TP011: Using AVI Files as Textures

Introduction

This paper describes how the video stream in an AVI file can be used as a RealiMation texture image. It describes the basic steps required to extract the image information from the AVI file and transfer it to RealiMation, along with pitfalls to be avoided, and tips on extracting the best performance.

Basic familiarity with the Win32 API and the RealiMation API (especially with regards to the way images are handled) is assumed. A code sample is provided which demonstrates these techniques in use.

The Basic Steps

Using an AVI file as a texture can be broken down into a number of simpler steps. The majority of the work is using the Win32 *AVI...* functions to extract the data from the video stream.

- Open the video stream from the .AVI file.
- Read the video stream format (image size, bit-depth)
- Create a RealiMation image of the correct size to match the video stream format.
- Initialise the video stream - this gets the video stream ready to decompress video frames.
- Calculate the correct frame number based on the time and extract the frame information for that frame.
- If required, perform any conversion from the AVI format to the RealiMation format.
- Send the image information to the RealiMation libraries, using the `RTSetImage()` function.

We repeat the last 3 steps each time the frame needs updating, until we've finished using the texture.

- Clean up when we've finished using the texture

The CAVIObject

We can encapsulate the functionality for playing AVI files a class called `CAVIObject`. This groups everything we need in one place, and makes adding AVI texture functionality to applications relatively easy.

The class definition of `CAVIObject` follows (source for the object is located in *Appendix A*). The constructor can take in either a RealiMation material or an image ID. If a material is used, then the `CAVIObject` creates a new image, and tells the material to use that image as a texture. If an image is used, then the pixel data for that image is replaced by the AVI texture.

After the class definition, we'll look in more detail at the parts which correspond to the stages listed above :

```
////////////////////////////////////////////////////////////////////////////
//
//     CAVIObject - A class which controls playback of an AVI file
//                  to beused as a RealiMation texture
//

// Playback mode - can either sync. the AVI file with the time given
// or just play every frame, ignoring the time

typedef enum {EveryFrameAVI = 0, RealTimeAVI} TAVIPlaybackMode;

class CAVIObject : public CObject
{
   public :
       // Construct AVI object, takes in either material ID or image ID
       CAVIObject (rtID id);

       virtual ~CAVIObject ();

       // Set which AVI file to use
       virtual BOOL SetAVIFilename (CString &avi_filename);

       // We've finished using the AVI file, release
       virtual void FinishAVIfile ();

       // Set the time to display at
       virtual void Update (Ord time);

       // Pick out a definate frame to use
       virtual void SetCurrentFrame (LONG frame_number);

       // Set and inquire the current playback mode
       virtual void SetPlaybackMode (TAVIPlaybackMode mode)
               { m_PlaybackMode = mode; } ;
       virtual TAVIPlaybackMode InqPlaybackMode ()
               { return m_PlaybackMode; };

protected :
       BOOL             m_bInitialised;    // Whether currently initialised
       TAVIPlaybackMode m_PlaybackMode;    // Current playback mode

       PAVISTREAM     m_pavi;  // Pointers to the video stream in the AVI file
       AVISTREAMINFO  m_avis;
       PGETFRAME      m_gapgf;

       Byte          *m_TempPixel;  // Holds onto the block of memory
                                    // allocated for the video image.
       rtID           m_VideoImage;
       rtID           m_VideoMaterial;
       rtImage        m_ImageInfo;
       DWord          m_ImageSize;

       DWord m_PreviousImageID;  // Previous image used by video material,
       BOOL  m_bCreatedImage;    // Whether image was temp one
                                  // created by CAVIObject

       LONG      m_lFrames;          // Number of frames in video stream
       LONG      m_CurrentIndex;     // Current frame
       LONG      m_LastIndex;        // Last frame displayed
       Ord       m_SamplesPerSecond; // Frame rate of video stream

       LPBITMAPINFOHEADER m_lpSrcFmt;      // Image format of source video stream
       LPBITMAPINFOHEADER m_lpbi;              // Frame read from video stream
       BOOL               m_bFlipImage;        // Whether image needs flipping
       Byte *             m_TempImageLine;     // Temp block of memory used
                                             // for flipping image if required

       // Extract frame_number from the AVI video stream
       virtual void GetFrame (LONG frame_number);

};
```

