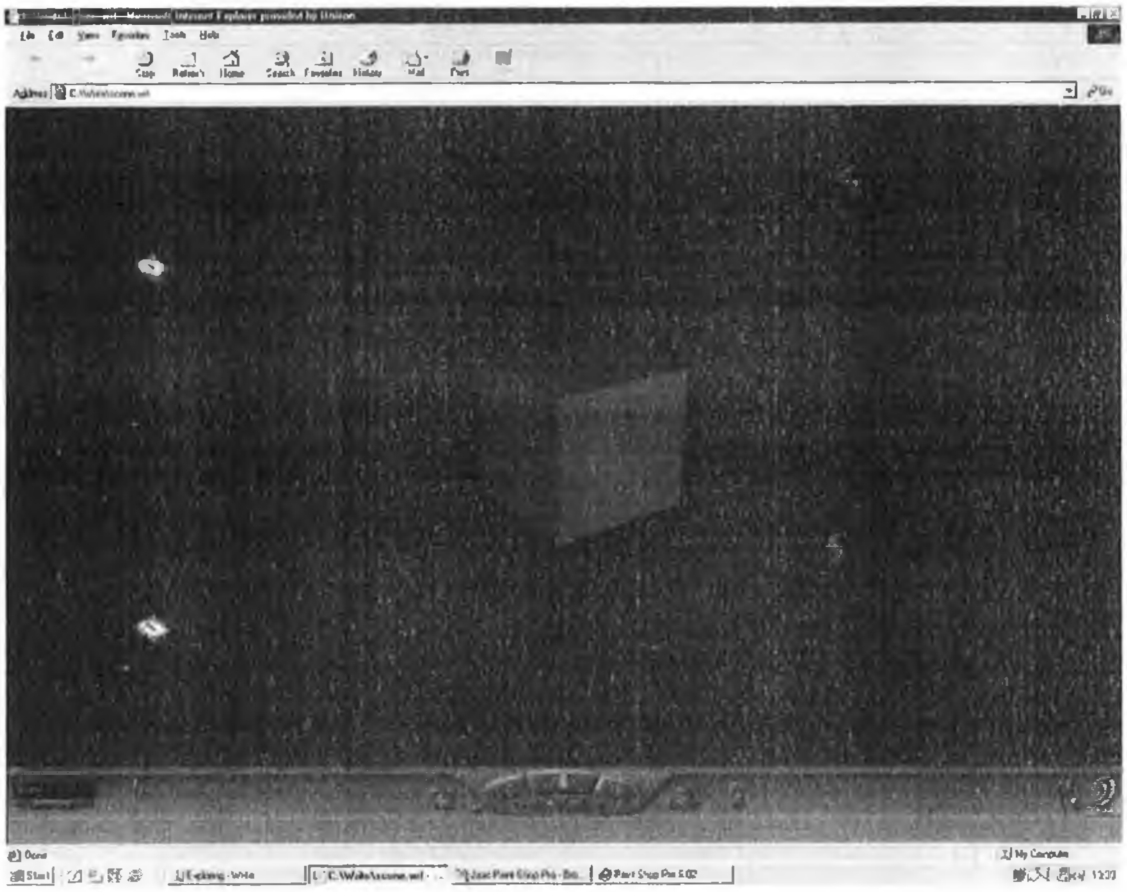


Figure 11: The sample scene displayed using Internet Explorer

4.3.6 BIFS has more Functionality

BIFS extends the VRML specification in the following areas:

New 2D nodes:

A BIFS scene can be fully 2D. This functionality was added to facilitate content creators who wish to provide low complexity scenes. The computer processing required for rendering and navigating fully 3D content is very large. Many industries will need a cheap decoder such as television set-top boxes. BIFS allows 2D and 3D data to be combined in the same scene.

Binary Representation:

VRML is only concerned with scene description, whereas MPEG-4 is also concerned with compression of the scene description and with optimised delivery of the content. The big difference between BIFS and VRML is that a VRML file is readable text file where the BIFS file is a efficient binary representation of the scene.

Facial animation:

A special set of nodes are provided within BIFS for facial animation. These nodes expose properties of a face model that allow that model to be animated in a realistic way. Both realistic facial expressions (smile, frown) and realistic lip synchronisation to speech are provided for. The facial animation parameters are sent by a separate facial animation Elementary Stream.

Enhanced Audio:

New audio nodes support advanced audio features. These nodes provide the functionality for synthesised sound. The audio nodes create their own scene graph called an *audio subgraph.*

Streaming Animations:

BIFS allows an initial scene to be loaded by the receiving terminal and then using a mechanism called BIFS-Anim streams animations in real time to the scene. This mechanism will be looked at in greater detail in chapter 4.

BIFS-Command:

The BIFS-Command protocol allows nodes in the scene to be added/deleted/replaced. Modification of fields and behavioural elements in the scene graph can be replaced. The BIFS-Command information is transmitted in the BIFS-Command elementary stream. The following are the operations provided by BIFS-Command.

- **Scene Replacement:** When a BIFS replace scene command is received, the old scene is deleted and a new scene graph based on the new BIFS scene is created.

- **Node Insertion:** A new node may be inserted into the scene but only into the children field of a grouping node. The node gets added at the desired position in the list of children nodes of an already existing node.

- **Indexed Value Insertion:** Indexed Value Insertion allows the insertion of a new value in a multiple field("MF" field) at the desired position.

- **ROUTE Insertion:** Allows a new ROUTE to be added to the list of ROUTEs for the current scene.

- **Node Deletion:** Node deletion deletes the node with a specific `nodeID`. A node receives an ID by use of the DEF command. Node deletion deletes all instances of that node along with all its fields and any ROUTEs related to the node.

- **Indexed Value Deletion:** Permits the deletion of a specific element in a multiple value field.

- **ROUTE Deletion:** Deletes a ROUTE with a given `routeID`, similar to the deletion of a node.

- **Node Replacement:** Allows the deletion of an existing node and replacement with a new node. All instances and ROUTEs related to the deleted node are also deleted.

- **Field replacement:** Changes the value of a field in a node.

- **Indexed Value replacement:** Changes the value of an element in any position of a multiple valued field.

- **ROUTE replacement:** Deletes an existing ROUTE and replaces it with a new one.

4.4 BIFS Compression

The BIFS language represents a trade off between compression efficiency on the one hand and parsing complexity on the other. A BIFS scene can often be compressed further due to the fact that certain types of data contain redundancy not eliminated by BIFS. One example is that of strings, as BIFS encodes strings as a sequence of characters.

4.4.1 Context Dependency

The use of context dependency is used heavily in BIFS [1]. This technique is based on the fact that once some scene graph information has been previously received, it is possible to anticipate the type and format of data to be received subsequently. This technique is easy to demonstrate. Consider a simple coding scheme with the following tags:

`<begin>`	- beginning of record
`<end>`	- end of record
`<break>`	- end of element in record
`<string>`	- text string follows
`<number>`	- number follows

We wish to use this scheme to code a record consisting of first name, last name and phone number, for example:

First name:	Jim
Last name:	Brown
Phone:	777 1234

With no knowledge context we would need to code this as:

<start><string>Jim<break><string>Brown<break><number>7771234<break><end>

If the context is known, i.e. we know that the structure of the record is "string, string, number" we do not have to spend bits specifying the type of each element:
<start>Jim<break>Brown<break>7771234<end>

VRML treats all nodes as being of type `SFNode` or `MFNode`, while BIFS makes use of the fact that only certain nodes can appear as children of other nodes.

4.4.2 Node Definition Type Tables

BIFS defines over 30 different Node Data Types (NDT) [27] and each node belongs to at least one of them. Each NDT table consists of a list of nodes for each NDT and some context information for that node. In every NDT, each node in the table receives a fixed-binary-length local ID value that corresponds to its position in the NDT. For example, the `Shape` node can be used in both a 3D and a 2D context and therefore appears both in the `SF2DNode` and the `SF3Dnode` node data types. In a 3D context, the `Shape` node can appear in the children field of a `Transform` node. This field has the type `MF3Dnode` and it accepts nodes of type `SF3DNode`. In a 2D context, the Shape node can appear in the children field of a `Transform2D` node, which has type `MF2DNode`. The binary ID code for a Shape node thus depends on the context in which it appears. When it is a `SF3DNode`, it has a 6-bit binary ID value of 100100 (decimal value 36), because it occupies position number 36 in the list of 48 total `SF3DNodes`. In the 2D case, its ID is represented by 5-bits with value 10111. There are only 30 different `SF2DNodes`, requiring only 5 bits to specify them all.

This node representation is quite efficient. It doesn't make sense to apply entropy-coding techniques based on the probabilistic distribution of nodes in scenes since data does not currently exist.

The Node Definition Table for the `SFGeometryNode` is shown in table 2. This table shows a list of the nodes belonging to `SFGeometryNode` with their index in the table. The next 4 columns refer to the number of bits needed to code all the fields of the 4 modes, `defID`, `inID`, `outID` and `dynID`. This will be explained in the next section on the Node Coding Tables.

SFGeome' 'Node		17 Nodes			
reserved	00000				
Bitmap	00001	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
Box	00010	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
Circle	00011	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
Cone	00100	2 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
Curve2D	00101	2 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
Cylinder	00110	3 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
ElevationGrid	00111	4 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
Extrusion	01000	4 DEF bits	2 IN bits	0 OUT bits	0 DYN bits
IndexedFaceSet	01001	4 DEF bits	3 IN bits	2 OUT bits	0 DYN bits
IndexedFaceSet2D	01010	3 DEF bits	3 IN bits	2 OUT bits	0 DYN bits
IndexedLin Set	01011	3 DEF bits	2 IN bits	1 OUT bits	0 DYN bits
IndexedLineSet2D	01100	3 DEF bits	2 IN bits	1 OUT bits	0 DYN bits
PointSet	01101	1 DEF bits	1 IN bits	1 OUT bits	0 DYN bits
Pointset2D	01110	1 DEF bits	1 IN bits	1 OUT bits	0 DYN bits
Rectangle	01111	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
Sphere	10000	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
Text	10001	2 DEF bits	2 IN bits	2 OUT bits	1 DYN bits

Table 2: Node Definition Table for the `SFGeometryNode`

4.4.3 Node Coding Tables

Every node has its own Node Coding Table (NCT). Each node's Node Coding Table holds information on how the fields in the node are coded. For every node its fields are indexed using a code called `fieldID`. This `fieldID` is not unique for each field of each node but varies depending on one of 5 usage categories. For each field of each node, the binary values of the `fieldIDs` for each category are defined in the Node Coding Tables. The categories are:

- `defID`

  Used for those fields that may have a value when declared. This corresponds to fields of type `exposedField` and `field` modes since these are the only modes that have values that can be specified.

- `inID`

  Used for fields whose data can be modified using BIFS command or ROUTEs. This corresponds to the `exposedField` and `EventIn` modes.

* `outID`

  Refers to fields of type `EventOut` and `ExposedField` modes. That is, fields that can be used as input values for ROUTEs.

* `dynID`

  Used for all fields that can be animated using BIFS-Anim. They refer to a subset of the fields represented by `inIDs`.

* `allID`

  Refers to all events and fields of that node. There is an `allID` for each field of a node.

Using these 5 different indexing modes a BIFS scene indexes fields using a minimal number of bits. For example the Viewpoint node defines a specific location in a local coordinate system from which the viewer can view the scene has 8 fields that can be defined at creation. So each field when indexed using the `allID` mode needs 3 bits. The Viewpoint node has only 3 fields that may be animated under BIFS-Anim (`dynID` mode), so only 2 bits are needed to index the field that is animated by BIFS-Anim. The Node Coding Table for the viewpoint node is shown in Table 3. The table shows the node name, its three Node Data Types and their binary representation. The name of each field in the node is then listed along with their field types and the binary ID for the 4 modes. Some fields have a maximum and minimum value specified as well as a quantization and animation category.

Viewpoint	SFWorldNode SF3DNode SFViewPointNode					1100001 110010 1		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
set_bind	SFBool		000					
fieldOfView	SFFloat	000	001	000	00	[0, 3.1415927]	6	8
jump	SFBool	001	010	001				
orientation	SFRotation	010	011	010	01		10	10
position	SFVec3f	011	100	011	10	[-I, +I]	1	1
description	SFString	100						
bindTime	SFTime			100				
isBound	SFBool			101				

Table 3: Node coding table for the Viewpoint node

4.4.4 Quantization

BIFS contains a special node for the purposes of Quantization: the `QuantizationParameter` node. The `QuantizationParameter` node will provide the quantization values to be applied to fields with a numerical type. Quantization of BIFS data efficiently is complex because no clear statistics can be derived from the source data [18]. For quantization 14 quantization categories have been defined (see table 4). Each numerical field in every node will fit into one of these categories. The use of a node structure to convey the quantization parameters allows the use of existing DEF and USE mechanisms that will enable the reuse of the `QuantizationParameter` node. The correct category for each field is got from the Q column of the relevant Node Coding Table.

The information contained within the `QuantizationParameter` for each category is, a boolean that turns the quantization on or off, the minimal and maximum values for the fields and the number of bits to be used for quantization. The `QuantizationParameter` also contains the fields `isLocal` and `useEfficientCoding`. If `isLocal` is false the node will provide the quantization parameters for its children nodes. If `isLocal` is true then quantization will only apply to the following child node. If there is no child node following the `QuantizationParameter` node declaration, the node has no effect. If `useEfficientCoding` is false then the float is encoded using the IEEE 32 bit format for float. If true then the float is coded as an efficient float. Quantization of Bifs-Anim will be looked at in detail in section 4.5.

Category	Description
0	None
1	3D position
2	2D positions
3	Drawing order
4	SFColor
5	Texture Coordinate
6	Angle
7	Scale
8	Interpolator keys
9	Normals
10	Rotations
11	Object Size 3D (1)
12	Object Size 2D (2)
13	Linear Scalar Quantization
14	CoordIndex
15	Reserved

Table 4: Quantization Categories

5 Animation in BIFS

5.1 *Introduction*

Animation is the process of introducing changes to the scene usually to make object in the scene move. The way animation works in BIFS is to change the values of a nodes field over time.

