Stereoscopic Metadata Format Specification

Version 1.4
May 23rd, 2012

3dtv.at
Peter Wimmer
Wankmüllerhof. 9
4020 Linz
Austria

http://www.3dtv.at
office@3dtv.at

Introduction

Stereoscopic movies can be encoded in several different formats. In this specification, the term *layout* is used for the way left and right views are arranged in video files—opposed to the general term *format*, which includes the container format, video and audio codec as well as the layout. Common layouts are side-by-side, over/under, interlaced, dual stream and separate left/right files. Information about the layout is usually not present in the file, so the user has to select the layout when opening the file in order to play it properly.

Although it would be advantageous to embed layout information in the file, it is not possible with all container formats or without recompressing the file. For these reasons, it makes sense to provide means to store metadata in separate files.

A specification for stereoscopic Windows Media files and a tool to embed stereoscopic metadata into Windows Media files are available on the 3dtv.at web site. It is recommended to embed stereoscopic metadata directly into Windows Media files and either provide stereoscopic metafiles for compatibility reasons only or do not provide them at all. For other container formats, it is highly recommended to use stereoscopic metafiles according to this specification.

The stereoscopic metadata format covered by this specification is implemented in Stereoscopic Player. Besides the layout, the stereoscopic metadata format holds several other bits of information about the corresponding video file. The stereoscopic metadata format may be implemented royalty free by other vendors.

Format Overview

The stereoscopic metadata format is a binary format which consists of three blocks.

1. File header
2. List of categories
3. List of video metadata

Because Stereoscopic Player uses the same format to store its video library, it can hold information for many files. The stereoscopic metadata format also supports the hierarchical

structure of the video library. Each video belongs to a *category*, which in turn can be a sub-item of another category.

Note: Recent versions of the Stereoscopic Player do not display the categories anymore. In the long term, we plan to replace the categories with tags, where multiple tags can be assigned to a video. The categories are still shown when the player is running in developer mode. The developer mode can be enabled by pressing Ctrl+Alt+D while the player's main window is active.

A stereoscopic metafile should contain information for a single video file only. It must contain the video's category and all its sub-categories. The file extension for stereoscopic metafiles is *.svi (because metafiles were called *stereoscopic video information files* in early versions of Stereoscopic Player).

Detailed Format Description

A stereoscopic metafile start with a signature which allows identifying the file type and format version. Up to now, four versions have been specified. If a field is not supported in all format versions, it is mentioned in the description. Stereoscopic metafiles should use the `StereoVideoInfo` signature. `Stereovideo-Library` is reserved for use by Stereoscopic Player only.

Field	Size	Description
Signature	21 or 25 bytes	`StereoVideoInfo[V1.0]` or `StereoVideoInfo[V1.1]` or `StereoVideoInfo[V1.2]` or `StereoVideoInfo[V1.3]` or `StereoVideoInfo[V1.4]` or `Stereovideo-Library[V1.0]` or `Stereovideo-Library[V1.1]` or `Stereovideo-Library[V1.2]` or `Stereovideo-Library[V1.3]` or `Stereovideo-Library[V1.4]`
Categories	4+X bytes	Number of categories, followed by category data (see Table 2).
Videos	4+X bytes	Number of videos, followed by video metadata (see Table 5). The number of videos should be one for stereoscopic metafiles.

Table 1: File header

Important: If the format version is 1.4 or greater, all strings in the category and video metadata blocks described below are stored in Unicode format (two bytes per character), otherwise strings are stored in ANSI format.

The signature is followed by category blocks. Each video metadata block must be assigned a category block which in turn must be assigned a parent category. Only root categories do not have a parent category. Root categories must not contain videos. Consequently, a stereoscopic

metafiles contains at least two categories: A category the video belongs to and one to the three root categories (see Table 3). Table 4 lists categories Stereoscopic Player uses for new videos. When defining your own categories, keep in mind they are identified by their ID—if two categories share the same name but have a different ID, they will show up as two different categories in Stereoscopic Player's video library.

Field	Size	Description
ID	8 bytes	Unique random number, identifying the category
Parent ID	8 bytes	ID of parent category or zero for root categories
Last change	8 bytes	Date and time in Borland Delphi `TDateTime` format
Flags	1 byte	Bit 1: Category contains at least one video which has already been opened Bit 2-7: Unused Bit 8: Reserved (public category) In stereoscopic metafiles, the first bit should always be set to one, the remaining bits should be set to zero.
Title	2+X bytes	Length of string, followed by string characters
Extension blocks	2+X bytes	Number of extension blocks, followed by extension blocks. No extensions blocks have been specified for categories yet. Writers should set this field to zero, readers should skip the specified number of blocks. To skip a block, read its size and skip the specified number of bytes (see Table 6). This field is only available in format version 1.1 or better.

