Java3D 기반 MPEG-4 시스템의 DMIF 및 BIFS 파서 구현

최정단, 장병태, 오광만°, 이민석, 곽진석, 전준근
ETRI 가상현실연구부, 넷코덱 기술연구소 MPEG-4 시스템 개발팀

Implementation of DMIF & BIFS Parser in Java3D-based MPEG4 System

J.D. Choi, B.T. Jang, K.M. Oh°, M.S. Lee, J.S. Kwak, J.K. Jeon
Virtual Reality Center, ETRI
MPEG-4 System Development Group of Netcodec co.
E-mail: {jdchoi, jbt}@etri.re.kr, {okman, mslee, jskwak,jgjeon}@Netcodec.com

요 약

인터넷을 통해 멀티미디어 데이터의 접근이 보편화됨에 따라 다양한 형태의 데이터와 사용자 인터렉션이 요구되었고, 또한 유선 및 무선등과 같은 다양한 통신 선로에서 Desktop-PC, PDA, Hand-Held PC등과 같은 다양한 단말기를 통해 멀티미디어 데이터 서비스를 받으려는 사용자의 요구가 증가되고 있다. 따라서 이런 요구를 효율적으로 지원할 수 있는 멀티미디어 시스템에 대한 개발이 요구되었고, 이를 위해 MPEG4 표준이 등장하게 되었다. MPEG-4(ISO/IEC 국제표준 14496)는 오디오, 비디오, 합성 오디오, 그리고 그래픽스 요소(material)를 포함하는 멀티미디어 데이터로 구성된 복잡한 씬(scene)을 구성하고, 이를 통신라인을 통해 사용자와 상호작용이 가능한 멀티미디어 시스템을 정의하는 표준규약을 말한다. 본 논문에서는 Java와 Java3D기반의 MPEG-4 표준 규약에 충실한 MPEG-4 시스템 구현에 대하여 기술한다.

1. 서론

인터넷 및 무선 통신이 보편화 되어 감에 멀티미디어 데이터에 대한 서비스의 요구가 증가되었고 또한 이러한 멀티미디어 데이터를 효율적으로 제작, 관리, 전송 및 플레이 해야 할 필요성이 증가되었다. 이에 부응하여 MPEG-4표준이 개발 되었다. MPEG-4(ISO/IEC 국제표준 14496)는 오디오, 비디오, 합성 오디오, 그리고 그래픽스 요소(material)를 포함하는 멀티미디어 데이터로 구성된 복잡한 환경(scene)을 구성하고, 통신라인을 통해 사용자와 상호작용이 가능한 멀티미디어 시스템을 정의하는 표준 규약을 말한다. MPEG-4 시스템은 이러한 표준 규약을 구현한 멀티미디어 시스템이라 말할 수 있다[1,2,3].

MPEG-4에서 중요한 요소로서 첫째, QoS (Quality of Service)는 다양한 통신 플랫폼이나 단말기에 맞는 서비스를 제공하여 그 응용분야를 광범위하게 하고자 하는 것이다. 따라서 이를 규정하기 위해 다양한 프로파일을 규정하고 있다. 이는 비디오, 오디오, 그래픽스등의 각 매체에 프로파일 레벨을 정의하고 해당 통신 플랫폼이나 단말기에 맞는 압축 방식과 환경(scene) 구성방법을 규정한다[1,2,3,6,7]. 둘째, 사용자 인터렉션을 지원하는데 이는 기존의 멀티미디어 서비스가 주로 단 방향의 매체를 지원하는데 반해서

양방향 통신이 가능하도록 하여 사용자의 욕구가 인터렉티브하게 반영될 수 있도록 한다. 이러한 두 가지 요소를 만족시킬 때 MPEG-4 시스템은 디지털 방송, 게임, 애니메이션등의 광범위한 분야에 응용될 수 있다.

