THE MICROSOFT JOURNAL FOR DEVELOPERS

**DECEMBER 2011** VOL 26 NO 12

# msdn magazine

Microsoft

# Saving and Reusing Video Encoding Settings

Adi Shavit

From remote surveillance systems to embedded robotic platforms, applications that automatically or autonomously record video are getting ever more ubiquitous as the price of equipment such as cameras and storage space drops and computational performance increases. However, configuring such systems remotely often isn't an option, while deployment and installation must be simple and automatic.

I recently needed a way to preconfigure video recording settings on a remote system so it would automatically load and use the same settings at startup time. A Web search brought up many similar questions in multiple forums going back more than a decade, but with only partial and unsatisfying solutions.

In this article, I'll present a simple yet general way to allow video-processing applications to save video with consistent compression settings, thus avoiding having to manually specify the codec settings each time the application or the machine is started. I'll show how to access the internal setting buffers of a codec so that they can be easily saved, reloaded and reused. This will work for any video codec installed on the machine, without requiring the use of any codec-specific APIs.

This article discusses:

- Saving video with VfW
- Common ways to save settings
- A better hybrid approach
- The advantages of this approach

Technologies discussed:

Video for Windows

Code download available at:

code.msdn.microsoft.com/mag201112Video

## Video 101

Raw video data is huge. Consider a modest VGA camera ambling along at a humble rate of 15 frames per second. Each three-channel 640×480 pixel frame is 900KB and generates a data rate of over 13MBps (that's about 105Mbps)! Slightly less modest video equipment, of the type you might find on a mobile phone, might have high definition (HD) resolutions, higher frame rates or more than 8 bits of color depth. A 90-minute HD movie will consume approximately 1TB in raw form. Fortunately, video data contains a lot of redundancy, and video compression algorithms can store video files quite efficiently. The type of compression algorithm chosen depends on the particular application, its requirements and its constraints. Several factors vary among compression methods, including real-time rates, performance considerations and the trade-off between visual quality and the resulting file size.

On Windows, video compression services are provided by codecs (COmpressor-DECompressor). I'll focus on the Video for Windows (VfW) API for saving compressed .avi video files. Audio Video Interleave (.avi) is a multimedia container file format that can contain multiple streams of both audio and video and allow synchronous audio-with-video playback of selected streams. I'll show how compression codec settings can be manually selected and stored in a persistent way so they can be loaded back later and automatically reused by the application on any computer on which the same codec is installed.

## Saving Video with VfW

Although it was introduced back in 1992, VfW is an enduring API, still widely used today. New codecs and codec implementations, such as the ffmpeg suite (ffmpeg.org), continue to provide a VfW interface on Microsoft Windows.

The typical flow for a video recording program usually takes the form of the following steps:

1. Create a new AVI file, create a new uncompressed video stream within it, choose the required compression settings and create a new compressed stream from the uncompressed one.
2. Add new frames as desired.
3. When done, release resources in reverse order.

Using VfW it looks as follows:

1. Open and prepare the AVI video file using the following:
    1. AVIFileInit: Initializes the AVIFile library. Must be called before using any other AVIFile functions.
    2. AVIFileOpen: Opens an AVI file.
    3. AVIFileCreateStream: Creates a new video stream within the AVI file. An AVI file can include multiple streams (of various types).
    4. AVISaveOptions: Presents a standard Compression Options dialog (see Figure 1). When the user is finished selecting the compression options, the options are returned in an AVICOMPRESSOPTIONS structure.
    5. AVIMakeCompressedStream: Creates a compressed stream from the uncompressed stream returned from AVIFileCreateStream and the compression filter returned from AVISaveOptions.
    6. AVIStreamSetFormat: Sets the image format for the stream.
2. Add frames to the video stream:
    1. AVIStreamWrite: Add a single frame to the video stream. Call repeatedly with additional frames as desired.
3. Clean up and close:
    1. AVIStreamRelease: Closes the compressed stream (from AVIMakeCompressedStream).
    2. AVIStreamRelease: Closes the *un*compressed stream (from AVIFileCreateStream).
    3. AVISaveOptionsFree: Frees the resources allocated by the AVISaveOptions function (this is often forgotten, leading to memory leaks).
    4. AVIFileRelease: Closes the AVI file (from AVIFileOpen).
    5. AVIFileExit: Exit the AVIFile library.