Table 2: Category block

Title	ID
Files	2810800629978329
DVDs	2810833035958617
URLs	657670585994094975
Devices	655086674328735564

Table 3: Root categories

Title	ID	Parent ID
New Files	2811666454519930	2810800629978329
New DVDs	2811709215229810	2810833035958617
New URLs	657670632571368574	657670585994094975

Table 4: Other categories

The stereoscopic metadata format supports different kind of *media types*. Whenever a file or DVD is opened in Stereoscopic Player, the player searches its video library if the video already exists. If it does, the information from the library item is used. Else, it looks for a stereoscopic metafile on the hard disk. If no stereoscopic metafile can be found, it tries to download the metafile from a web server. If no metafile is available on the server either, Stereoscopic Player prompts the user to select the layout and aspect ratio.

Searching for stereoscopic metadata requires that the file or DVD can be uniquely identified, which is possible by using a hash value calculated from the file's or DVD's content. Web streams are identified by their URL, because streamed content is downloaded during playback and therefore not available when the stream is opened. The field *hash* is zero in this case.

Many fields in the *video metadata block* are not required to play back the video properly, but are used for informal purposes only. For example, they are shown in Stereoscopic Player's video library or video properties dialog. Nevertheless, it is strongly recommended to set all values according to the specification.

Field	**Size**	**Description**
Media type	1 byte	0: File 1: DVD 2: URL -1: Capture device -2: Separate files (format version 1.3 or better)
ID	8 bytes	Unique random number
Hash	8 bytes	Hash value identifying the video file or DVD (*media type* 1 or 2). Zero for URLs (*media type* 2). For separate files, the hash value is made up of the hash value of all files, including the audio file if *audio mode* is set to 1. The exclusive OR operation is used to combine the hash values.
Category ID	8 bytes	ID of category the video belongs to. This must be a valid category present in the file.
Last change	8 bytes	Date and time in Borland Delphi `TDateTime` format
Title	2+X bytes	Length of string, followed by string characters
Video file count	1 byte	This field is only present when *media type* is set to -2. For separate left and right files, *video file count* must be set to two. More than two files are allowed, but ignored by the current implementation of Stereoscopic Player.
Audio mode	1 byte	0: No audio 1: Separate audio file 2: Use audio stream of left file 3: Use audio stream of right file This field is only present when *media type* is set to -2.
Video file names	2+X bytes	Length of string, followed by string characters. The string contains either the file name, DVD path (without *VIDEO_TS*) or the URL, depending on the *media type*.

		For stereoscopic metafiles, the file name should be stored without path, because the metafile must be stored in the same directory as the video file anyway. The DVD path should be empty (length of string set to zero, no following characters). If the field *video file count* is present, at least two file names must be specified. Files should be ordered from left to right. Because left and right files might be stored in different directories, relative paths are allowed in this case (including '../' to access the parent directory).
Audio file name	2+X bytes	Length of string, followed by string characters. This field is only present when *media type* is set to -2 and *audio mode* is set to 1.
Information	2+X bytes	Length of string, followed by string characters
Source	2+X bytes	Length of string, followed by string characters
Layout	1 byte	0: Monoscopic 1: Interlaced, right line first 2: Interlaced, left line first 3: Side-by-side, right image first 4: Side-by-side, left image first 5: Over/under, right image top 6: Over/under, left image top 7: Separate streams (*media type* = 1) or multiple files (*media type* = -2), left to right 8 2D and depth 9 Depth and 2D 10 Multi-view (tiled), top to bottom (format version 1.4 or better) 11 Multi-view (tiled), bottom to top (format version 1.4 or better) 12 Frame-sequential, right frame first 13 Frame-sequential, left frame first 14 Multi-view (five tiles), top to bottom (format version 1.4 or better) 15 Multi-view (five tiles), bottom to top (format version 1.4 or better) 16 Separate streams, right to left 128 SIS attachment 129 SENSIO Hi-Fi 3D
Separation	2 bytes	Distance in pixels between left and right image (for side-by-side and over/under layout only, else zero).
Horizontal tiles	2 bytes	Number of horizontal tiles for layouts 10, 11, 14 and 15, otherwise the value is not present. The value is ignored for layouts 14 and 15. Format version 1.4 or better.