There are 3 ways to animate a scene in MPEG-4 systems, using interpolator nodes common to both VRML and MPEG-4, using BIFS-Command and using BIFS-Anim.

The aim of this chapter is to give the reader a good understanding of the underlying concepts behind BIFS-Anim and to give a view of the types of applications BIFS-Anim can be used for.

5.2 *Interpolator Nodes*

Interpolator nodes are used to generate linear keyframed animation in the scene and are common to both BIFS and VRML. To use an interpolator you specify a list of keyframe times with one set of location values for each keyframe. The playing terminal will interpolate the values in-between the keyframes. Interpolators are restricted to doing only linear interpolation. There are 2 ways to perform non-linear interpolation in VRML.

- You can tessellate the curve into a linear approximation. The resulting interpolator would have many keyframe values close together to simulate a curve by connecting several short lines.

- Write a script that implements the mathematics of the curve function. This way is more efficient in terms of memory usage but is not as fast at executing and displaying the animation.

The six interpolator nodes are,

- `ColorInterpolator` implements keyframe animation on a color value.
- `CoordinateInterpolator` allows keyfame animation on a set of vertices.
- `NormalInterpolator` performs keyframe animation on a set of normals.
- `OrientationInterpolator` implements keyframe animation on a rotation value.
- `PositionInterpolator` allows keyframe animation on a position in 3D space.
- `ScalarInterpolator` allows keyframe animation on a single floating point value.

```
#VRML V2.0 utf8
Group {
     DEF location Transform {
          Translation    1 1 1
          Children [
               Shape {
                    ggeometry Cylinder {
                         radius 5
                         height 6
                    }
               }
          ]
     }
}

DEF mover PositionInterpolator {
     Key        [0,0.1,0.5,0.8,1]
     KeyValue  [1 1 1, 2 1 1, 3 1 1, 6 1 1, 7 7 7]
}

DEF timer TimeSensor {
     CycleInterval 2
}

ROUTE timer.fraction_changed to mover.Key
ROUTE mover.KeyValue to location.Translation
```

Figure 12: A simple VRML scene utilising a `PositionInterpolator`.

Figure 12 shows a VRML scene where a 3D cylinder moves from position 1,1,1 to 7,7,7 over a period of 2 seconds. In the scene the 3 important nodes are the Transform node named location, the PositionInterpolator node named mover and the TimeSensor node named timer. By default when the scene is loaded the timer starts producing fraction_changed events which is the fraction of the CycleInterval completed from 0(start of cycle) to 1(end of cycle). This value is Routed to the Key field of the PositionInterpolator. The PositionInterpolator then produces KeyValue events, which are routed to the Translation field of the Transform node. As the values of the Transform field change the cylinder moves.

5.3 BIFS-Command

It is possible to use the BIFS-Command protocol to perform animations in a BIFS scene. The field replacement operation can be used to change the values in the fields of the node you wish to animate. This method would only be suitable for small animations.

5.4 BIFS-Anim

BIFS-Anim is a streaming protocol provided with MPEG-4 systems. BIFS-Anim was designed to provide a low overhead mechanism for the continuous animation of changes to the numerical values in the scene. BIFS-Anim is a more efficient way of animating the parameters in a scene. It provides a further way of compressing the animation data.

A BIFS-Anim stream is one of many possible elementary streams that can make up an MPEG-4 scene. There are 2 special elementary streams, the scene description scene and the object descriptor stream. The scene descriptor stream describes the spatial and temporal relationships between the audio-visual objects present in the scene. It is the values in the scene descriptor scene that a BIFS-Anim stream will change over time. The object descriptor stream contains information that the receiving terminal needs to interpret the rest of the elementary streams. This is important for BIFS-Aim as the object descriptor stream contains a lot of the information needed to play a BIFS-Anim bitstream correctly.

5.4.1 Configuration

A BIFS-Anim session has two components, the `AnimationMask` and the `AnimationFrame`. Only the `AnimationFrame` is sent in the BIFS-Anim elementary stream.

The `AnimationMask` is sent in the object descriptor for the BIFS-Anim elementary stream. The `AnimationMask` specifies which fields from what nodes in the scene are to be animated. The `AnimationMask` also provides the initial quantization parameters.

The `AnimationFrames` contains the access units of the BIFS-Anim elementary stream. They contain the changes to the numerical information in the scene. The `AnimationFrame` can send information in either intra (the absolute value is sent) or predictive (the difference between the current and last value is sent) mode.

5.4.2 Allowable Nodes

Only updateable nodes, those which have an assigned `nodeID` (using the DEF statement) may be updated by BIFS-Anim. Only fields that have an assigned animation category in the node coding tables may be animated. Finally only those fields of type `eventIN` and `exposedField` (called Dynamic fields) may be animated.

The list of dynamic fields is as follows,

- `SFInt32/MFInt32`
- `SFFloat/MFFloat`
- `SFRotation/MFRotation`
- `SFColor/MFColor`
- `SFVec2f/MFVec2f`
- `SFVec3f/MFVec3f`

5.5 Animation Mask

The `AnimationMask` contains all the set-up information needed by the receiving terminal to play the BIFS-Anim bitstream. This information is contained in the object descriptor stream. The `AnimationMask` contains the following information:

- The number of bits used to represent `nodeIDs`, which defines the maximum possible number of `NodeIDs` available to the session. This is set as 5 bits in the bitstream.
- If the random access is possible (randomAccess flag)
- The number of nodes to be animated
- The ID of each node that will be animated,
- Which fields for each node will be animated,
- If the field is a `MFField`, which of the fields individual elements are to be animated.
- Initial quantization information for each field.

5.5.1 Quantization

In BIFS-Anim field data is always quantized. The AnimationMask defines all the parameters need for quantization [27]. For each animated field the following information is needed,

- Upper bounds of data (floatMax),
- Lower Bounds of data (floatMin),
- Number of bits used to encode a value.

The field is quantized depending on the animation method from the node coding tables.

The following table lists the animation categories.

Category	**Description**
0	None
1	Position 3D
2	Positions 2D
3	Reserved
4	Color
5	Reserved
6	Angle
7	Float
8	BoundFloat
9	Normals
10	Rotation
11	Size 3D
12	Size 2D
13	Integer
14	Reserved
15	Reserved

Table 5: Animation Categories

The value of floatMin is set according to the table 6.

AnimType		**aqp.useDefault**	**floatMin**
4	Color	True	fct.min[0], fct.min[0], fct.min[0]
		False	aqp.IMin[0], aqp.IMin[0], aqp.IMin[0]
8	BoundFloat	True	fct.min[0]
		False	aqp.IMin[0]
1	Position 3D	False	aqp.IMin
2	Position 2D	False	aqp.IMin
11	Size 3D	False	aqp.IMin[0], aqp.IMin[0], aqp.IMin[0]
12	Size 2D	False	aqp.IMin[0], aqp.IMin[0]
7	Float	False	aqp.IMin[0]
6 9 10	Angle Normal Rotation	False	0.0
13	Integer	False	NULL
14,15	Resevea		NULL

Table 6: Value of floatMin, depending on animType

And the value of floatMax is set according to the table 7.

AnimType		**aqp.useDefault**	**FloatMax**
4	Color	true	fct.max[0], fct.max[0], fct.max[0]
		false	aqp.IMax[0], aqp.IMax[0], aqp.IMax[0]
8	BoundFloat	true	fct.max[0]
		False	aqp.IMax[0]
1	Position 3D	False	aqp.Imax
2	Position 2D	False	aqp.Imax
11	Size 3D	False	aqp.IMax[0], aqp.IMax[0] , aqp.IMax[0]
12	Size 2D	False	aqp.IMax[0], aqp.IMax[0]
7	Float	False	aqp.IMax[0]
6	Angl	False	2*Pi
9 10	Normal Rotation	False	1.0
13	Integer	False	NULL
14,15	Resevea		NULL

Table 7: Value of **floatMax**, depending on **animType**

A number of animation types have set bounds, which are `Color` and `BoundFloat`. The animation category `Color` is for every field of `SFColor/MFColor` type and the animation category `BoundFloat` is for those nodes of type `SFFloat/MFFloat` that have an upper and lower bound defined in the Node Coding Tables. For example the field intensity from the node `PointLight` is defined to be within the bounds of 0 to 1. For these animation categories we have a choice to take the default values or to assign or own bounds specified by `aqp.usedefault`.

In the above tables *fct* refers to the Field Coding Table, a data structure defined within MPEG-4. The Field Coding Table contains data need for the quantization of that field. The values for the Field Coding Table are got from the relevant Node Coding Table.

In the above tables *aqp* refers to `AnimFieldQP`, a data structure defined within MPEG-4. The `AnimFieldQP` contains the quantization parameters needed for the animation of ınat field. The `AnimFieldQP` is always set in the `AnimationMask` and the values may be updated by the `AnimationFrames`.

The value of intMax is set according to table 8.

animType	**intMin**
1,2,4,6,7,8 9,10,11,12	NULL
13	aqp.IminInt[0]
14,15	NULL

Table 8: value of intMin depending on animation category

It is also assumed that the following function is available.

`Int getNbbounds(AnimFieldQP aqp)`

Which returns the number of bounds for the animation category received for the following table.

aqp.animType	**value returned**
4,6,7,8 9,10 11,12,13	1
2	2
1	3

Table 9: Return values of getNbBounds

Only the animation categories `Position2D` and `Position3D` have a specific set of bounds for their components. These two categories define quantization bounds for each component of their field. The category `Position2D` is always of field type `SFVect2f` which contains a pair of floating point values. Separate quantization bounds are needed for each of these two values.

The following piece of code in Syntax Description Language defines the bitstream needed to set the values of the `AnimFieldQP` for a field. The `InitialAnimQP` is set in the `AnimationMask`.

```
InitialAnimQP(animFieldQP aqp) {

  aqp.useDefault=FALSE;
  switch(aqp.animType) {

      case  4:        // Color
      case  8:        // BoundFloats
        bit(1)    aqp.useDefault
      case  1:        // Position 3D
      case  2:        // Position 2D
      case 11:        // Size 3D
      case 12:        // Size 2D
      case  7:        // Floats
        if (!aqp.useDefault) {
          for (i=0;i<getNbBounds(aqp);i++) {
            bit(1)      useEfficientCoding
            GenericFloat  aqp.Imin[i](useEfficientCoding);
          }
          for (i=0;i<getNbBounds(aqp);i++) {
            bit(1)      useEfficientCoding
            GenericFloat  aqp.Imax[i](useEfficientCoding);
          }
        }
        break;

      case 13:        // Integers
        int(32)   aqp.IminInt[0];
      break;
  }
  unsigned int(5)     aqp.INbBits;

   for (i=0;i<getNbBounds(aqp);i++) {
      int(INbBits+1) vq
      aqp.Pmin[i] = vq-2^aqp.INbBits;
  }

  unsigned int(4)     aqp.PNbBits;

}
```

The above code does the following,

- Sets aqp.useDefault to be false.
- Performs a switch on the animation category.
- If the current animation category is 4 (colour) or 8 (bound floats)
    - Read a bit from the bitstream
    - If the bit is a 1 set aqp.useDefault to be true
- If the animation category is 4 (colour), 8 (bound floats), 1 (position 2D), 2 (position 3D), 11 (size 3D), 12 (size 2D) or 7 (floats) and aqp.usedDefault is true.
    - Do a loop getNbBounds times
        - Read a bit from the bitstream
        - If bit is a 1 we will use efficient floats
        - Read a float for the bitstream. If we are not using efficient floats this will be 32 bits. Store value in aqp.Imin[I]

        - Do a loop getNbBounds times
            - Read a bit from the bitstream
            - If bit is a 1 we will use efficient floats
            - Read a float from the bitstream. If we are not using efficient floats this will be 32 bits. Store value in aqp.Imax[I]

- If the animation category is 12 (integer)
    - Read 32 bits from the bitstream
    - Store value in aqp.IminInt

- Read 5 bits from the bitstream
- Store value in aqp.InbBits

- Do a loop getNbBounds times
    - Read (aqp.InbBits+1)
    - Store value in aq
    - Assign aqp.Pmin[i] = vq-2^aqp.InbBits

- Read 4 bits from the bitstream
- Store value in aqp.PNBits.

The function GenericFloat encodes values as floats. If `useEfficientCoding` is false then the float is encoded using the IEEE 32 bit format for float. If true then the float is coded as an efficient float.

For Quantization we have,
int quantize (float Vmin, float Vmax, float v, int Nb), which returns

$$v_q = \frac{v - V_{\min}}{V_{\max} - V_{\min}}(2^{Nb} - 1)$$

For Inverse Quantization,
float invQuantize (float Vmin,float Vmax,int vq, int Nb), which returns

$$\hat{v} = V_{\min} + v_q \frac{V_{\max} - V_{\min}}{2^{\max(Nb,1)} - 1}$$

For Animation types 1,2,4,6,7,8,11,12 the process is:

For each component of the field, the float quantization is applied:

$$v_q[i] = \text{quantize}(\text{floatMin}[i], \text{floatMax}[i], \text{v}[i], \text{nbBits})$$

For the inverse quantization:

$$\hat{v}[i] = \text{invQuantize}(\text{floatMin}[i], \text{floatMax}[i], \text{v}_\text{q}[i], \text{nbBits})$$

For integers, the quantized value is the integer shifted to fit the interval [0, 2^{nbBits} -1].

$$v_q = v - \text{intMin}$$

The inverse quantization process in then:

$$\hat{v} = \text{intMin} + v_q$$

For norn :ls and rotations, the quantization method is as follows.