본 논문에서는 위의 요소를 극대화 시키기 위해 MPEG-4 시스템의 구현에 Java를 이용하였다. 주지하는 바와 같이 Java는 플랫폼에 독립적인 언어로서 이식성이 매우 높다. 또한 그래픽스 요소를 위하여 Java3D API를 사용하였다.

2. MPEG-4 시스템

2.1 MEPG-4

MPEG-4는 1993년 7월부터 표준화 작업이 시작되어 1999년 11월에 Committee Draft가 제시되었고, 현재의 version 1은 거의 표준화 작업이 종결되었다. MPEG-4는 기존의 MPEG 표준들을 결합하고 다음과 같은 3가지 요구를 반영하는 새로운 표준이 되었다. 첫째, 저작자의 측면에서 오늘날 디지털 TV나 애니메이션, WWW 등과 같은 개별적인 기술보다는 이들을 통합하여 유연하고 재사용도가 높은 컨텐츠의 제작에 대한 요구이다. 물론 여기에는 기존의 방식보다 컨텐츠에 대한 저작권의 보호와 관리가 용이하게 되어야 한다. 둘째, 네트워크 서비스 제공자의 측면에서 투명한(transparent) 정보의 제공이다. 그래서 다양한 언어를 제공하기 위해서는 단지 관련 신호만 보내면 되도록 하는 것이다. 또한 QoS와 같은 네트워크나 단말기 특성에 따라서 서비스의 품질을 조정하는 방법의 제공이다. 셋째, 사용자의 측면에서 하나의 단말기를 통해 다양한 기능을 접근할 수 있고 또한 컨텐츠와 높은 인터렉션에 대한 요구이다.

위와 같은 요구를 만족시키기 위해서 MPEG-4에서는 Coding, Composition, Multiplex, Interaction에 대한 규약을 기술하고 있다[1,2,3].

2.2 MPEG-4 시스템

그림 1에서와 같이 최하부의 로컬 저장영역이나 통신 선로를 통해 데이터가 입력되고 그 상위의 Delivery Layer에서 데이터의 송수신을 담당한다. Sync Layer에서는 Delivery Layer의 FlexMux를 통해 나온 데이터를 SL packet형태로 만들어 Composition Layer로 건네주게 된다. Composition Layer에서는 Object Descriptor, Scene Description, AV 데이터를 각 스트림별로 분류하고 이를 해당 디코더나 파서를 통해서 장면(scene)을 구성한다.

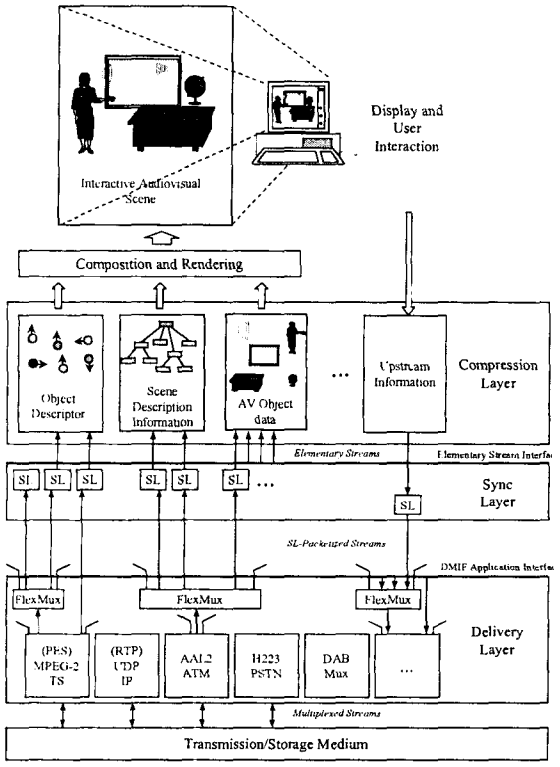

그림 1 MPEG-4 System

그림 1과 같은 MPEG-4 시스템은 크게 DMIF(Delivery Multimedia Integration Framework, Decoder, Composition과 Rendering부분으로 나누어질 수 있다.