Explaining in detail how to use VfW for recording video is beyond the scope of this article. You can find many examples and tutorials online. I'll focus here only on Step 1.4: the compressor selection and codec settings part. The source code accompanying this article is based on the C++ implementation from the OpenCV (opencv.willowgarage.com) open source project and can be downloaded from code.msdn.microsoft.com/mag201112Video. However, the techniques I show here are applicable to—and easily integrated into—any video recording application that uses the VfW API and isn't specifically targeted at a particular implementation or programming language.

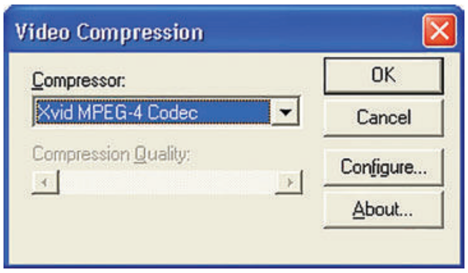

Figure 1 **The Video Compression Dialog Box Opened by the AVISaveOptions Function**

## Common Solutions

As shown earlier, the common way to specify the compression settings is by calling the AVISaveOptions function to display the Compression Options dialog box. The user can then manually select the desired codec. The chosen codec may or may not have a codec "Configure…" option, which allows making codec-specific settings customizations. Naturally, on systems without an active user or even a GUI, this dialog box isn't an option.

The common alternative to manual selection is to programmatically select a particular codec using the codec's *fourcc* (four character code; see fourcc.org) identifier, and to set some additional, general settings common to all codecs, such as data rate, quality and key frame rate. The major drawback of this method is that it doesn't allow setting any *codec-specific* settings. Further, it makes the codec use whatever its current internal settings are. These may be defaults or, even worse, settings arbitrarily set by other programs. Additionally, not all fourcc codecs available on a system support the VfW API, which may cause the compressed stream creation to fail.

Some codecs provide special configuration tools that allow changing their internal settings externally. These tools may take the form of a GUI or a command-line tool. However, most codecs don't provide such tools, and because each tool is customized for a particular codec, you'd have to learn its particular syntax and usage to take advantage of it.

What I wanted was the flexibility of the manual selection mode with automated selection of the codec and all its internal settings.

## The Best of Both Worlds

In the first approach, the AVISaveOptions call returns a fully filled-in AVICOMPRESSOPTIONS object. The second approach is, essentially, filling up some of the values in the AVICOMPRESSOPTIONS object, from within the code, without using AVISaveOptions.

In fact, it's possible to save the *full, internal* codec state after a call to AVISaveOptions so this state can later be restored without having to call AVISaveOptions again. The secret of *internal* video-compressor-specific data and format is in the opaque pointers lpParms and lpFormat of AVICOMPRESSOPTIONS. When returned from AVISaveOptions, these two pointers contain everything the AVIMakeCompressedStream function needs to make a compressed stream (these are presumably the resources released by AVISaveOptionsFree).

The essence of my technique, then, is saving the opaque data of lpParms and lpFormat to allow it to be reused. Fortunately, this is simple to do. The AVICOMPRESSOPTIONS struct contains two helpful integer members, cbParms and cbFormat, which state the size of the opaque binary buffers pointed to by lpParms and lpFormat, respectively. These sizes change according to the particular codec selected.

To summarize, there are three binary "blobs" that need to be saved after a call to

AVISaveOptions. These are the AVICOMPRESSOPTIONS object itself and the two binary buffers pointed to by its lpParms and lpFormat members. The length in bytes of these buffers is given, respectively, by cbParms and cbFormat. Restoring the codec settings is just reversing the process: Read the AVICOMPRESSOPTIONS object from the file and set its lpParms and lpFormat members to point at the respective binary buffers read from the file. The restored binary buffers are allocated and managed by the application itself.

There are many ways to store binary data. As shown in **Figure 2**, my sample code saves the data in binary form. Alternatively, in cases where nonprintable characters should be avoided (such as .txt or .xml files), I've successfully used Base-64 encoding to store the buffers.