Normals are first renormalized :

$$v[0]=\frac{n_x}{\sqrt{n_x{}^2+n_u{}^2+n_z{}^2}},\quad v[1]=\frac{n_y}{\sqrt{n_x{}^2+n_u{}^2+n_z{}^2}},\quad v[2]=\frac{n_z}{\sqrt{n_x{}^2+n_u{}^2+n_z{}^2}}$$

Rotations (axis $\vec{n}$, angle α) are first written as quaternions :

$$v[0]=\cos(\frac{\alpha}{2})\quad v[1]=\frac{n_x}{\|\vec{n}\|}.\sin(\frac{\alpha}{2})\quad v[2]=\frac{n_y}{\|\vec{n}\|}.\sin(\frac{\alpha}{2})\quad v[3]=\frac{n_z}{\|\vec{n}\|}.\sin(\frac{\alpha}{2})$$

The number of reduced components is defined to be N: 2 for normals, and 3 for rotations. Note that v is then of dimension N+1. The compression and quantization process is the same for both :

The orientation k of the unit vector v is determined by the largest component in absolute value: $k=\text{argMax}(|v[i]|)$. This is an integer between 0 and N that is encoded using two bits.

The direction of the unit vector v is 1 or -1 and is determined by the sign of the component $v[k]$. Note that this value is not written for rotations (because of the properties of quaternions).

The N components of the compressed vector are computed by mapping the square on the unit sphere $\left\{v \;\middle|\; 0\le\frac{v[i]}{v[k]}\le 1\right\}$ into a N dimensional square:

$$v_c[i]=\frac{4}{\pi}\tan^{-1}\left(\frac{v[(i+k+1)\text{mod}(N+1)]}{v[k]}\right)\qquad i=0,\dots,N$$

If nbBits=0, the process is complete. Otherwise, each component of v_c (which lies between -1 and 1) is quantized as a signed integer as follows :

$$v_q[i]=\text{quantize}(\text{floatMin}[0],\text{floatMax}[0],v_c[i],\text{nbBits}-1)$$

The value encoded in the bitstream is

$$2^{\text{nbBits}-1}+v_q[i]$$

The decoding process is the following :

The value decoded from the stream is converted to a signed value

$$v_q[i]=v_{decoded}-2^{\text{nbBits}-1}$$

The inverse quantization is performed

$v_c[i] = \text{invQuantize}(\text{floatMin}[0], \text{floatMin}[0], v_q[i], \text{nbBits} - 1)$

After extracting the orientation (k) and direction (dir) , the inverse mapping can be performed :

$$\hat{v}[k] = \text{dir}.\frac{1}{\sqrt{1 + \sum_{i=0}^{i<N} \tan^2 \frac{\pi.v_c[i]}{4}}}$$

$$\hat{v}[(i+k+1) \bmod (N+1)] = \tan\left(\frac{\pi.v_c[i]}{4}\right).\hat{v}[k] \quad i = 0,\dots,N$$

If the object is a rotation, v can be either used directly or converted back from a quaternion to a SFRotation

5.6 *AnimationFrames*

An AnimationFrame [27] is broken into two, a header and a data part. The header part contains timing information and the data part contains the data for all the nodes to be animated. An AnimationFrame can be sent in Intra or Predictive mode. In intra mode the actual value is quantized and coded, in predictive mode the difference between the quantized value of the current and the last transmitted value of the field is coded.

5.6.1 AnimationFrame Header

The AnimationFrame header contains the following information,

Start Code

A start code may be sent at the start of each frame to enable resynchronisation. The next 23 bits of the bitstream are read ahead and stored in a variable. If this variable is equal to zero, then the next 32 bits are read as the Start Code.

If the frame is Intra or Predictive

A bit is read from the bitstream and if this bit is one then the following frame is an intra frame.

Active Nodes

A bit for every node currently animated. If this bit is 1 then that node will be animated this frame.

Time Code

A bit is read from the bitstream and if 1, then a timecode is expected, so timecode structure is an optional structure.

The time code specifies the time base for the current frame. It has the format shown in table 10 The market bit is a 1 bit integer whose value must always be 1. The purpose of the marker bits is to avoid start code emulation.

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

Table10: TimeCode

Frame Rate

A bit is read from the bitstream if 1, and then a FrameRate structure is expected. Figure 13 shows using syntax description language the structure of FrameRate.

```
class FrameRate {
  unsigned int(8) frameRate;
  unsigned int(4) seconds;
  bit(1) frequencyOffset;
}
```

Figure 13: FrameRate

`FrameRate` is an 8-bit integer defining the reference frame rate.

Seconds is a 4-bit integer defining the fractional reference frame rate.

The frame rate is,

Frame rate = (`FrameRate` + seconds/16)

`FrequencyOffset` is a 1 bit flag, which when set to 1 defines that the frame rate uses the NTSC frequency offset of 1000/1001. If set to 1 the frame rate is,

Frame rate = (1000/1001) * (`FrameRate` + seconds/16)

Skip Frames.

A bit is read from the bitstream and if 1, then a Skipframe structure is expected. Figure 14 shows using syntax description language the structure of Skipframe

```
class SkipFrame {
  int nFrame = 0;
  do {
    bit(4) number_of_frames_to_skip;
    nFrame = number_of_frames_to_skip + nFrame;
  } while (number_of_frames_to_skip == 0b1111);
}
```

Figure 14: `SkipFrame`

`number_of_frames_to_skip` is a 4-bit integer defining the number of frames to be skipped. If `number_of_frames_to_skip` is equal to 1111, then another 4-bit integer follows, if the 8-bit pattern equals 11111111, then another 4-bits follows and so on. Each 4-bit pattern of 1111 increments the `number_of_frames_to_skip` by 15.

5.6.2 AnimationFrame Data

The animation frame data contains the new values for the fields that are being animated for the current frame. The main data structure in the animation frame data is the AnimationField, figure 15.

```
class AnimationField(FieldData field, boolean isIntra) {
  AnimFieldQP aqp = field.aqp;
  if (isIntra) {
      bit(1) hasQP;
      if(hasQP) {
        AnimQP QP(aqp);
      }
      int i;
      for (i=0; i<aqp.numElements; i++)
        if (aqp.indexList[i])
          AnimIValue ivalue(field);
  } else {
      int i;
      for (i=0; i<aqp.numElements; i++)
        if (aqp.indexList[i])
          AnimPValue pvalue(field);
  }
}
```

Figure 15: Animation Field

If the frame is intra then new quantization parameters may be sent. The AnimQP is similar to the `InitialAnimQP`. If new quantization parameters are sent they will remain valid until the next intra frame. If the value of `hasQP` is false then the boolean value `randomAccess` will determine the qunatization parameters to use.

If `randomAccess` is true,

- The `InitialAnimQp` will be used until the next intra frame
- The arithmetic decoder models for the current field will be reset to the uniform models.

If `randomAccess` is false,

- Then the `AnimQP` that was used for the last frame shall be used.
- The arithmetic decoder models for the current field will only be reset if a new `AnimQP` is received.

For intra frame a number of bits equal to InbBits (define in the InitiaAnimQP) is read from the bitstream for each field value. For predictive frames the values is decoded from the bitstream using an adaptive arithmetic decoder as defined in Annex A.

5.7 *Intra and Predictive modes*

If the frame is Intra then the data is read from the bitstream and inverse quantized. The number of bits used to encode the data will be the number of bits read from the bitstream. For predictive frames the value is decoded from the bitstream using an adaptive arithmetic decoder.

In predictive mode, the difference between the quantized value of the current and the last transmitted value of the field are coded [27].

The table that follows shows a simple example. The first row is the value that we want to send. The second row is that value quantized. The third row is the actual value that is encoded into the bitstream. The first value is sent as Intra, the rest as predictive.

Here are our quantization values:
So, 1,2,3,4,5 quantize to 0 and 6,7,8,9,10 quantize to 1 and 11,12,13,14,15 quantize to 2 and 16,17,18,19,20 quantize to 3.
And 0 inverse quantizes to 2, 1 inverse quantizes to 7, 2 inverse quantizes to 12 and 3 inverse quantizes to 17.

Real Value	1	3	4	8	9	12	14	16	18
Quantized Value	0	0	0	1	1	2	2	3	3
Sent Value	0	0	0	1	0	1	0	1	0

On the Decoder Side:

First value: 0 (is intra) and decodes that to 2,

Second value: 0 (is predictive) and adds that to 0 giving 0 decodes to 2,

Third value: 0 (is predictive) and adds that to 0 giving 0 decodes to 2,

Forth value: 1 (is predictive) and adds that to 0 giving 1 decodes to 7,

Fifth value: 0 (is predictive) and adds that to 1 giving 1 decodes to 7,

Sixth value: 1 (is predictive) and adds that to 1 giving 2 decodes to 12,

Seventh value: 0 (is predictive) and adds that to 2 giving 2 decodes to 12,

Eight value: 1 (is predictive) and adds that to 2 giving 3 decodes to 17,

Ninth value: 0 (is predictive) and adds that to 3 giving 3 decodes to 17.

But what about the following? We can not send a –3 as it is not within our quantization bounds.

Real Value	1	3	4	8	9	12	14	15	18	1
Quantized Value	0	0	0	1	1	2	2	3	3	0
Sent Value	0	0	0	1	0	1	0	1	0	-3

Lets take a look at the spec here is part of the `InitialAnimQP`

```
    for (i=0;i<getNbBounds(aqp);i++) {
      int(INbBits+1) vq
      aqp.Pmin[i] = vq-2^aqp.INbBits;
  }

     unsigned int(4)                    aqp.PNbBits;
```

We send 2 values `vq` coded using `INbBits+1` and `PNbBits` coded using 4 bits. `Pmin` is then calculated from `vq`. `INbBits` is the number of bits used to encode intra values, and `2^INbBits` is the number of levels in the quantizer.

If `INbBits` is 2 and `vq` is 0 then `Pmin` will be -4.

Taking `Pmin` to be 4, in our example above we add (subtract –4) 4 to every sent value (predictive only), and that value becomes the new sent value. The decoder will decode the value from the bitstream and then subtract `Pmin`. This value is then inverse quantized.

PNbBits is used to determine the number of symbols in our tables for our Adaptive Arithmetic Encoding.

5.8 Advantages

There are a number of advantages to using BIFS-Anim over interpolator nodes to animate a scene. These are

- Smaller initial download: VRML files can be very large and the whole file needs to be downloaded before any part of the scene is displayed. With BIFS-Anim the initial scene will not contain any animation data and so the initial download is smaller and the scene will be displayed quicker. Once the initial download happens the animation data can then be streamed.

- Compression: BIFS-Anim bitstreams use a number of compression mechanisms, the two main ones being arithmetic decoding and quantization.

- Live transmission: A video camera recording a live event can transmit this data live over a number of network types. A similar idea holds for BIFS-Anim. It is possible to take real world data and use this data to animate a scene in real time.

The VRML standard provides a format for defining a 3D world and as such it is a revolutionary standard allowing the web to be browsed in 3D. It opens up so many possibilities for using 3D content over the Internet.

The two main features lacking in VRML are,

1. No streaming,
2. No binary coding scheme.

Together they mean that the transmission of VRML content is inefficient. Added to the fact that VRML is mainly used over the Internet this inefficiency soon restricts the size of available VRML scenes. The lack of efficiency resulting in large downloads is clearly the biggest reason that VRML has not been successful. Broadband Internet access is seen by some [7] as a solution to this problem. While the introduction of broadband Internet access is likely to increase the success of future web 3D technology, bandwidth will be limited and more efficient coding schemes will always provide a large advantage.

Any future web 3D technology that does not offer streaming is doomed. Right now the Internet can be used to listen to streamed music, streamed radio stations and watch streamed video, why not streamed 3D content. The efficiency gain from streaming is enormous. The next generation of VRML called X3D aims to extend the VRML syntax. Essentially, XML is just another file syntax and there are no plans to extend the VRML standard to include streaming.

6 BIFS-Anim encoding tool

In the last chapter we took an in depth look at BIFS-Anim. We saw how it provides streaming updates to the 3D part of an MPEG-4 scene. In this chapter we will look at some of the work done by the group responsible for providing the reference software for MPEG-4 systems. Then we will examine our work done in developing a BIFS-Anim elementary stream encoding tool, and some results provided by the tool.

6.1 IM1

IM1 is synonymous with the Ad-hoc Group on Systems Version 1.0 Software Implementation [30]. Their mandate is to develop, integrate and demonstrate Systems Version 1.0 software with the help of the committed partners, that means in particular to support the creation of the MPEG-4 reference code and to provide for software and test streams for demonstrating MPEG-4 capabilities before it becomes an international standard.

6.1.1 IM1 Tools

The IM1 group has produced a number of tools which allow MPEG-4 scenes to be created and played. These tools are free to the members of MPEG to use or modify in hardware or software products claiming conformance to the MPEG-4 Systems standard.

The reference software produced by the IM1 group is very important. One purpose of the reference code is to clear up any ambiguities that remain in the standard. In many ways what is contained in the IM1 software becomes the de-facto standard.

- BIFS-Encoder (Bifsenc)

  BIFS-Encoder is a console application which receives one argument – the input file name. The default extension is **.txt**. The application produces two files. Both have the same name as the input file, one with the extension **.bif** the other with the extension **.od**. In addition, a text file with same name and the extension **.lst** is produced. This file lists all the input lines, each followed by error descriptions, if any, and a textual description of the binary encoding.

  The input file contains a textual description of a scene, scene updates and `ObjectDescriptor` commands. The textual scene description that is used by Bifsenc is very similar to VRML. The two output files are binary. The file with the **.bif** extension contains the BIFS elementary stream and the file with the **.od** extension contains the `ObjectDescriptor stream.`

- Multiplexer (Mux)

  The off-line multiplexer is a software tool which reads a set of files, each containing an MPEG-4 elementary stream, and multiplexes them according to FlexMux specifications into one bitstream. The multiplexing includes the process of creating the SL-packetized streams.

  The Mux is a console application. It requires one command argument - a name of a script file. The default extension is **.scr**. The output of the Multiplexer is a file with the same name, and with the .mp4 extension.

  The script file describes the streams that you wish to multiplex. It contains a series of textual descriptions of `ObjectDescriptor` objects, identical to the objects used as input to the BIFS Encoder. Each of the `ObjectDescriptors` contains one or more `ESDescriptor` objects, which describe the elementary streams used by the scene.