2.2.1 DMIF

DMIF는 로컬 디스크나 네트워크상의 멀티미디어 데이터를 읽어 들이는 곳으로 가장 중요한 개념은 Transparency를 보장이다. 즉 그림 1의 Layer에서 볼 때 모든 데이터가 하나의 스트림 형태로 보이도

록 보장하는 것이다. 로컬 파일인 경우 Quick Time 포멧을 채용한 MP4 파일 포멧을 파싱하고 이를 스트림 형태로 만들어 Decoder Buffer에 넣어 주게 된다. 그리고 네트웍을 통해서 전달된 데이터는 네트워크 프로토콜을 통해 데이터를 받고 이를 미디어 타입별로 분류하여 Decoding Buffer에 넣어 상위 Layer에서 Decoding할 수 있도록 한다[4].

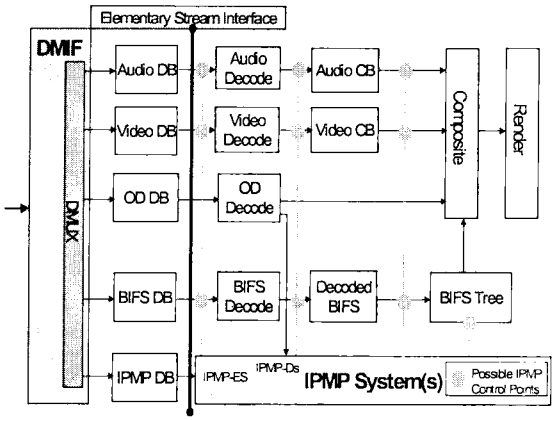

그림 2 DMIF, Decoder and Composition

2.2.2 Decoder

Decoder는 미디어 타입별 디코딩을 하는데 스트리밍을 해야 할 경우 데이터가 지연 되는 경우가 발생하기 때문에 이를 보완하기 위해서 Decoder Buffer를 두고 지연을 완충하여 디코딩을 수행한다. 디코더에는 비디오, 오디오, BIFS, 그리고 Object Descriptor를 위한 디코더가 있다.

2.2.3 Composition 과 Rendering

MPEG-4 시스템에서 Scene은 BIFS(Binary Format for Scene)에 의해서 기술된다. Composition은 BIFS에 의해서 기술된 Scene을 파싱하고 렌더링하는 과정이라 할 수 있는데, 여기에 비디오와 오디오를 결합하는 명령어가 들어있다. 따라서 BIFS를 파싱하고 이를 렌더링하는 과정이 Composition의 과정이라 할 수 있다. BIFS는 VRML와 유사한 형태를 가지며[5] 또한 그림 3과 같이 VRML2.0에 들어있는 노드 이외에 2D Nodes, Audio Nodes, FBA등이 추가되어 전체적으로 114개의 노드를 가진다.

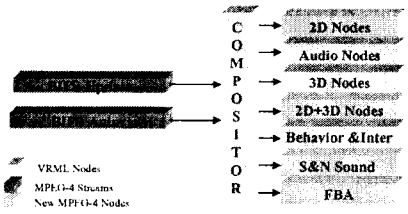

그림 3 BIFS Nodes

3. 구현 및 고찰

본 절에서는 MPEG-4 시스템 구현에 있어서 주로 mp4파일을 파싱하여 미디어 스트림을 뽑아내는 DMIF와 BIFS를 파싱하여 이로부터 Scene을 합성하는 과정을 기술하고 이의 구현 시 고려해야 할 점들에 대해서 기술한다.

3.1 DMIF 구현 및 mp4 파일 파싱

그림 4에서 보는 바와 같이 MP4 파일은 Apple사에서 개발한 QuickTime 포멧을 채용하고 있다. 파일 구조를 대략 살펴보면, Atom과 Object Descriptor로 구성 되어 있다. 모든 정보는 Atom이라 불리는 단위 container에 들어간다. 먼저 데이터의 관련된 정보를 가지는 moov(Movie Atom)와 mdat(Media Data Atom)로 구성되어 있다. moov에는 IOD(Initial Object Descriptor)라는 header 정보가 오는데 이는 mp4 파일 전체를 파싱하는데 필요한 정보들을 가지고 있다. 그 다음에 오는 track은 각 미디어 타입별로 하나의 track이 오게 되며, 그림 4에서는 BIFS, OD, Video, Audio가 올 경우 4개의 track이 존재함을 보여준다.