Figure 2 **Example of Storing and Restoring Codec Settings**

```
bool CvVideoWriter_VFW::writeCodecParams( char const* configFileName ) const
{
  using namespace std;
  try
  { // Open config file
    ofstream cfgFile(configFileName, ios::out | ios::binary);
    cfgFile.exceptions ( fstream::failbit | fstream::badbit );
    // Write AVICOMPRESSOPTIONS struct data
    cfgFile.write(reinterpret_cast<char const*>(&copts_), sizeof(copts_));

    if (copts_.cbParms != 0)// Write codec param buffer
      cfgFile.write(reinterpret_cast<char const*>(copts_.lpParms), copts_.cbParms);

    if (copts_.cbFormat != 0)// Write codec format buffer
      cfgFile.write(reinterpret_cast<char const*>(copts_.lpFormat),
        copts_.cbFormat);
  }
  catch (fstream::failure const&)
  { return false; } // Write failed
  return true;
}

bool CvVideoWriter_VFW::readCodecParams( char const* configFileName )
{
  using namespace std;
  try
  { // Open config file
    ifstream cfgFile(configFileName, ios::in | ios::binary);
    cfgFile.exceptions (
      fstream::failbit | fstream::badbit | fstream::eofbit );
    // Read AVICOMPRESSOPTIONS struct data
    cfgFile.read(reinterpret_cast<char*>(&copts_), sizeof(copts_));

    if (copts_.cbParms != 0)
    { copts_Parms_.resize(copts_.cbParms,0);              // Allocate buffer
      cfgFile.read(&copts_Parms_[0], copts_Parms_.size());  // Read param buffer
      copts_.lpParms = &copts_Parms_[0];                  // Set lpParms to buffer
    }
    else
    { copts_Parms_.clear();
      copts_.lpParms = NULL;
    }

    if (copts_.cbFormat != 0)
    { copts_Format_.resize(copts_.cbFormat,0);            // Allocate buffer
      cfgFile.read(&copts_Format_[0], copts_Format_.size());// Read format buffer
      copts_.lpFormat = &copts_Format_[0];               // Set lpFormat to buffer
    }
    else
    { copts_Format_.clear();
      copts_.lpFormat = NULL;
    }
  }
  catch (fstream::failure const&)
  { // A read failed, clean up
    ZeroMemory(&copts_,sizeof(AVICOMPRESSOPTIONS));
    copts_Parms_.clear();
    copts_Format_.clear();
    return false;
  }
  return true;
}
```

## Sample Code Notes

The sample code captures video from a video camera or webcam and saves it to an AVI file. It uses the OpenCV VideoCapture class to access the camera and capture the video. Saving the video is done using a version of the OpenCV CvVideoWriter_VFW class that I modified to support saving the codec settings. I tried to keep the changes to the original code as small as possible. In the code, the second argument of CvVideoWriter_VFW::open is an std::string. If this string is four characters long (as in a *fourcc*), it's assumed to be a fourcc and the codec corresponding to this fourcc is chosen—converting from the four 1-byte characters to the fourcc integer code is commonly done with the mmioFOURCC macro, as in mmioFOURCC('D','I','V','X'); see tinyurl.com/mmioFOURCC for details. Otherwise, the string is taken as the name of a codec settings file. If the file exists, the settings are read from the file and used for the video compression. If it doesn't exist, then the Compression Options dialog box opens and a codec and its settings can be manually selected; the chosen settings are then saved to the file of the selected name to be reused the next time the application is run. Note: You'll need an OpenCV installation to compile and run the sample code.

## Several Advantages

The solution I presented has several advantages. Changing the compression settings for an application is just a matter of changing one of its arguments. There's no need to recompile it or even stop the running app. The method is codec-independent and works for all VfW codecs. Some codecs even allow for additional customized image processing such as frame resizing and deinterlacing. All these options are automatically captured within the codec settings file.

The method works just as well on legacy Microsoft Windows systems (from Windows 2000 onward) and also with older compilers such as Visual Studio 6.0. Also, it can be used with a variety of languages that expose the VfW interface, such as Visual Basic and C#. It can be deployed to multiple machines regardless of the current internal codec state of each machine.

The main caveat is that this solution is guaranteed to work only when the exact same version of the codec is used. If a codec is missing on a particular machine, or is a different version of the one used to make the settings file, the results might be undetermined. Obviously, nonVfW codecs can't be used, because this solution is based on the VfW API.

Possible extensions to this work include similarly saving audio compressor settings, which are handled identically to video compressor settings. Also, because AVISaveOptions natively supports AVIs with multiple streams, the same solution will also work in these cases, and the codec settings for each of the streams can be written to a single file.

The other common and more modern video and audio processing API on Windows is DirectShow. It should be possible to implement a similar solution for DirectShow as well, but that's a subject for a future article. ■

**ADI SHAVIT** *is a computer vision consultant. He has been working on image and video processing systems for more than 15 years, specializing in real-time video analysis. He can be reached at adishavit@gmail.com.*

**THANKS** to the following technical experts for reviewing this article: *Dana Shavit and Matthew Wilson*