- IM1Player

The IM1Player is a “core” player that provides a platform for the full implementation of an MPEG-4 player. It includes demultiplexing, BIFS decoding, scene construction and manages synchronized flow of data between the multiplexer, the decoders and the compositor through decoding and composition buffers. It supports the API for Decoder, DMIF (Delivery Multimedia Integration Framework) and IPMP (Intellectual Property Management and Protection) plug-ins. The core module implements the IM1 Application Programming Interface (API) [31]. The core module is the foundation layer for customised MPEG-4 applications. It contains hooks for plugging in all kind of decoders and customised compositors. Its code is platform independent and has been used by the group as the infrastructure for applications that run on either Windows or Unix.

The core module is accompanied by a test application. The test application is a Win95 “console” application which reads a multiplexed file, uses H.263 and G.723 decoders to decode video and audio streams, and produces two text files. One file shows the “presentation” time of each composition unit (CU), i.e., the time when a plug-in compositor would receive the CU for presentation, compared to the composition time stamp attached to the encoded unit. The other file shows a textual recording of the binary scene description (BIFS).

6.1.2 Compositors

A number of compositors were built using the core module to provide complete MPEG-4 players. There is a 2D only player and two 3D players know as the “Telnor” player and the “Pact” player. These players added a rendering engine to the core module. The “Telnor” player uses OpenGL and the “Pact” player uses DirectX.

Primarily the one person developed the core module over the lifetime of MPEG-4. Parts of the core were contributed by others such as the IPMP module or the BIFS-Anim decoder. This was not the case with the compositors, which witnessed a large number of different people working on them over their lifetime. As such the compositors tended not to be updated along with the core and many nodes were not implemented in the compositor software. It must be remembered that the companies involved in the providing the IM1 are providing their resources for free.

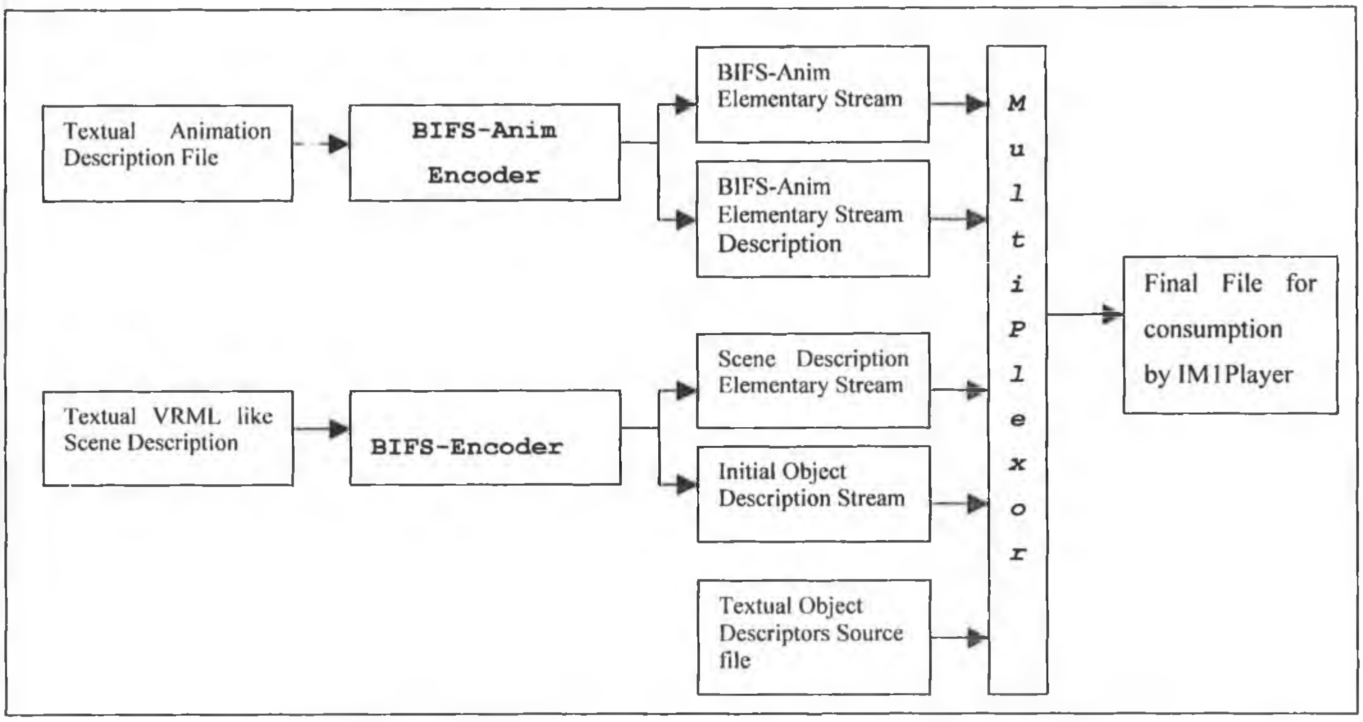
6.1.3 Usage of the 3 tools

By using the 3 tools provided by the IM1 group it is possible to create an MPEG-4 scene and then to play that scene (see figure 16). The procedure is as follows.

First a VRML like text file is produced with the scene description. Following the scene description is the textual description of `ObjectDescriptors` and `ESDescriptors`. This file is passed to the application BIFS Encoder which produces the BIFS file and the `ObjectDescriptor` stream. The BIFS Encoder will convert the textual VRML like scene description into BIFS.

Secondly a script file for the Multiplexer is created. The script file describes the streams that you wish to multiplex. It contains a series of textual descriptions of `ObjectDescriptor` objects, almost identical to the objects used as input to the BIFS Encoder. Each of the `ObjectDescriptors` contains one or more `ESDescriptor` objects, which describe the elementary streams used by the scene. This script file is used for input to the Multiplexer. The Multiplexer will multiplex the files referenced by the script file into one bitstream.

Finally the bitstream created by the Multiplexer is passed to the IM1Player which will consume the bitstream. Annex D provides an example of such a scene.



Textual Animation Description File
BIFS-Anim Encoder
BIFS-Anim Elementary Stream
BIFS-Anim Elementary Stream Description
Textual VRML like Scene Description
BIFS-Encoder
Scene Description Elementary Stream
Initial Object Description Stream
Textual Object Descriptors Source file
Multiplexor
Final File for consumption by IM1Player

Figure 16: Usage of the tools to produce an MPEG-4 bitstream

6.1.4 Verification of IM1

It is also an aim of this project to produce bitstreams that will verify that the BIFS-Anim decoder used within the IM1 software is performing according to the standard. This is done by creating test bitstreams, running these through the IM1 player and then comparing the received output with the expected output. The test bitstreams provided by the tool described here were the only BIFS-Anim bitstreams provided for verification of the BIFS-Anim decoder provided within the IM1 software.

6.2 Content Creation

To create a tool for the encoding of BIFS-Anim bitstreams we need a data format to encode from. So the first problem with creating a BIFS-Anim content creation tool is where to get animation content. A high-level animation description is needed which can then be encoded into a BIFS-Anim bitstream.

A BIFS-Anim bitstream cannot exist by itself as it is linked to whole scene description. BIFS-Anim provides streaming updates to the values in the scene. Not only is a high-level animation description needed but also the tool for building scenes and the playing of those scenes. The only tools available for creating MPEG-4 scenes and playing MPEG-4 content are the tools provided by the IM1 group. It was decided to provide a textual description of the values to be encoded. The textual description is defined in such a way that the data from VRML interpolators can easily be used for our values. This allows VRML scenes to be created using a text editor or a 3D graphical tool and then these VRML scenes can be modified and used as the input to the BIFS-Anim encoder and the IM1 tools.

6.2.1 Structure of Anim description file

This section gives the format of the file consumed by the BIFS-Anim decoder as well as an example file. The structure of the Anim description file was chosen for its simplicity and also ease of parsing.

[NodeIDbits]
followed by the
[NodeID bits]
followed by the
[FrameRate]
Followed by the
[frame rate]

The start of the information for each node must begin with:
[NewNode]

Followed by the:
[nodeID]
[dynID's for Node]

Then for each animated field the:

[field Type in dynID order]

If the node is a MF node then,

[number of multiple values]

[Data]

The file must end with:

[End]

Allowable field types are:

- **SFRotation**
- **SFFloat**
- **SFColor**
- **SFVect3f**
- **MFVect3f**
- **SFInt**
- **Normal**

Example:

NodeID 500 is a Viewpoint node

NodeID 501 is a Coordinate node.

[FrameRate]

[2]

[NewNode]

[500]

[0,1,1]

[SFRotation]

```
[0.0 1.0 0.0 45,1.0 0.0 0.0 90]
[SFVect3f]
[1 2 3,4 5 6]
[NewNode]
[501]
[1]
[MFVect3f]
[4]
[1 2 3 1 2 3 1 2 3 1 2 3,2 3 4 2 3 4 2 3 4 2 3 4]
[End]
```

6.2.2 Scene Creation

Creation of the animation description, scene description and the Multiplexer script file go hand in hand.

The animation description file will reference the following data from the scene description,

- The number of bits used to represent each `nodeID`
- The `nodeID` of each node to be animated
- The type of the node to be animated
- The fields in each node to be animated

The scene description will reference the following from the BIFS-Anim creation tool.

- The name of the Animation Mask file. The Animation Mask file is one of the outputs used by the BIFS creation tool. The animation mask is part of the Object Descriptor stream. The animation mask is binary while the scene description contains a textual description of the object descriptors. The BIFS Encoder will convert the textual object descriptors into binary and will insert the binary Animation Mask into the Object Descriptor stream at the correct location.

The script file will reference the following from the BIFS Encode.

- The name of the BIFS stream.
- The name of the Object descriptor stream.

The script file will reference the following from the BIFS-Anim creation tool.

- The name of the file containing the BIFS-Anim stream

The Multiplexer will multiplex the 3 streams, BIFS, Object Descriptor and BIFS-Anim according to FlexMux specifications into one bitstream. This bitstream will be consumed by the Im1Player.

6.2.3 Analysis of Anim Description file

It is observed from the systems specification that each field type (`SFFloat`, `SFColor` etc) in MPEG-4 will be encoded for BIFS-Anim in its own way. It does not matter which Node or Field the field type belongs to. For example the position field in a Viewpoint node is encoded in the same way as a scale field in a Transform node is encoded. Both fields are of type `SFVect3f`.

The only information that a BIFS-Anim encoder needs to know about each node is the nodes NodeID, the `dynId's` for the node, and the field type for each dynID. The number of `dynId's` for each node is defined in the Node Coding Tables. Each `dynId` is represented in the Anim description file as a '1' or a '0'. If the `dynId` is '1' the field that the dynID represents will be animated. The order of the `dynID's` is the order the field is defined in the Node Coding Tables read from top to bottom.

Each field type representing a positive `dynID` and the data associated with the field follow the `dynID` definition. These field types must be defined in `dynID` order due to the context dependency inherent in BIFS.

6.3 The Encoding Tool

6.3.1 Overview

The encoding tool was developed using MS VC++ 6 and under Windows 95. The encoder takes as its input the Anim Description file, parses this file and produces two binary output files containing the BIFS-Anim bitstreams.

The BIFS-Anim creation tool consumes a file containing the Anim descriptions and produces two files containing the Animation Mask and the Animation Frames. The Animation Mask contains the information necessary for the decoder (player) to read the Animation Frames. The operation of the tool is split into two parts. The first part parses the description file and stores its contents into data structures. The second part performs operations on the data structures to write the Animation Mask and the Animation Frames. The Animation Frames will consist of an Animation Frame header followed by the animation data for every frame of animation.

6.3.2 Parsing and main Data Structures

The description file is parsed and the data is stored in the relevant data structures. The main data structure is the NodeData class. The parsing fills in the following variables:

- The `NodeID`
- Number of Fields in the node
- The Number of fields to be animated
- An array containing the `dynIDs`
- A data structure called FieldData containing the data for each field in the node to be animated. This is an array containing 1 entry for each field to be animated.

The NodeData structure also contains the following,

- A binary variable containing the active mask.

The Parsing fills in for the FieldData structure the following,

- Number of Bounds (`nbBounds` see table 9)
- Number of Components to the data in the field (`SFColor` has 3, `SFFloat` has 1 etc)
- The number of elements for MF fields
- An array containing the data. The data is store sequentially in the order that each piece of data is read from the file.

The FieldData structure also contains the following,

- The value of `INbBits` (The number of bits used to encode a value for quantization for this field)
- The value of `PNbBits` (Used to determine the number of symbols in our tables for the Adaptive Arithmetic Encoding.)
- An array, containing the minimum bounds for the field. This array will be number of Bounds for the field in size.
- An array, containing the maximum bounds for the field. This array will be number of Bounds for the field in size.
- An array, containing the last transmitted value, used in predictive mode. This array will be the number of components for the field in size.
- A variable containing the value of VQ.
- An array, containing the symbols for the arithmetic model for the current field. This array will be `PNbBits` in size. There will be a separate array for the number of bounds for the current field (a 2 dimensional array).
- A variable (AUcounter) that acts as a pointer to the next piece of data from the data array.
- An array which will be used to store the data for the current frame. This will be used to compare the current data with the next frames to determine the active mask and the number of skip frames.

6.3.3 Utility Functions

A number of utility procedures were created to deal with outputting binary values to a buffer. The three main functions are

- int IntegerToBinary (int IntValue, int nBits)
- void FloatToBinary (float FloatValue)
- void FloatToEfficientFloat (float FloatValue)

The function IntegerToBinary is called with 2 parameters. IntegerToBinary will encode the value IntValue using nBits to the array. FloatToBinary will encode the value FloatValue using 32 bits and FloatToEfficientFloat will encode the float as an efficient float. For example to encode the value '0' using two bits the function IntegerToBinary is called with the parameters (0,2).

6.3.4 Operation of the Encoder

I will now describe the main operations of the Encoder.

1. The Animation Description file is parsed and the relevant data structures are filled
2. The Animation Mask is generated. This is a two stage process: First a buffer is filled with the binary data then this buffer is written to a file. All the data needed for the animation mask is easily retrieved directly from the data structures.
3. The Animation Frames are then generated once again a buffer is filled with the binary data and this buffer is written to a file.
4. The Animation Frame header is created.