Vertical tiles	2 bytes	Number of vertical tiles for layouts 10, 11, 14 and 15, otherwise the value is not present. The value is ignored for layouts 14 and 15. Format version 1.4 or better.
Left tile	2 bytes	Tile to be used as the left view for layouts 10, 11, 14 and 15, otherwise the value is not present. Format version 1.4 or better.
Right tile	2 bytes	Tile to be used as the right view for layouts 10, 11, 14 and 15, otherwise the value is not present. Format version 1.4 or better.
Left cropping	2 bytes	Left cropping in pixels. Format version 1.1 or better.
Right cropping	2 bytes	Right cropping in pixels. Format version 1.1 or better.
Top cropping	2 bytes	Top cropping in pixels. Format version 1.1 or better.
Bottom cropping	2 bytes	Bottom cropping in pixels. Format version 1.1 or better.
Parallax (horiz.)	2 bytes	Horizontal parallax adjustment in pixels. Negative values are allowed. Format version 1.1 or better.
Parallax (vert.)	2 bytes)	Vertical parallax adjustment in pixels. Negative values are allowed. Format version 1.1 or better.
Aspect ratio X	2 Bytes	Either specifies aspect ratio or set to zero for default aspect ratio (assuming square pixels).
Aspect ratio Y	2 Bytes	Either specifies aspect ratio or set to zero for default aspect ratio (assuming square pixels).
Width	2 Bytes	Horizontal video resolution in pixels.
Height	2 Bytes	Vertical video resolution in pixels.
File size	8+X Bytes	Size of video file or DVD data in bytes. If the field *video file count* is present, the size of each file is specified separately as well as the size of a separate audio file (if *audio mode* is set to 1).
Duration	8 Bytes	Video duration in DirectX `REFTIME` format (seconds in double precision floating point format).
Flags	1 Byte	Bit 1: Half horizontal resolution Bit 2: Half vertical resolution Bit 3: Video has already been opened Bit 4: Metadata have been updated Bit 5: Deinterlacing on Bit 6: Left view one field ahead Bit 7: Right view one field ahead Bit 8: Deinterlacing auto Bit 1 and 2 are mutual exclusive and will be ignored if an aspect ratio has been specified (should both be zero in this case). If no aspect ratio has been specified, these bits cause the player to stretch the video to double width or height, respectively. Bit 3 is set by Stereoscopic Player once a video has been opened, so that it is possible to hide sample video library

		items which has never been opened. Bit 4 is used by the Stereoscopic Player video library to identify items which have not been updated with information from a metadata server yet. Bit 5 should be set for files which require deinterlacing). Bit 8 should be set if the player should automatically decide is deinterlacing is required. Bits 5 and 8 are mutually exclusive. Bits 6 and 7 are mutually exclusive. If left and right image have been captured at the same time, none of the bits should be set. Bits 3-4 should be set zero for stereoscopic metafiles.
Rotation flags	1 byte	Bit 1: Left image rotated left Bit 2: Left image rotated right Bit 3: Right image rotated left Bit 4: Right image rotated right Bit 4-8: Reserved. Bits 1 and 2 as well as bits 3 and 4 are mutual exclusive. If left rotation bit is set, a right rotation bit must be set as well. Note that *left image rotated left* means the left image will be rotated right during playback to compensate the left rotation in the file. This field is only available in format version 1.2 or better.
JPEG image	4+X bytes	Size of embedded JPEG preview image (160 x 120 pixels, max. 10240 bytes), followed by image data.
Extension blocks	2+X bytes	Number of extension blocks followed by extensions blocks. Only one extension block has been specified yet (see Table 7). Writers should either set this field to one and write the extensions block or set it to zero. Readers should skip all unknown blocks. To skip a block, read its size and skip the specified number of bytes (see Table 6). This field is only available in format version 1.1 or better.

Table 5: Video metadata block

Starting with format version 1.1, category blocks and video metadata blocks may include extension blocks which allow extending the stereoscopic metadata format without introducing incompatibilities with previous versions. Readers can skip extension block without knowing the meaning of their content.

Field	**Size**	**Description**
ID	2 bytes	Extension block ID
Size	2 bytes	Extension block data size
Data	X bytes	Data

Table 6: Extension block

Field	Size	Description
Author	2+X bytes	Length of string, followed by string characters
Copyright	2+X bytes	Length of string, followed by string characters

Table 7: Video metadata extension block 0

Calculating Hash Values

Video files are identified by a 64 bit hash value, which is derived from 117 bytes of the file content, using the exclusive OR operation. Each time two bytes have been processed, the resulting value is shifted left by one bit. Although last shifting operation is unnecessary and causes the loss of one bit, the algorithm cannot be changed anymore because of compatibility reasons. The following sample code shows a possible implementation of the hash algorithm.