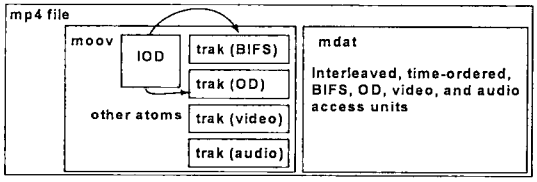

그림 4 mp4 file structure

다음은 mp4 파일을 파싱한 결과를 가지고 Atom 들의 계층관계를 보여준다.

```
Mp4MovieAtom(moov) {
 Mp4MovieHeaderAtom(mvhd) {  }
 Mp4ObjectDescriptorAtom(iods) {
  MP4_IOD {
   Es_Id_IncDescriptor { }
   Es_Id_IncDescriptor { }
  }
 }
 Mp4TrackAtom(trak) {
  Mp4TrackHeaderAtom(tkhd) { }
  Mp4MediaAtom(mdia){
   Mp4MediaHeaderAtom(mdhd) { }
   Mp4MediaInformationAtom(minf) {
    Mp4DataInformationAtom(dinf) {
     Mp4DataReferenceAtom(dref) {
      Mp4DataEntryURLAtom(url) { }
     }
    }
    Mp4SampleTableAtom(stbl) {
     Mp4TimeToSampleAtom(stts) { }
     Mp4SampleDescriptionAtom(stsd) {
      Mp4VisualSampleEntryAtom(mp4v) {
       Mp4EsdAtom(esds) {
        ES_Descriptor {
         DecoderConfigDescriptor {
          DecSpecificInfoDescriptor { }
          I got decoderSpecificInfo
         }
         SLConfigDescriptor { }
        } // end of es_descriptor
       }
      }
     } // end of sample table atom
     Mp4SampleSizeAtom(stsz) { }
     Mp4SampleToChunkAtom(stsc) { }
     Mp4ChunkOffsetAtom(stco) { }
     Mp4CompositionOffsetAtom(ctts) { }
     Mp4SyncSampleAtom(stss) { }
    }
    Mp4VideoMediaHeaderAtom(vmhd) { }
   }
   Mp4HandlerAtom(hdlr) { }
  }
 }
} // end of moov
```

위의 결과는 moov mp4 파일에 필요한 헤더 정보를 읽어들이는 부분이고, 다음에 실제로 미디어 데이터인 mdat가 온다. 기본적으로 mp4에서 모든 데이터는 ES_Descriptor에 실려 오게 된다. 각 track마다 해당 미디어 타입을 디코딩할 때 필요한 정보를 DecoderConfigDescriptor에 실려 보내는데 BIFS 미디어의 경우 이는 이후에 BIFS 파서에서 사용되고, 오디오나 비디오의 경우는 오디오 및 비디오 코덱에서 사용하게 된다.

3.2 BIFS 파서의 구현

VRML이 utf8로 인코딩 된 텍스트 파일인데 비해 BIFS는 binary로 코딩 되어 있다. 따라서 파서의 구현 시 Bit 단위로 데이터를 읽어야 하기 때문에 기존의 Lex나 Yacc를 이용하여 토큰을 분리하는 방식을 사용하기에는 어렵게 된다. 그림 5와 같이 BIFS는 4가지 command를 통해 Scene이 구성되고 update가 이루어진다. 초기 Scene은 SceneReplacementCommand에 의해서 구성된다.