   - The frame rate is encoded in the Animation Frame header and is optional. The frame rate for the sequence is only encoded once and is sent with the first Animation Frame.

- In the Animation Frame header it is possible to exclude nodes from the current frame. For each node a ‘1’ or ‘0’ is sent (mask.isActive). If the node is to be animated in this frame a ‘1’ is sent. If the values for each field in a node are the same as for the last frame this node is excluded from the current frame and a ‘0’ value is sent for that node. It is not possible to skip a field in a node if any of its other fields are to be animated this frame.

- The number of skip frames is encoded next and is optional. If the values for every field in every node for the current frame are the same as the last frame, skip frames will be sent. The value of skip frames will be the same as the number of identical frames in sequence.

5. The Animation Frame data is encoded next, the process is slightly different depending on whether the frame is Intra or Predictive.

```
If the frame is Intra
    The arithmetic model for that field is reset.
    For each component in the field,
            The next value is read from the data array
            The value is quantized
            The quantized value is put into the buffer using InbBits bits
            The last transmitted value is assigned the quantized value
            The pointer to the next piece of data is incremented

If the frame is Predictive.
    For each component in the field,
            The next value is read from the data array
            The value is quantized
            The last transmitted value and value of Pmin is subtracted from the
            quantized value (see 3.7)
            This value is encoded to the buffer using the arithmetic encoder
            The last transmitted value is assigned the quantized value
            The pointer to the next piece of data is incremented
```

6. Once the header and all the data has been encoded to the buffer, this buffer is then saved to a file

7. A function is then called to determine the number of skipped frames if any and the active mask for the next non skipped frame by reading ahead and comparing the data for the current frame with the data for the following frame. If there are skipped frames the pointer to the next piece of data for each field will point to the start of next non skipped frame.

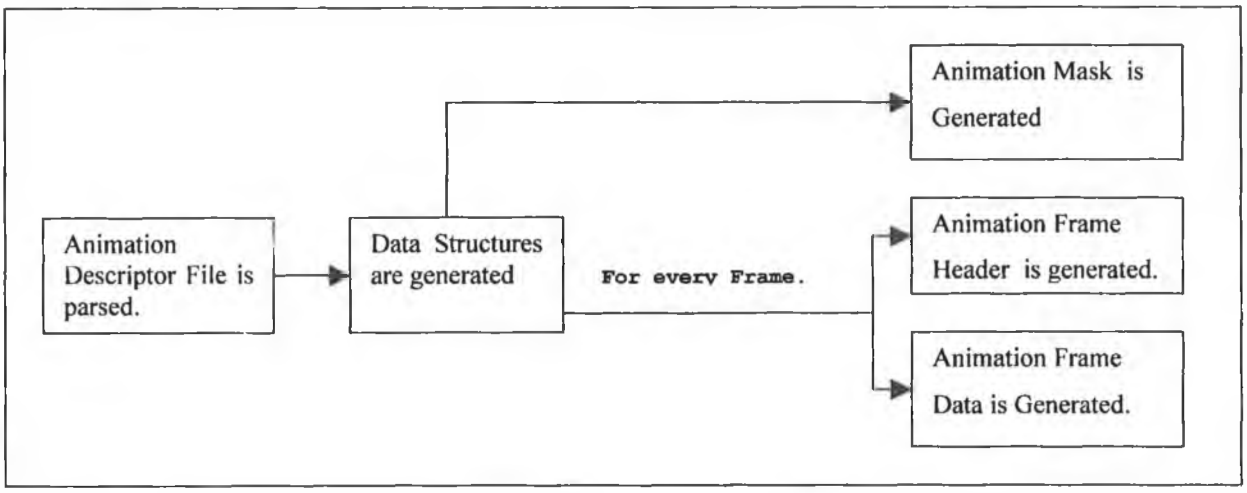

Animation Mask is Generated
Animation Descriptor File is parsed.
Data Structures are generated
For every Frame.
Animation Frame Header is generated.
Animation Frame Data is Generated.

Figure 17: Overview of the operation of the Encoder

6.3.5 IM1 BIFS-Anim Decoder

To test the BIFS-Anim encoder some simple BIFS scenes with corresponding BIFS-Anim bitstreams were created. The IM1 player was used to play the scene and the outputted results compared to the expected results. It was soon noticed that the outputted results were not as expected. This was found to be due to errors in the BIFS-Anim decoder code within the IM1 player. The person who developed the original code was no longer available to work for IM1, so the errors in the BIFS-Anim sections of the IM1 player needed to be fixed. The main errors in the code were.

- Predictive frames not implemented correctly.
  The decoder was not expecting to be sent the difference between the quantized current value and the last transmitted value. It appears that the decoder was expecting the difference between the quantized current value and the last real value sent, which was completely wrong.
- Pmin not taken into account.
  The value decoded from the bitstream did not have Pmin subtracted from it for predictive frames.

- Inverse quantization of fields with number of bounds equal to one was incorrect.
  Any field that had more than one component but only one bound (e.g. `SFColor`) was inverse quantized incorrectly.
- Normals and Rotations not inverse quantized correctly.
  There were a number of errors with the inverse quantization process for rotations and normals. One of these was due to an error in the standard, which called for 2^nbbits-1 to be subtracted twice. This error will be corrected in a future corrigendum to the systems standard.

6.4 Testing

There were 2 separate phases of testing. The first was to test the operation of the encoder and the IM1 decoding tools. The second was to produce results showing the compression gains from using BIFS.

6.4.1 Verification Testing

For the first round of testing a number of simple scenes and animation data for the scenes were created. For each field type that could be animated a scene was created. This animation data was encoded with the BIFS-Anim Decoder tool and the resulting scene was played using the IM1Player. The purpose of this testing was to verify that the IM1 software was decoding the BIFS-Anim bitstreams correctly. Any errors in the decoding were corrected (see 5.3.5 above). The scene description and animation description for these scenes was created using a text editor. This testing verified that the encoder could encode BIFS-Anim bitstreams for each field type and that the player could decode the BIFS-Anim bitstreams and play the scene.

6.4.2 Compression Testing

To look at the compression rates provided by BIFS-Anim, some real world data needed to be generated. To generate this data using a text editor would be very difficult. Also it proved impossible to use existing VRML scenes as the animation was expressed as interpolator nodes. The animation description format was designed so that it is easy to convert from VRML interpolators to the animation description format but the VRML interpolator must include a key value for every frame of the animation.

The commercial 3D modelling and animation package 3D studio Max was used to generate the test sequences. One of the features included with 3D studio Max is the ability to export its scenes as VRML. With the 3D Studio Max VRML exporter it is possible to specify a frame rate for the animated sequence. The exported VRML interpolators then have a key value for each frame in the animation. For example if when exporting the sequence a frame rate of 20 frames per second is chosen and the animation lasts 1 second, then the exported interpolators will have 20 key frames and 20 key values. For a simple animation, say moving an object from 1 point to another with a constant velocity, only 2 key frames and 2 key values are needed to represent this as a VRML interpolator. When 3D Studio Max exports this scene it will have many key frames and key values which is an inefficient way of representing the VRML but suits the needs of the animation description format.

Two test sequences were created using 3D Studio Max. The first was a series of five simple shapes moving around the screen, the second is an animated 3D scene of a man walking (see figure 17).

The process for generating the test scenes is as follows. First a scene is created using the graphical tools provide by 3D Studio Max. This scene is then exported to VRML with the desired frame rate specified. The VRML scene can then be viewed using any VRML browser. The VRML file is then split into two. The first part represents the static scene and the second part contains the VRML interpolator data. The first part after some modification is consumed by the BIFS-Encoder tool to produce the scene description elementary stream and the object descriptor elementary stream. The modification consists of adding `nodeID's` to the VRML and changing the VRML syntax for the BIFS-Encoder tool. The part containing the interpolator data is then converted to the animation description format. The animation description file is then consumed by the BIFS-Anim encoder tool to produce two files. The first file contains the Animation Mask and the second contains the Animation Frames. A script file for the Multiplexer is created using a text editor. The Multiplexer is then run, which produces a file with the extension **.mp4,** which is consumed by the player.

6.4.3 Description of the 3D scenes

The first scene contained 5 simple shapes; a sphere, a box, a cylinder, a cone and a pyramid. These 5 shapes moved around the screen in all 3 directions. The total animation time was 16 seconds with a frame rate of 10 frames per second.

The VRML file for the scene with the 5 simple shapes is 24 KB. The scene is composed of 5 separate interpolators, one for each of the shapes. This scene split into the static scene at 3 KB and the interpolator data at 19 KB. The interpolator data was then converted into an animation description file at 13 KB in size. The difference in size result from the fact that the animation description data only contains the keyframe values and not the keyframe times. The keyframe times are not needed as it is assumed that the frame rate will be constant over the length of the scene.

The "figure walking" scene is composed of 16 simple objects each representing a different body part. Each of these objects when exported to VRML is represented by the `IndexedF ceSet` node. A colour was assigned to each of the body parts. The total animation time is 12.3 seconds and the frame rate is 10 frames per second.

The size of the VRML file resulting for the 3D scene of the "figure walking" is 340 KB. The VRML scene is composed of 53 separate interpolators each running at the same time. This split into the static scene at 40 KB and the interpolator data at 269 KB (unnecessary spaces were removed). The interpolator data was then converted into an animation description file at 130 KB in size.

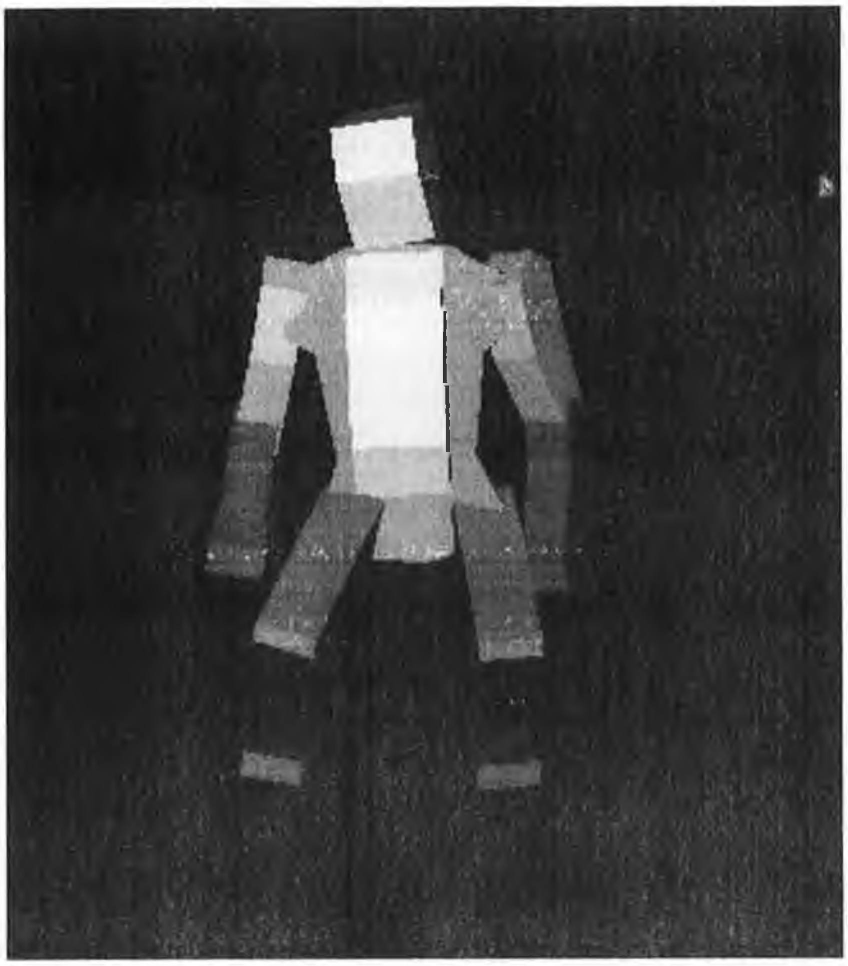


Figure 18: Animated Figure

6.4.4 Results

The first frame is always encoded as an intra frame and the tests were run twice, once using predictive frames and once using intra frames. No quantization was used in the scene description. The results are as follows (see tables 11 and 12).

- Scene with the simple shapes:
  For the scene with the simple shapes the size of the scene description elementary stream was less than 1 KB (485 bytes).

Firstly using predictive frames the size of the BIFS-Anim bitstream was 0.8 KB. When the scene was run using only intra frames the resulting BIFS-Anim bitstream was 1 KB.

- Scene of the “figure walking”:

  For the scene with the simple shapes the size of the scene description elementary stream was 16 KB.

  Firstly using predictive frames the size of the BIFS-Anim bitstream was just under 5 KB. When the scene was run using only intra frames the resulting BIFS-Anim bitstream was 9 KB.

Sizes	**VRML**	**Interpolator data**	**Animation description**	**Animation Frame**
“Simple Shapes”	24 KB	19 KB	13 KB	.8 KB
“Figure Walking”	340 KB	269 KB	130 KB	5 KB

Table11 Test values for Predictive frames

Sizes	**VRML**	**Interpolator data**	**Animation description**	**Animation Frame**
“Simple Shapes”	24 KB	19 KB	13 KB	1 KB
“Figure Walking”	340 KB	269 KB	130 KB	9 KB

Table12 Test values for Intra frames

6.4.5 Analysis of Results

Our example scenes shows clearly the advantages of the MPEG-4 approach. In the case of the "figure walking" scene a download of 340 KB to play a 12 second scene was reduced into a 16 KB initial scene description download followed by a 5 KB stream, which can be streamed at less than 1 KB per second.

If we assume an Internet connection of 5 KB per second we have an 86 second delay before the VRML scene is displayed. With the MPEG-4 scene there will be slightly more than a 3 second delay before we can begin to display the scene. The BIFS-Anim approach offers considerable compression gains when compared to VRML. The gains in compression are similar to what others have achieved [19].