Delphi implementation:

```
function GetFileHash(Filename: String): Int64;
var
  i: Integer;
  FileStream: TFileStream;
  NextByte: Byte;
begin
  Result := 0;

  try
    FileStream := TFileStream.Create(Filename, fmOpenRead);
  except
    Exit;
  end;

  for i := 1 to 57 do begin
    FileStream.Position := (FileStream.Size-1) * (2*i-1) div (2*57);
    FileStream.ReadBuffer(NextByte, SizeOf(NextByte));
    Result := Result xor NextByte;

    FileStream.Position := (FileStream.Size-1) * i div 57;
    FileStream.ReadBuffer(NextByte, SizeOf(NextByte));
    Result := Result xor NextByte;

    Result := Result shl 1;
  end;

  FileStream.Free;
end;
```

C# implementation:

```
public static long GetFileHash(string filename) {
  FileStream fileStream = null;
  try {
```

```
    fileStream = new FileStream(filename, FileMode.Open, FileAccess.Read,
                                FileShare.Read);

    long result = 0;
    for (int i = 1; i <= 57; i++) {
      fileStream.Position = (fileStream.Length-1) * (2*i-1) / (2*57);
      int nextByte = fileStream.ReadByte();
      result = result ^ nextByte;

      fileStream.Position = (fileStream.Length-1) * i / 57;
      nextByte = fileStream.ReadByte();
      result = result ^ nextByte;

      result = result << 1;
    }
    return result;
  } catch {
    return 0;
  } finally {
    if (fileStream != null) {
      fileStream.Close();
    }
  }
}
```

The absolute value of the hash value is used as file name for metadata server files (stereoscopic metafiles located on a web server).

To identify DVDs, only the file VIDEO_TS.IFO is used. Taking all files into in the *VIDEO_TS* folder account would take too long, because the access time of DVD drives is quite bad. The file is read in 64 bits block and the exclusive OR operation is applied.

Delphi implementation:

```
function GetDVDHash(Folder: String): Int64;
var
  i: Integer;
  FileStream: TFileStream;
  NextValue: Int64;
begin
  Result := 0;

  try
    FileStream := TFileStream.Create(Folder+'\VIDEO_TS\VIDEO_TS.IFO',
                                     fmOpenRead);
  except
    Exit;
  end;

  for i := 1 to FileStream.Size div SizeOf(NextValue) do begin
    FileStream.ReadBuffer(NextValue, SizeOf(NextValue));
    Result := Result xor NextValue;
  end;

  FileStream.Free;
end;
```

C# implementation:

```
public static long GetDVDHash(string folder) {
  FileStream fileStream = null;
  try {
    fileStream = new FileStream(folder + "\\VIDEO_TS.IFO", FileMode.Open,
                                FileAccess.Read, FileShare.Read);
    BinaryReader reader = new BinaryReader(fileStream);

    long result = 0;
    for (int i = 1; i <= fileStream.Length / 8; i++) {
      long nextValue = reader.ReadInt64();
      result = result ^ nextValue;
    }
    return result;
  } catch {
    return 0;
  } finally {
    if (fileStream != null) {
      fileStream.Close();
    }
  }
}
```

Date and Time Format

The *last change* fields in the video metadata block and category block are encoded in Borland Delphi's `TDateTime` format. `TDateTime` maps to a floating point value at double precision. The integral part of a Delphi `TDateTime` value is the number of days that have passed since 12/30/1899. The fractional part of the `TDateTime` value is fraction of a 24 hour day that has elapsed. Following are some examples of `TDateTime` values and their corresponding dates and times (taken from the Borland Delphi documentation):

Value	Date and Time
0	12/30/1899 12:00 am
2.75	1/1/1900 6:00 pm
-1.25	12/29/1899 6:00 am
35065	1/1/1996 12:00 am

Table 8: Sample TDateTime values

Whereas conversion of time to the fractional part of `TDateTime` is a straightforward task, date conversion is not. For this reason, here are code samples which show how to do the conversion on the Windows platform.