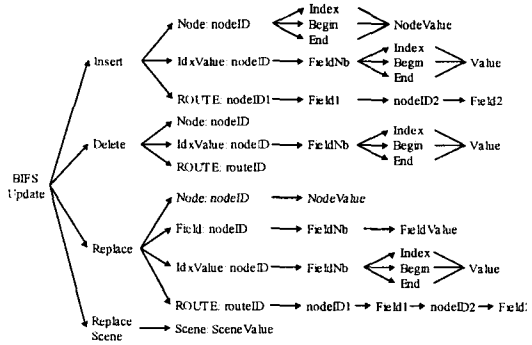

그림 5 Command Structure of BIFS

다음은 우리가 구현한 파서의 개략적 흐름을 보여준다.

```
parseScene() {
 do {
  int code = decoder.parseInt(2);
  switch(code) {
   case 0: parseInsertionCommand(); break;
   case 1: parseDelectionCommand(); break;
   case 2: parse ReplacementCommand(); break;
   case 3: sceneReplacementCommand(); break;
  }
  cont = decoder.parseInt(1);
 } while( cont == 1);

sceneReplacementCommand() {
 int reserved  = decoder.parseInt(6);
 useNames  = decoder.parseInt(1);
 parseProtoList();
 parseSFNode(NT_TOPNODE);
 int hasRoutes = decoder.parseInt(1);
 if(hasRoutes == 1) {
  parseRoutes();
 }
}
```

위의 루틴은 초기에 BIFS Scene을 구성하기 위해 사용된다. useName은 VRML의 USE를 파일에서 사용하고 있는지를 나타내는 전역변수이다. parseProtoList()는 프로토로 선언된 노드들을 파싱하고 paseSFNode()는 SingleField노드를 파싱하는 것으로 NT_TOPNODE의 의미는 현재 SceneRoot부터 파싱함을 나타낸다. 여기서 VRML이 BIFS가 다른점이 발견 되는데 바로 TOPNODE의 선언이다. VRML은 텍스트 형태로 파일에 들어가기 때문에 그냥 노드 이름이 들어가지만, BIFS의 경우 노드를 파일에 넣을 땐 코딩을 하게 된다. 이를 위하여 BIFS에서는 SF2DNode, SF3DNode, SFTopNode등의 31개의 NDT를 사용하는데, 같은 노드라도 NDT가 다르면 다른 Bits수로 코딩이 된다. 위와 같이 NDT가 TopNode인 경우에는 3bit로 노드를 코딩하고 있다.

다음은 각 Node를 파싱하는 루틴으로 Scene Root를 시작으로 각 BIFS노드와 그 필드를 파싱하게 된다. 여기서 노드는 children으로 또 다른 Node를 가질 수 있으므로 recursion이 발생하게 된다.

```
BaseNode parseSFNode(int parentNDT) {
 BaseNode node = null;
 int isReused = decoder.parseInt(1);
 if(isReused == 1) { // refers another node
  int nodeID =
    decoder.parseInt(bifsConfig.nodeIDbits);
 }
 else {
  int nbBits = getNDTnbBits(parentNDT);
  int childNodeType = decoder.parseInt(nbBits);
  int nodeType =
   getNodeType(parentNDT, childNodeType);
  int isUpdateable = decoder.parseInt(1);
  if(isUpdateable == 1) {
    int nodeID=
    decoder.parseInt ( (bifsConfig.nodeIDbits));
   if(useNames == 1) {
    parseString();
   }
  }
 }
 if(childNodeType == 0) { } // extension case
 node = createNode(new Integer(nodeType));
 node.setID(-1);
 bifsScene.nodes.add(node);
 if(root == null) {
  root = node;
  node.bifsScene.setSceneRoot(root);
 }
 int isUpdateable = decoder.parseInt(1);
 if(isUpdateable == 1) { // equivalent to DEF
    node.nodeID =
     decoder.parseInt(bifsConfig.nodeIDbits);
  if(useNames == 1) {
```

```
    node.defName = decoder.parseString();
   }
  }
  int maskDesc = decoder.parseInt(1);
  if(maskDesc == 1)  // mask node description
    parseMaskNodeDescription(node);
  else { // list node description
    parseListNodeDescription(node);
  }
  return node;
}
```