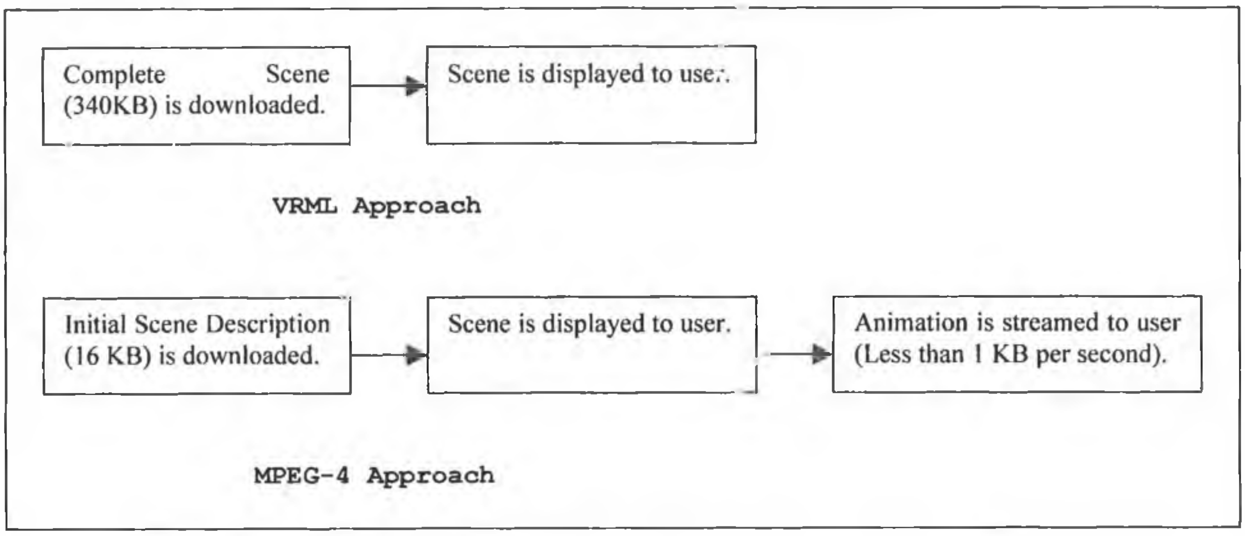

Complete Scene (340KB) is downloaded.
Scene is displayed to user.
VRML Approach
Initial Scene Description (16 KB) is downloaded.
Scene is displayed to user.
Animation is streamed to user (Less than 1 KB per second).
MPEG-4 Approach

Figure 19: Comparison of VRML and MPEG-4 approaches to Animation

The "figure walking" scene compresses better and shows a greater difference between using predictive and Intra frames (Table 13). This is due to the distribution of data in both scenes. Each of the 5 shapes in the "simple shape" scene are moving erratically about the screen and at each frame the x, y and z values for its position are changing. The range of values is higher for the "simple scene" and so the number of quantization levels and Arithmetic Encoding levels will be larger. The change to the fields in the "figure walking" scene happens in a more linear way. For many frames only one value out of 3 will be required to change. Also in the "figure walking" scene nodes are skipped for some frames (`mask.isActive`).

	Predictive	**Intra**
"Simple Shapes"	1/23	1/19
"Figure Walking"	1/53	1/29

Table13 Size of Animation Frame bitstream compared to Interpolator data

It should be noted that the 2 approaches to animating a scene are fundamentally different. BIFS-Anim is like video where every frame will be encoded, while an interpolator node defines a linear function. There are definitely times when using interpolators will be the more efficient method. If there is a small amount of keyframe times and values in relation to the cycle interval then it may be more efficient to use an interpolator. For example if the cycle interval in figure 15 was changed to 1,000 seconds, then representing that animation as an interpolator would be efficient. If that animation was transmitted as a BIFS-Anim stream then that BIFS-Anim stream would contain values for every frame of the animation.

7 Conclusions

This document has largely concentrated on the technical aspects ofMPEG-4, VRML and BIFS-Anim. Chapter 2 gave an overview of the MPEG-4 standard, Chapter 3 continued this overview but looked in detail at MPEG-4 scene description and the improvement it offered over VRML. Chapter 4 looked in detail at one protocol of the scene description language, namely BIFS-Anim. Chapter 5 described a tool for encoding BIFS-Anim and gave some results from using that tool. Finally Chapter 6 gave some examples of real world applications using BIFS-Anim and the issues surrounding online multi-user communities was examined.

This chapter will take a critical look will be taken at BIFS-Anim and its place in the spectrum web 3D Internet technology. The term web 3D is used for any technology that offers a means of displaying 3D content over the Internet.

.Further work is needed to be done on this topic, mainly in the areas of the type of database technology need to store a persistent 3D world.

7.1 *Future of BIFS-Anim*

For BIFS-Anim to become a common technology 3 things are needed:

- Content creation tools
- MPEG-4 players
- MPEG-4 Internet standard.

7.1.1 Internet deliver layer

The transport multiplex is not defined by MPEG-4 as there are many possible deliver protocols that may be used to transmit MPEG-4 content. The Internet is one of the more important delivery protocols and the Internet standardisation body (IETF) is currently working on a standard for streaming of MPEG-4 using Real Time Protocol [33], [34].

7.1.2 Content Creation

There are two main problems with creating VRML content. The first, which is common to BIFS, is that 3D content is difficult to create. Creating simple scenes using a text editor was asy, but to create anything complex proved to be time consuming and required some artistic talent. The graphical 3D creation tool used to create the scenes for the test data was feature rich but difficult to use.

MPEG-4 defines how to decode a bitstream. There is no mention about how that bitstream is to be created. This is left up to the developers of the encoding tools. The developers of the encoding tool will need to decide how to encode content into an efficient MPEG-4 bitstream. With MPEG-4 there is another problem, which is how to create the content in the first place. For example, tools are needed to extract objects from video before they can be encoded as MPEG-4 shape coded video, or tools are needed to create 3D objects and animations before these 3D objects can be encoded. Once the individual objects are created then tools are needed that will allow content creators to assemble the individual objects into a complete scene. One of the goals of MPEG-4 was to allow the reusability of objects, but without useful tools to access and create scenes made of objects this goal will not be realised.

To create the animated figure used for the test data, the professional tool 3D Studio Max was used. The scene was saved as VRML and then the VRML was converted to BIFS. 3D Studio Max has a plug-in architecture allowing developers to provide extra functionality to the product. There are a number of companies that exist to provide plug-in tools for Studio Max. The plug-in architecture allows you to take the internal Studio Max representation of a 3D model and convert that representation to BIFS and export as a BIFS elementary stream. This would then provide a graphical tool that many professional animators are proficient in for creating 3D MPEG-4 content. The problem with this approach is that Studio Max is a professional 3D creation tool it is difficult to use and it is also expensive.

The tools available for creating VRML content fall into 2 categories; tools for professionals and tools for amateurs. The are a number of very good professional VRML creation tools of which 3D Studio Max is one example. Like Studio Max they tend to be expensive and very difficult to use. The are very few low end VRML creation tools [16] and this is a big problem. Any tools designed for amateurs must provide ease of use for people coming from a non artistic background. The low end VRML tools could have added extra functionality to export content as BIFS. The BIFS-Anim encoder could be included as part of a content creation tool, taking the internal 3D representation of the animation as input.

Is the content more important than the technology? Compelling content is what matters most, and does not matter what tools are used to create that content or what tools are used to transport the content. Recently the largest corporate merger in history took place between America Online and Time Warner [5]. This merger creates a company with a large Internet user base and expertise in content delivery over the Internet from AOL and the expertise in content creation from Time Warner. This merger is an indication of how important quality content will become in the future of the Internet.

7.1.3 Players

Without MPEG-4 players, MPEG-4 content can't be played. Currently there are no commercial MPEG-4 players in existence. The feature set of an MPEG-4 player will depend on the environment in which it is used: A player used in a television broadcast will need different features to a player designed to play 3D animations over the Internet. Here we will look at the needs of 3D MPEG-4 content over the Internet.

For BIFS-Anim the following is needed from an MPEG-4 player.

- The rendering needs to be of high quality
- It needs to be free
- The player should work as an attachment to an Internet browser

Providing a fast efficient 3D Rendering engine is not a trivial task. If a future MPEG-4 player has poor quality rendering then the whole technology will look bad. A comparison of the Cosmo VRML player and the Microsoft VRML player shows that the Microsoft player produces poor results even when the scene consists of simple objects and lighting [14]. The loss of smooth colour gradients and accurate lighting completely destroys the sense of object depth, and therefore of the perception of 3D space using the Microsoft player. The amount of effort required to produce a good rendered image is large, and even companies with a lot of resources don't always get it right.

The cost of developing a professional MPEG-4 player will be very large and for that player to reach a large audience it must be given away free. It is unlikely that consumers will pay money for players when they are used to getting them for free. Netscape Communicator and Internet Explorer are both free as are most VRML browsers and Real Player. I think it is correct to say that the consumer will not pay for an MPEG-4 player. Not only will the player need to be free but it is also possible that the creators of the player will need to pay royalties to the companies that own patents on MPEG-4 technology. The company that produced the best VRML player (Cosmo) went out of business.

7.2 Future of Web 3D

There are a number of companies providing web 3D solutions [35]. Most of these are proprietary. These companies typically provide a free player for their technology and then provide the content creation tools at a cost. This model is not open to MPEG-4 as the standard is open and any company can provide a content creation tool.

The number of companies providing web 3D solutions suggests that this is an important technology. I confidently predict that 3D technology over the Internet will become a “killer app”. Whether this solution is MPEG-4 or another technology does not really matter. This thesis has shown the advantages of such a technology and any successful 3D web technology will surely provide a feature set as rich as MPEG-4 BIFS.

7.3 References

[1] Olivier Avaro et al, "Mpeg-4 Systems: Overview", Signal Processing: Image Communication, 1999, pages 6,7,10,12.

[2] Karlheinz Brandenburg, Oliver Kunz and Akihiko Sugiyama, "MPEG-4 Natural Audio Coding", Signal Processing: Image Communication, 1999, pages 1-2, 10.

[3] Bell G., Parisi, A., Pesce, M. 1995. "The Virtual Reality Modeling Language, Version 1.0 Specification"
http://vrml.wired.com/vrml.tech/vrml10-3.html.

[4] Bell, G., Carey, R., Marrin, C. 1996. The Virtual Reality Modeling Language, Version 2.0 Specification, ISO/IEC CD 14772.

[5] Joe Carroll, "Time Warner and Net firm in world's biggest merger", Irish Times, Tuesday, January 11, 2000.

[6] Bob Crispen, "Cyberspace", VRMLWorks, May 1999.

[7] Leonard Daly and Laurel Daly, "Broadband and Internet2: Are We Ready?" www.realism.com/e3d/.

[8] Bruce Damer, "Multi-User VRML Enviroments", VRML Site Magazine, April 97, www.vrmlsite.com/apr97/a.cgi/spot2.html.

[9] Touradj Ebrahimi and Caspar Horne, "MPEG-4 Natural Video Coding an Overview", Signal Processing: Image Communication, 1999, pages 6-12.

[10] William Gibson, "Neuromancer", Ace books July 1994 (ISBN:0441000681).

[11] C. Herpela and A. Eleftheriadis, "MPEG-4 Systems: Elementary Stream Management", Signal Processing: Image Communication, 1999, Pages 1,2,11.

[12] Tiby Kantrowitz, "VRML and Hollywood?", VRML Site Magazine, March 97, www.vrmlsite.com/mar97/a.cgi/spot4.html.

[13] Rob Koenen, "MPEG-4 Multimedia for our time", IEEE Spectrum February 1999 Volume 36 Number 2, page 3.

[14] Robert Polevoi, VRML--Embers in the Ashes - Part 1, http://www.webreference.com/3d.

[15] Dr. Tim Regan, "Taking Living Worlds Into Peoples Living Rooms", VRML 98 Symposium Monterey California.

[16] Sandy Ressler, "Musings on the Future of Web3D", http://web3d.about.com/compute/web3d/library/weekly/aa010500a.htm.

[17] Eric D. Scheirer, Youngjik Lee and Jae-Woo Yang, "Synthetic and SNHC Audio in MPEG-4", Signal Processing: Image Communication, 1999, pages 1,4,6.

[18] J. Signès, Y. Fisher, A. Eleftheriadis, "MPEG-4's Binary Format for Scene Description", Signal Processing: Image Communication, 1999, pages 4,17,18.

[19] J. Signès, "Binary format for scene (BIFS): combining MPEG-4 media to build rich multimedia services. [Conference Paper] SPIE-Int. Soc. Opt. Eng. Proceedings of Spie - the International Society for Optical Engineering, vol 3653, pt. 1-2, 1998, pages 10-12.

[20] Jeff Sonstein, "Pratical Applications for VRML 2.0" - VRML Site Magazine August 1996.

[21] Misty West, "The Territory is the Map: Approaching Storytelling in 3D space", VRML Site Magazine, April 97, www.vrmlsite.com/apr97/a.cgi/spot1.html

[22] MPEG-4 Requirements, ISO/IEC JTC1/SC29/WG11 N2323.

[23] MPEG-1 Video Group, "Information Technology - Coding of Moving Pictures and Associated Audio for Digital Storage Media up to about 1.5 Mbit/s: Part 2 - Video," ISO/IEC 11172-2, International Standard, 1993.

[24] MPEG-2 Video Group, "Information Technology - Generic Coding of Moving Pictures and Associated Audio: Part 2 - Video," ISO/IEC 13818-2, International Standard, 1995.

[25] Coding of Audio-Visual Objects: Visual, ISO/IEC 14496-2 Final Draft International Standard, ISO/IEC JTC1/SC29/WG11 N2502, December 1998.

[26] Coding of Audio-Visual Objects: Audio, ISO/IEC 14496-3 Final Draft International Standard, ISO/IEC JTC1/SC29/WG11 N2503, December 1998.

[27] Coding of Audio-Visual Objects: Systems, ISO/IEC 14496-1 Final Draft International Standard, ISO/IEC JTC1/SC29/WG11 N2501, December 1998, pages 19-21, 29-32, 86-104, 131-136, 218-226.