C++ implementation:

```
// Return today's date in Borland Delphi compatible format.
static int Date() {
  SYSTEMTIME sSystemTime;
  GetLocalTime(&sSystemTime);
  int iDate;
  EncodeDate(sSystemTime.wYear, sSystemTime.wMonth, sSystemTime.wDay,
             &iDate);
  return iDate;
```

```
}

// Encode a date the same way as Borland Delphi does.
static bool EncodeDate(WORD wYear, WORD wMonth, WORD wDay, int *pDate) {
  const int iMonthDays[2][12] = {
    {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
  };

  int iLeapYear = IsLeapYear(wYear) ? 1 : 0;

  if (wYear >= 1 && wYear <= 9999 &&
      wMonth >= 1 && wMonth <= 12 &&
      wDay >= 1 && wDay <= iMonthDays[iLeapYear][wMonth]) {

    for (int i = 0; i < wMonth - 1; i++) {
      wDay += iMonthDays[iLeapYear][i];
    }

    int iTemp = wYear - 1;
    *pDate = iTemp * 365 + iTemp / 4 - iTemp / 100 + iTemp / 400 + wDay -
             693594;

    return true;
  }

  return false;
}

// Is specified year a leap year?
static bool IsLeapYear(WORD wYear) {
  return (wYear % 4) == 0 && ((wYear % 100) != 0 || (wYear % 400) == 0);
}
```

C# implementation:

```
public static class DelphiDate {

  // Return today's date in Borland Delphi compatible format.
  public static int Date() {
    DateTime date = DateTime.Now;
    return ConvertDate(date);
  }

  // Convert .Net to Delphi date.
  public static int ConvertDate(DateTime date) {
    return EncodeDate(date.Year, date.Month, date.Day);
  }

  // Convert .Net to Delphi date/time.
  public static double ConvertDateTime(DateTime date) {
    return EncodeDate(date.Year, date.Month, date.Day) + (date - new
             DateTime(date.Year, date.Month, date.Day)).TotalDays;
  }

  // Convert Delphi to .Net date.
  public static DateTime ConvertDate(int delphiDate) {
    int year, month, day;
    DecodeDate(delphiDate, out year, out month, out day);
    return new DateTime(year, month, day);
```

```
}

// Convert Delphi to .Net date/time.
public static DateTime ConvertDateTime(double delphiDateTime) {
  int year, month, day;
  DecodeDate((int)delphiDateTime, out year, out month, out day);
  return new DateTime(year, month, day) +
              TimeSpan.FromDays(delphiDateTime - (int)delphiDateTime);
}

// Encode a date the same way as Borland Delphi does.
public static bool EncodeDate(int year, int month, int day,
                              out int date) {
  UInt16[,] monthDays = {
    {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
  };

  int leapYear = IsLeapYear(year) ? 1 : 0;

  if (year >= 1 && year <= 9999 &&
      month >= 1 && month <= 12 &&
      day >= 1 && day <= monthDays[leapYear, month-1]) {

    for (int i = 0; i < month - 1; i++) {
      day = day + monthDays[leapYear, i];
    }

    int temp = year - 1;
    date = temp * 365 + temp / 4 - temp / 100 + temp / 400 +
           day - 693594;

    return true;
  }

  date = 0;
  return false;
}

// Another version for convenience.
public static int EncodeDate(int year, int month, int day) {
  int delphiDate;
  EncodeDate(year, month, day, out delphiDate);
  return delphiDate;
}

// Helper method for modulo division.
private static void DivMod(int dividend, ushort divisor,
                           out int result, out int remainder) {
  result = dividend / divisor;
  remainder = dividend % divisor;
}

// Decode a Delphi date to year, month and day.
public static bool DecodeDate(int delphiDate, out int year,
                              out int month, out int day) {
  UInt16[,] monthDays = {
    {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
  };
```

```
    const int d1 = 365;
    const int d4 = d1 * 4 + 1;
    const int d100 = d4 * 25 - 1;
    const int d400 = d100 * 4 + 1;
    int y, m, d, i;

    delphiDate += 693594; // Days between 1/1/0001 and 12/31/1899;

    if (delphiDate <= 0) {
      year = 0;
      month = 0;
      day = 0;
      return false;
    } else {
      delphiDate--;
      y = 1;
      while (delphiDate >= d400) {
        delphiDate -= d400;
        y += 400;
      }
      DivMod(delphiDate, d100, out i, out d);
      if (i == 4) {
        i--;
        d += d100;
      }
      y += (i * 100);
      DivMod(d, d4, out i, out d);
      y += (i * 4);
      DivMod(d, d1, out i, out d);
      if (i == 4) {
        i--;
        d += d1;
      }
      y += i;
      int leapYear = IsLeapYear(y) ? 1 : 0;
      m = 1;
      while (true) {
        i = monthDays[leapYear, m-1];
        if (d < i)
          break;
        d -= i;
        m++;
      }
      year = y;
      month = m;
      day = d + 1;
      return true;
    }
  }

  // Is specified year a leap year?
  public static bool IsLeapYear(int year) {
    return (year % 4) == 0 && ((year % 100) != 0 || (year % 400) == 0);
  }
}
```