위의 코드에서 isReused는 VRML의 USE와 같은 의미로 앞서서 DEF가 선언된 노드를 참조하는 경우에 해당한다. BIFS에서 DEF선언된 노드는 DefName이나 NodeId를 부여해서 사용하고 있는데 여기서는 USE 문을 만났을 때 NodeId를 가지고 DEF가 선언된 노드를 찾게 된다. 앞서 말한바와 같이 NDT에 따라서 자식노드를 코딩하는 bits수가 달라지게 되므로 부모 NDT를 가지고 파싱해야 할 자식의 코딩된 bits수를 얻게 된다. 따라서 getNDTnbBits()는 부모 NDT를 주고 자식의 파싱시 읽어야 할 bits수를 얻어오는 함수이다. bifsConfig는 각 node, route, proto에 대해서 코딩할 때 사용하고 있는 bits수에 대한 정보를 가지고 있는데 이는 DMIF의 DecoderConfigDescriptor내에 있는 decoderSpecificInfo로부터 그 값을 읽어 들인다. isUpdatable은 VRML에서 DEF 선언문에 대응한다. 따라서 선언된 노드에 NodeId를 부여하게 되고 만일 스트링까지 부여하면 useNames 필드가 세팅되어 있고 defName을 스트링 형태로 읽으면 된다. 여기까지의 파싱 과정을 VRML 형태로 적어보면 다음과 같다.

DEF NodeID defName Group { fields }
USE nodeID defName

다음은 실제 BIFS Node를 만드는 곳으로 createNode()에서 하게 된다. 여기서는 해당 nodeType을 주면 미리 HashTable에 만들어진 각 class의 생성자를 불러서 해당 node class를 만들게 된다. 각 BIFS Node class에 대한 생성자를 정확하게 만드는 것이 BIFS 파서에 있어서 중요하다. 실제 문서에는 각 Node들에 대해서 Field들만 정해져 있고 이들 필드들이 NDT에 따라서 몇 bits로 코딩되는지 정보만 들어있다. 따라서 이를 참조해서 제대로 된 노트 클래스 생성자를 만드는 것이 중요하다.

다음은 Group노드의 클래스 정의와 생성자를 보여준다.

```
public class BIFSGroup extends BaseNode {
  MFNode addChildren;              // eventIn
  MFNode  removeChildren;          // eventIn
  MFNode  children;                // exposedField
  int numFields = 3;
  public BIFSGroup(Scene s) {
    super(s);
    field = new BaseField[numFields];
    ndt   = NT_3D;
    addChildren    = new MFNode(NT_3D);
    removeChildren = new MFNode(NT_3D);
    children       = new MFNode(NT_3D);
    field[0] = addChildren;
    field[1] = removeChildren;
    field[2] = children;
    // from node coding table
    numALLfields = numFields;
    numDEFfields = 0;
    numDYNfields = 0;
    nDEFbits   = 0;
    nINbits    = 2;
    nOUTbits   = 0;
    def2all    = new int[numFields];
    in2all     = new int[numFields];
    dyn2all    = new int[numFields];
    def2all[0] = 2; def2all[1] = 0; def2all[2] = 0;
    in2all[0]  = 0; in2all[1]  = 1; in2all[2]  = 2;
    dyn2all[0] = 0; dyn2all[1] = 0; dyn2all[2] = 0;
  }
}
```

여기서 주의해야 할 것은 MF3DNode로 정의된

addChildren, removeChildren, children의 field를 만드는 것이다. 실제로 이들은 children으로써 또 다른 node를 갖기 때문에 이 children을 파싱하고자 할 때 몇 bits로 노드들이 코딩되어 있는지를 알아야 하기 때문이다. 표준문서에 보면 Node Coding Table이 있는데, 이도 주의 깊게 사용해야 한다. BIFS에서 사용하는 fieldID는 5가지 모드를 갖는다. 즉 ALL, DEF, IN, OUT, DYN등으로 ALL은 보통 fieldId와 매핑되고 DEF, IN, OUT, DYN은 각각 모드에서 사용하고 있는 Field와 코드를 node coding table에서 기술하고 있다. 위의 노드 생성자에서 def2all, in2all, dyn2all은 각각의 모드에서 해당 field를 찾고자 할 때 쓰인다. 이에 대한 좀 더 구체적인 다음에서 설명한다.