[28] Overview of the MPEG-4 Standard, Final Draft March 1999, ISO/IEC JTC1/SC29/WG11 N2725.

[29] Flavour Web Site, http://www.ee.columbia.edu/flavour

[30] Coding of Audio-Visual Objects: Reference Software, ISO/IEC 14496-5 Final Draft International Standard, ISO/IEC JTC1/SC29/WG11 N2505, December 1998.

[31] API's for Systems Software Implementation ISO/IEC JTC1/SC29WG11 M3111, Contribution for San Jose, California, January 1998.

[32] Orads Web site, http://www.orad.co.il.

[33] Internet Draft "drsft-ietf-avt-rtp-mpef4-01.txt" RTP Payload for MPEG-4 Streams.

[34] Internet Draft "draft-guillemot-genrtp-01.txt" RTP Payload Format for MPEG-4 with Scaleable & Flexible Error Resiliency.

[35]http://web3d.about.com/compute/web3d/library/blw3dcomp.htm.

Annex A FieldCodingTable Data Structure

This data structure contains parameters relating to the quantization of the field. It is created from the field's entry in the relevant node coding table.

```
Class
FieldCodingTable {
```

`float floatMin[];`	The minimum default bounds for fields of type SFFloat, SFVec2f and SFVec3f. These values are obtained from the "[m, M]" column of the node coding table.
`float floatMax[];`	The minimum default bounds for fields of type SFFloat, SFVec2f and SFVec3f. These values are obtained from the "[m, M]" column of the node coding table.
`float intMin[];`	The minimum default bounds for fields of type SFInt32. These values are obtained from the "[m, M]" column of the node coding table.
`float intMax[];`	The minimum default bounds for fields of type SFInt32. These values are obtained from the "[m, M]" column of the node coding table.
`int defaultNbBits;`	The number of bits used by default for each field. Only used when the quantization category of the field is 13. For quantization category 13, the number of bits used for coding is also specified in the node coding (e.g "13 16" in the node coding table means category 13 with 16 bits).

```
}
```

Annex B AnimFieldQP Data Structure

This data structure contains the necessary quantization parameters and information for the animation of a field. It is updated throughout the BIFS-Anim session.

```
class AnimFieldQP {
```

`int animType;`	The animation method for the field. This is given by the "A column of the node coding table for each node.
`boolean useDefault;`	If this bit is set to TRUE, then the bounds used in intra mode are those specified in the "[m, M]" column of the node coding table. The default value is FALSE.
`boolean isTotal;`	If the field is a multiple field and if this boolean is set t TRUE, all the components of the multiple field are animated.
`int numElement;`	The number of elements being animated in the field. This is 1 for all single fields, and equal to or greater than 1 fo multiple fields.
`int indexList[];`	If the field is a multiple field and if `isTotal` is false, this is the list of the indices of the animated `SFFields`. Fo instance, if the field is an `MFField` with elements 3,4 and 7 being animated, the valuse of `indexList` will be {3,4,7}
`float[] Imin;`	The minimum values for bounds of the field in intra mode This value is obtained from the "[m, M]" column of the node coding table (if `useDefault` is TRUE), the `InitialAnimQP` (if `useDefault` is FALSE and the las intra did not hold any new `AnimQP`), or the `AnimQP`.
`float[] Imax;`	The maximum values for bounds of the field in intra mode This value is obtained from the "[m, M]" column of the semantics table (if `useDefault` is TRUE), the `InitialAnimQP` (if `useDefault` is FALSE and if the las intra did not hold any new `AnimQP`), or the `AnimQP`.
`int[] IminInt;`	The minimum value for bounds of variations of intege fields in intra mode. This value is obtained from the `InitialAnimQP` (if the last intra did not hold any new `AnimQP`) or `AnimQP` structure.
`int[] Pmin;`	The minimum value for bounds of variations of the field i predictive mode. This value is obtained from the `InitialAnimQP` (if the last intra did not hold any new `AnimQP`) or `AnimQP`.
`int INbBits;`	The number of bits used in intra mode for the field. This value is obtained from the `InitialAnimQP` or `AnimQP`.
`int PNbBits;`	The number of bits used in predictive mode for the field This value is obtained from the `InitialAnimQP` (if the last intra did not hold any new `AnimQP`) or `AnimQP` structure.

```
}
```

Annex C Adaptive Arithmetic Decoder for BIFS-Anim

The follwing procedures, in C code, describe the adaptative arithmetic deoder used in a BIFS-Anim session. The model is specified through the array `int* cumul_freq[ ]`. The decoded symbol is returned through its index in the model.

First, the following integers are defined :

```
static long bottom=0, q1=2^14, q2=2^15, q3=3*2^14,
top=2^16;
```

The decoder is initialized to start decoding an arithmetic coded bitstream by calling the following procedure.

```
static long low, high, code_value, bit, length, sacindex,
cum, zerorun=0;

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++) { //16 bits are read
ahead
    bit_out_psc_layer();
    code_value = 2 * code_value + bit;
  }
  used_bits = 0;
}
```

In the BIFS-Anim decoding process, a symbol is decoded using a model specified through the array `cumul_freq[]` and by calling the following procedure.

```
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);
}
```

The model is specified in the array `cumul_freq[ ]`. It is reset with the following procedure.

```
void model_reset(int nbBits)
{
  int nbValues = (1<<nbBits)+1;
  int* cumul_freq = (int*) malloc(sizeof(int)*nbValues);
  int i;
  for (i=1;i<=nbValues;i++) {
    cumul_freq[i] = nbValues-i;
}
```

The model is updated when the value `symbol` is read with the following procedure.

```
void update_model(int cumul_freq[ ], int symbol) {
  if (cumul_freq[0] == q1) { //The model is rescaled to
avoid overflow
    int cum = 0;
    for(int i=nb_of_symbols-1; i>=0; i--) {
      cum += (cumul_freq[i]-cumul_freq[i+1]+1)/2;
      cumul_freq[i] = cum;
    }
    cumul_freq[nb_of_symbols] = 0;
  }
```

```
    while(symbol>0)
      cumul_freq[symbol--] ++;

}
```

Annex D Example Files used by Im1 BIFS Encoder and Mux Tools

This code is an example of a file consumed by the BIFS Encoder. The scene description information is at the top of the file and is similar to VRML. The textual description of the Object Descriptors is in the second half of the example.

```
Group {
DEF location Transform {
          Translation    1 1 1
          Children [
               Shape {
                    ggeometry Cylinder {
                         radius 5
                         height 6
                    }
               }
          ]
     }
     DEF Anim AnimationStream {
              loop FALSE
              url 10
           }
}
UPDATE OD [
           {
              objectDescriptorID        10
              es_descriptor {
                es_Number 1
                url earthStream.mp4
                 streamData 5
                 decConfigDescr {
                    streamType 4       // AnimationStream
                bufferSizeDB 1000
                bifsDecoderConfig {
                   nodeIDbits 10
                   routeIDbits 10
                   isCommandStream FALSE
                   animMask earthStream.mask
                }
             }
             alConfigDescr {
                useTimeStampsFlag TRUE
                timeStampLength 10
             }
          }
       }
```

This code is an example of a file consumed by the Mux.

```
// Initial OD:
{
   objectDescriptorID    0
   es_descriptor [
           {
              es_Number 1
              fileName earth.od
              decConfigDescr {
              streamType 2         // OD Stream
              bufferSizeDB 1000
              }
              alConfigDescr {
              useAccessUnitStartFlag TRUE
              useAccessUnitEndFlag TRUE
              useRandomAccessPointFlag TRUE
              useTimeStampsFlag TRUE
```

```
                timeStampResolution 1000
                timeStampLength 14
                }
             }
             {
                es_Number 2
                fileName earth.bif
                decConfigDescr {
                streamType 4        // BIFS Stream
                bufferSizeDB 1000
                bifsDecoderConfig {
                   nodeIDbits 10
                   routeIDbits 10
                   isCommandStream TRUE
                   pixelMetric TRUE
                }
                }
                alConfigDescr {
                useAccessUnitStartFlag TRUE
                useAccessUnitEndFlag TRUE
                useRandomAccessPointFlag TRUE
                useTimeStampsFlag TRUE
                timeStampResolution 100
                timeStampLength 14
                }
             }
            ]
}

// Media streams ODs
{
   objectDescriptorID    10
   es_descriptor {
      es_Number 1
```

```
            fileName earthStream.frames
            streamData 5
            decConfigDescr {
            streamType 8        // AnimationStream
            bufferSizeDB 1000
            }
            alConfigDescr {
            useTimeStampsFlag TRUE
            //timeStampResolution 1000 // clock ticks/s
            timeStampLength 10
            }
        }
    }
```

Annex E Syntactic Description Language

E.1 Introduction

The elementary constructs are described first, followed by the composite syntactic constructs, and arithmetic and logical expressions. Finally, syntactic control flow and built-in functions are addressed. Syntactic flow control is needed to take into account context-sensitive data. Several examples are used to clarify the structure.

E.2 Elementary Data Types

The SDL uses the following elementary data types:

1. Constant-length direct representation bit fields or Fixed Length Codes — FLCs. These describe the encoded value exactly as it is to be used by the appropriate decoding process.

2. Variable length direct representation bit fields, or parametric FLCs. These are FLCs for which the actual length is determined by the context of the bitstream (e.g., the value of another parameter).

3. Constant-length indirect representation bit fields. These require an extra lookup into an appropriate table or variable to obtain the desired value or set of values.

4. Variable-length indirect representation bit fields (e.g., Huffman codes).

These elementary data types are described in more detail in the clauses to follow immediately.

All quantities shall be represented in the bitstream with the most significant byte first, and also with the most significant bit first.

E.2.1 Constant-Length Direct Representation Bit Fields

Constant-length direct representation bit fields shall be represented as:

Rule E.1: Elementary Data Types
[`aligned`] ***`type`***[(*length*)] *element_name* [= *value*]; // C++-style comments allowed

The ***`type`*** may be any of the following: **`int`** for signed integer, **`unsigned int`** for unsigned integer, **`double`** for floating point, and **`bit`** for raw binary data. The *length* attribute indicates the length of the element in bits, as it is actually stored in the bitstream. Note that a data ***`type`*** equal to **`double`** shall only use 32 or 64 bit lengths. The *value* attribute shall be present only when the value is fixed (e.g., start codes or object IDs), and it may also indicate a range of values (i.e., '0x01..0xAF'). The ***`type`*** and the optional *length* attributes are always present, except if the data is non-parsable, i.e., it is not included in the bitstream. The keyword **`aligned`** indicates that the data is aligned on a byte boundary. As an example, a start code would be represented as:

```
aligned bit(32) picture_start_code=0x00000100;
```

An optional numeric modifier, as in **aligned(32)**, may be used to signify alignment on other than byte boundary. Allowed values are 8, 16, 32, 64, and 128. Any skipped bits due to alignment shall have the value '0'. An entity such as temporal reference would be represented as:

```
unsigned int(5) temporal_reference;
```

where **unsigned int(5)** indicates that the element shall be interpreted as a 5-bit unsigned integer. By default, data shall be represented with the most significant bit first, and the most significant byte first.

The value of parsable variables with declarations that fall outside the flow of declarations (see E.5) shall be set to 0.

Constants shall be defined using the keyword **const**.

EXAMPLE —

```
const int SOME_VALUE=255; // non-parsable constant
const bit(3) BIT_PATTERN=1;  // this is equivalent to the bit string "001"
```

To designate binary values, the **0b** prefix shall be used, similar to the **0x** prefix for hexadecimal numbers. A period ('.') may be optionally placed after every four digits for readability. Hence 0x0F is equivalent to 0b0000.1111.

In several instances, it may be desirable to examine the immediately following bits in the bitstream, without actually consuming these bits. To support this behavior, a '*' character shall be placed after the parse size parentheses to modify the parse size semantics.

Rule E.2: Look-ahead parsing
  [**aligned**] ***type*** **(***length***)** * *element_name*;

For example, the value of next 32 bits in the bitstream can be checked to be an unsigned integer without advancing the current position in the bitstream using the following representation:

```
aligned unsigned int (32)* next_code;
```

E.2.2 Variable Length Direct Representation Bit Fields

This case is covered by Rule E.1, by allowing the *length* attribute to be a variable included in the bitstream, a non-parsable variable, or an expression involving such variables.

EXAMPLE —

```
unsigned int(3) precision;
int(precision) DC;
```

E.2.3 Constant-Length Indirect Representation Bit Fields

Indirect representation indicates that the actual value of the element at hand is indirectly specified by the bitstream through the use of a table or map. In other words, the value extracted from the bitstream is an index to a table from which the final desired value is extracted. This indirection may be expressed by defining the map itself:

Rule E.3: Maps
```
    map MapName (output_type) {
       index, {value_1, … value_M},
       …
    }
```

These tables are used to translate or map bits from the bitstream into a set of one or more values. The input type of a **map** (the *index* specified in the first column) shall always be `bit`. The *output_type* entry shall be either a predefined type or a defined class (classes are defined in E.3.1). The **map** is defined as a set of pairs of such indices and values. Keys are binary string constants while values are *output_type* constants. Values shall be specified as aggregates surrounded by curly braces, similar to C or C++ structures.

EXAMPLE —

```
class YUVblocks {// classes are fully defined later on
   int Yblocks;
   int Ublocks;
   int Vblocks;
}

// a table that relates the chroma format with the number of blocks
// per signal component
map blocks_per_component (YUVblocks) {
   0b00, {4, 1, 1}, // 4:2:0
   0b01, {4, 2, 2}, // 4:2:2
   0b10, {4, 4, 4}  // 4:4:4
}
```

The next rule describes the use of such a **map**.

Rule E.4: Mapped Data Types
type **(***MapName***)** *name*;

The ***type*** of the variable shall be identical to the ***type*** returned from the **map**.

EXAMPLE —

```
YUVblocks(blocks_per_component) chroma_format;
```

Using the above declaration, a particular value of the **map** may be accessed using the construct: `chroma_format.Ublocks`.