각 노드의 필드들은 Vector나 리스트 형태로 들어가 있을 수 있다. 여기서는 리스트 형태로 들어간 것을 기준으로 파서의 구현을 설명한다.

```
void parseListNodeDescription(BaseNode node) {
  int endFlag = decoder.parseInt(1);
  while(endFlag == 0) {
    if(node.proto != null) { }     // proto case
    else {                         // non-proto case
      int fieldRef;
      fieldRef  = decoder.parseInt(node.nDEFbits);
      int selField = node.def2all[fieldRef];
      BaseField f = node.field[selField];
      this.parseField(f);
    }
    endFlag = decoder.parseInt(1);
  }
}
```

위의 함수를 보면 endFlag가 1이 될 때까지 필드들을 계속 읽게 되는데, endFlag는 VRML의 "}"에 대응된다. 그리고 필드를 읽을 때 node.nDEFbits 만큼 읽어서 각 노드 생성자에서 넣어준 값인 def2all을 통해서 어떤 field 인지를 찾고 나서 필드 파싱을 시작하게 된다. 이밖에도 각 노드의 필드라든가 route등의 파싱이 고려 되어야 한다.

위와 같이 파싱을 하고 나면 Java3D를 사용해서 Scene Graph를 구성한다. VRML이나 BIFS의 Scene graph는 Java3D에서 제공하는 Scene graph 구조와 비슷하고, 또한 Node들도 일대 일로 매핑 되는 경우가 많아 node들을 Java3D 노드로 구현하는데 용이하다.

5. 결론

본 논문에서는 MPEG-4 시스템의 구현에 있어서 중요한 컴포넌트들인 DMIF와 BIFS 파서를 Java와 Java3D기반으로 구현할 때의 고려 사항들에 대하여 기술하였다. 이들 언어나 그래픽스 API를 쓸 때 DMIF나 BIFS 파서 구현 시 다른 언어를 사용했을 때하고 별다른 차이는 없기 때문에, 여기서는 Java와 Java3D를 이용하였을때의 구현의 장단점 정도만 서술 하였다. 본 논문에서 강조하고자 하는 것은 MPEG-4 시스템 전체를 Java와 Java3D를 가지고 구현하고 있다는 것이다. 현재 MPEG-4 시스템을 완벽하게 구현한 사례가 세계적으로 없고, 일부만 구현되어 있다. 그 중에서도 Java기반으로 구현된 것은 전무하다. 그러므로 MPEG-4가 다양한 응용분야를 추구하고 이식성이 높은 시스템을 추구하고 있기 때문에 그 구현에 있어서 Java와 Java3D를 사용하는 것은 대단히 의미가 있는 일이라 할 수 있다.

[참고문헌]

[1] MEPG-4 official site: http://www.cselt.it/mpeg/

[2] MPEG-4 System site: http://garuda.imag.fr/MPEG4/

[3] MPEG-4 Part 1: Systems(ISO 14496-1), doc, N2501, Atlantic City, N.J., USA, Oct. 1998.

[4] MPEG-4 Part 6: DMIF(ISO 14496-6), doc, N2506, Atlantic City, N.J., USA, Oct. 1998.

[5]VRML(ISO 14772-1), "Virtual Reality Modeling Language", April 1997.

[6]임영권, "기술적 퓨전, 멀티미디어 스트리밍", 마이크로소프트웨어, 2001.6, pp 212-219.

[7]이재용, "스트리밍에 활력 불어넣는 MPEG-4 의 힘", 마이크로소프트웨어, 2001.6, pp 238-244.