E.2.4 Variable Length Indirect Representation Bit Fields

For a variable length element utilizing a Huffman or variable length code table, an identical specification to the fixed length case shall be used:

```
class val {
   unsigned int foo;
   int bar;
}

map sample_vlc_map (val)  {
   0b0000.001,     {0, 5},
   0b0000.0001, {1, -14}
}
```

The only difference is that the indices of the **map** are now of variable length. The variable-length codewords are (as before) binary strings, expressed by default in '0b' or '0x' format, optionally using the period ('.') every four digits for readability.

Very often, variable length code tables are partially defined. Due to the large number of possible entries, it may be inefficient to keep using variable length codewords for all possible values. This necessitates the use of escape codes, that signal the subsequent use of a fixed-length (or even variable length) representation. To allow for such exceptions, parsable type declarations are allowed for **map** values.

EXAMPLE — This example uses the class type 'val' as defined above.

```
map sample_map_with_esc (val)  {
   0b0000.001,      {0, 5},
   0b0000.0001, {1, -14},
   0b0000.0000.1,  {5, int(32)},
   0b0000.0000.0,  {0, -20}
}
```

When the codeword 0b0000.0000.1 is encountered in the bitstream, then the value '5' is assigned to the first element (`val.foo`). The following 32 bits are parsed and assigned as the value of the second element (`val.bar`). Note that, in case more than one element utilizes a parsable type declaration, the order is significant and is the order in which elements are parsed. In addition, the type within the **map** declaration shall match the type used in the class declaration associated with the **map**'s return type.

E.3 Composite Data Types

E.3.1 Classes

Classes are the mechanism with which definitions of composite types or objects is performed. Their definition is as follows.

Rule C.1: Classes

[**aligned**] [**abstract**] [**expandable**[**(***maxClassSize***)**]] **class** *object_name*
[**extends** *parent_class*] [: **bit(***length***)** [*id_name*=] *object_id* | *id_range*] {
[*element*; …] // zero or more elements
}

The different elements within the curly braces are the definitions of the elementary bitstream components discussed in 12.2 or control flow elements that will be discussed in a subsequent subclause.

The optional keyword **extends** specifies that the **class** is "derived" from another **class**. Derivation implies that all information present in the base **class** is also present in the derived **class**, and that, in the bitstream, all such information *precedes* any additional bitstream syntax declarations specified in the new **class**.

The optional attribute *id_name* allows to assign an *object_id*, and, if present, is the key demultiplexing entity which allows differentiation between base and derived objects. It is also possible to have a range of possible values: the *id_range* is specified as *start_id* .. *end_id*, inclusive of both bounds.

If the attribute id_name is used, a derived **class** may appear at any point in the bitstream where its base **class** is specified in the syntax. This allows to express polymorphism in the SDL syntax description. The actual **class** to be parsed is determined as follows:

The base **class** declaration shall assign a constant value or range of values to *object_id*.

Each derived **class** declaration shall assign a constant value or ranges of values to *object_id*. This value or set of values shall correspond to legal *object_id* value(s) for the base **class**.

NOTE 1 — Derivation of classes is possible even when object_ids are not used. However, in that case derived classes may not replace their base **class** in the bitstream.

NOTE 2 — Derived classes may use the same *object_id* value as the base **class**. In that case classes can only be discriminated through context information.

EXAMPLE —

```
class slice: aligned bit(32) slice_start_code=0x00000101 .. 0x000001AF {
   // here we get vertical_size_extension, if present
   if (scalable_mode==DATA_PARTITIONING) {
      unsigned int(7) priority_breakpoint;
   }
   ...
}

class foo {
   int(3) a;
   ...
}

class bar extends foo {
   int(5) b; // this b is preceded by the 3 bits of a
   int(10) c;
   ...
}
```

The order of declaration of the bitstream components is important: it is the same order in which the elements appear in the bitstream. In the above examples, `bar.b` immediately precedes `bar.c` in the bitstream.

Objects may also be encapsulated within other objects. In this case, the *element* in Rule C.1 is an object itself.

E.3.2 Abstract Classes

When the **`abstract`** keyword is used in the **`class`** declaration, it indicates that only derived classes of this **`class`** shall be present in the bitstream. This implies that the derived classes may use the entire range of IDs available. The declaration of the abstract **`class`** requires a declaration of an ID, with the value 0.

EXAMPLE —

```
abstract class Foo : bit(1) id=0 { // the value 0 is not really used
   ...
}

// derived classes are free to use the entire range of IDs
class Foo0 extends Foo : bit(1) id=0 {
   ...
}

class Foo1 extends Foo : bit(1) id=1 {
   ...
}

class Example {
   Foo f;  // can only be Foo0 or Foo1, not Foo
}
```

E.3.3 Expandable classes

When the **`expandable`** keyword is used in the **`class`** declaration, it indicates that the **`class`** may contain implicit arrays or undefined trailing data, called the "expansion". In this case the **`class`** encodes its own size in bytes explicitly. This may be used for classes that require future compatible extension or that may include private data. A legacy device is able to decode an expandable **`class`** up to the last parsable variable that has been defined for a given revision of this **`class`**. Using the size information, the parser shall skip the **`class`** data following the last known syntax element. Anywhere in the syntax where a set of expandable classes with *object_id* is expected it is permissible to intersperse expandable classes with unknown *object_id* values. These classes shall be skipped, using the size information.

The size encoding precedes any parsable variables of the **`class`**. If the **`class`** has an *object_id*, the encoding of the *object_id* precedes the size encoding. The size information shall not include the number of bytes needed for the size and the *object_id* encoding. Instances of expandable classes shall always have a size corresponding to an integer number of bytes. The size information is accessible within the class as class instance variable `sizeOfInstance`.

If the **`expandable`** keyword has a *maxClassSize* attribute, then this indicates the maximum permissible size of this **`class`** in bytes, including any expansion.

The length encoding is itself defined in SDL as follows:

```
int sizeOfInstance = 0;
bit(1) nextByte;
bit(7) sizeOfInstance;
while(nextByte) {
   bit(1) nextByte;
   bit(7) sizeByte;
   sizeOfInstance = sizeOfInstance<<7 | sizeByte;
}
```

E.3.4 Parameter types

A parameter type defines a **`class`** with parameters. This is to address cases where the data structure of the **`class`** depends on variables of one or more other objects. Since SDL follows a declarative approach, references to other objects, in such cases, cannot be performed directly (none is instantiated). Parameter types provide placeholders for such references, in the same way as the arguments in a C function declaration. The syntax of a **`class`** definition with parameters is as follows.

Rule C.2: Class Parameter Types

[**`aligned`**] [**`abstract`**] **`class`** *object_name* [**(***parameter list***)**] [**`extends`** *parent_class*]
[: **`bit`(***length***)** [*id_name*=]
object_id | *id_range*] **{**
[*element*; …] // zero or more elements
}

The parameter list is a list of ***`type`*** names and variable name pairs separated by commas. Any element of the bitstream, or value derived from the bitstream with a variable-length codeword, or a constant, can be passed as a parameter.

A **`class`** that uses parameter types is dependent on the objects in its parameter list, whether **`class`** objects or simple variables. When instantiating such a **`class`** into an object, the parameters have to be instantiated objects of their corresponding classes or types.

EXAMPLE —

```
class A {
   // class body
   ...
   unsigned int(4) format;
}

class B (A a, int i) {     // B uses parameter types
   unsigned int(i) bar;
   ...
   if( a.format == SOME_FORMAT ) {
      ...
   }
   ...
}

class C {
   int(2) i;
   A a;
   B foo( a, I); // instantiated parameters are required
}
```

E.3.5 Arrays

Arrays are defined in a similar way as in C/C++, i.e., using square brackets. Their length, however, can depend on run-time parameters such as other bitstream values or expressions that involve such values. The array declaration is applicable to both elementary as well as composite objects.

Rule A.1: Arrays
    typespec *name* [*length*];

---

typespec is a ***type*** specification (including bitstream representation information, e.g. '**int**(2)'). The attribute *name* is the name of the array, and *length* is its length.

EXAMPLE —

```
unsigned int(4) a[5];
int(10) b;
int(2) c[b];
```

Here 'a' is an array of 5 elements, each of which is represented using 4 bits in the bitstream and interpreted as an unsigned integer. In the case of 'c', its length depends on the actual value of 'b'. Multi-dimensional arrays are allowed as well. The parsing order from the bitstream corresponds to scanning the array by incrementing first the right-most index of the array, then the second, and so on .

E.3.6 Partial Arrays

In several situations, it is desirable to load the values of an array one by one, in order to check, for example, a terminating or other condition. For this purpose, an extended array declaration is allowed in which individual elements of the array may be accessed.

Rule A.2: Partial Arrays
    typespec *name*[[*index*]];

Here *index* is the element of the array that is defined. Several such partial definitions may be given, but they shall all agree on the ***type*** specification. This notation is also valid for multidimensional arrays.

EXAMPLE —

```
int(4) a[[3]][[5]];
```

indicates the element a(5, 3) of the array (the element in the 6th row and the 4th column), while

```
int(4) a[3][[5]];
```

indicates the ent.. e sixth column of the array, and

```
int(4) a[[3]][5];
```

indicates the entire fourth row of the array, with a length of 5 elements.

NOTE — **a[5]** means that the array has five elements, whereas **a[[5]]** implies that there are at least six.

E.3.7 Implicit Arrays

When a series of polymorphic classes is present in the bitstream, it may be represented as an array of the same type as that of the base **class**. Let us assume that a set of polymorphic classes is defined, derived from the base **class** Foo (may or may not be abstract):

```
class Foo : int(16) id = 0 {
   ...
}
```

For an array of such objects, it is possible to implicitly determine the length by examining the validity of the **class** ID. Objects are inserted in the array as long as the ID can be properly resolved to one of the IDs defined in the base (if not abstract) or its derived classes. This behavior is indicated by an array declaration without a length specification.

EXAMPLE 1 —

```
class Example {
   Foo f[];   // length implicitly obtained via ID resolution
}
```

To limit the minimum and maximum length of the array, a range specification may be inserted in the specification of the length.

EXAMPLE 2 —

```
class Example {
   Foo f[1 .. 255];   // at least 1, at most 255 elements
}
```

In this example, 'f' may have at least 1 and at most 255 elements.

E.4 Arithmetic and Logical Expressions

All standard arithmetic and logical operators of C++ are allowed, including their precedence rules.

In order to accommodate complex syntactic constructs, in which context information cannot be directly obtained from the bitstream but only as a result of a non-trivial computation, non-parsable variables are allowed. These are strictly of local scope to the **class** they are defined in. They may be used in expressions and conditions in the same way as bitstream-level variables. In the following example, the number of non-zero elements of an array is computed.

```
unsigned int(6) size;
int(4) array[size];
...
int i; // this is a temporary, non-parsable variable
for (i=0, n=0; i<size; i++) {
   if (array[[i]]!=0)
      n++;
}

int(3) coefficients[n];
// read as many coefficients as there are non-zero elements in array
```

E.5 Syntactic Flow Control

The syntactic flow control provides constructs that allow conditional parsing, depending on context, as well as repetitive parsing. The familiar C/C++ if-then-else construct is used for testing conditions. Similarly to C/C++, zero corresponds to false, and non-zero corresponds to true.

Rule FC.1: Flow Control Using If-Then-Else

```
if (condition) {
  ...
}[else if (condition) {
  ...
}][else {
  ...
}]
```

EXAMPLE 1 —

```
class conditional_object {
   unsigned int(3) foo;
   bit(1) bar_flag;
   if (bar_flag) {
      unsigned int(8) bar;
   }
   unsigned int(32) more_foo;
}
```

Here the presence of the entity 'bar' is determined by the 'bar_flag'.

EXAMPLE 2 —

```
class conditional_object {
   unsigned int(3) foo;
   bit(1) bar_flag;
   if (bar_flag) {
      unsigned int(8) bar;
   } else {
      unsigned int(some_vlc_table) bar;
   }
   unsigned int(32) more_foo;
}
```

Here we allow two different representations for 'bar', depending on the value of 'bar_flag'. We could equally well have another entity instead of the second version (the variable length one) of 'bar' (another object, or another variable). Note that the use of a flag necessitates its declaration before the conditional is encountered. Also, if a variable appears twice (as in the example above), the types shall be identical.

In order to facilitate cascades of if-then-else constructs, the 'switch' statement is also allowed.

Rule FC.2: Flow Control Using Switch

```
switch (condition) {
      [case label1: ...]
      [default:]
}
```

The same category of context-sensitive objects also includes iterative definitions of objects. These simply imply the repetitive use of the same syntax to parse the bitstream, until some condition is met (it is the conditional repetition that implies context, but fixed repetitions are obviously treated the same way). The familiar structures of 'for', 'while', and 'do' loops can be used for this purpose.

Rule FC.3: Flow Control Using For

```
for  (expression1; expression2; expression3) {
        ...
}
```

---

expression1 is executed prior to starting the repetitions. Then *expression2* is evaluated, and if it is non-zero (true) the declarations within the braces are executed, followed by the execution of *expression3*. The process repeats until *expression2* evaluates to zero (false).

Note that it is not allowed to include a variable declaration in *expression1* (in contrast to C++).

Rule FC.4: Flow Control Using Do

```
do {
        ...
} while (condition);
```

Here the block of statements is executed until *condition* evaluates to false. Note that the block will be executed at least once.

Rule FC.5: Flow Control Using While

```
while (condition) {
        ...
}
```

The block is executed zero or more times, as long as *condition* evalutes to non-zero (true).

E.6 Built-Ir Operators

The following built-in operators are defined.

Rule O.1: lengthof() Operator

```
lengthof(variable)
```

This operator returns the length, in bits, of the quantity contained in parentheses. The length is the number of bits that was most recently used to parse the quantity at hand. A return value of 0 means that no bits were parsed for this variable.

E.7 Scoping Rules

All parsable variables have class scope, i.e., they are available as class member variables.

For non-parsable variables, the usual C++/Java scoping rules are followed (a new scope is introduced by curly braces: '{' and '}'). In particular, only variables declared in class scope are considered class member variables, and are thus available in objects of that particular type.