The `CAVIObject` constructor just looks at the type of ID being used (image or material), then creates any additional ID's it requires. The main work of initialisation is done in the `SetAVIFilename()` function. This is where the video stream in the file is opened, the correct size RealiMation image is created and the stream is prepared for reading.

Opening the video stream and extracting the information is done with simple calls to the Win32 API. First, we need to initialise the AVI file library, then open the video stream in the AVI file:

```
        // Initialise the AVI File library
    AVIFileInit();
        PAVISTREAM m_pavi;
        AVIStreamOpenFromFile(&m_pavi, avi_filename, streamtypeVIDEO,
                              0, OF_READ | OF_SHARE_EXCLUSIVE, NULL);
```

This opens the stream, and allows us to query the stream about number of frames, source format, etc. First, we get the `AVISTREAMINFO` structure :

```
AVISTREAMINFO     m_avis;
AVIStreamInfo(m_pavi, &m_avis, sizeof(m_avis))
```

This fills in the structure with information about the video stream. The only parts we are currently interested in are `dwRate` and `dwScale`. The combination of these gives us the frame rate at which the stream is designed to play. We will need this information for when we have to synchronise the AVI to the real time clock.

```
        m_SamplesPerSecond = (Ord)m_avis.dwRate / (Ord)m_avis.dwScale;
```

We also need to find out what the source format is - size and bitdepth :

```
        // Get the source format - currently can only deal with 24bpp 8-8-8
        // video streams. However, should be possible to construct images at
    // different bitdepths/pixel formats if we need to...
    LONG lFmtLength;
        AVIStreamFormatSize(m_pavi, 0, &lFmtLength);

        m_lpSrcFmt = (LPBITMAPINFOHEADER)MMmalloc(lFmtLength);
        memset (m_lpSrcFmt, 0, lFmtLength);

        AVIStreamReadFormat(m_pavi, 0, m_lpSrcFmt, &lFmtLength);
```

Once we have the source format, we can create a RealiMation image which matches the size of the source.

There is one point to note about the images which will be returned from the video stream. Windows allows images to either be 'top-down' (i.e., with the origin in the top-left corner of the image), or 'bottom-up' (with the origin in the lower-left corner). RealiMation assumes images are all 'top-down'. To tell which sort of image we are dealing with, we need to look at the biHeight field of the source format - a negative indicates a top-down image. If we get an image of the wrong direction, every time we extract an image from the video stream, we will need to flip it vertically. The source code for `CAVIObject` includes a static function '`InvertBitmap()`' which will do this for us.

The final stage of initialisation is to call `AVIStreamGetFrameOpen()`. This prepares the stream for getting the decompressed data. The second parameter is left as `NULL` - this means we are pulling the information out in the source format. We could use this parameter in order to extract the information in another format, matching the texture format desired by the display driver (see the section on optimising playback, later in this paper).

With everything set-up, the video stream is now ready for reading. Each time the video image needs to be updated, then a simple call to `CAVIObject::Update()` is all that is required.

The `CAVIObject` allows for playback in two distinct modes - '*RealTime*' and '*EveryFrame*'.

In the RealTime mode, then the `Update()` function will calculate which frame in the AVI file is required based on the `current_time` parameter, using the `m_SamplesPerSecond` calculated during the initialisation.

In the EveryFrame mode, then the `Update()` function ignores the `current_time` parameter, and simply advances to the next frame in the AVI stream each time it is called.

For both modes, the AVI will loop if the frame goes outside the range of defined frames in the file. The final check is to make sure we don't waste time and processing power by repeatedly extracting the same frame - we just store the last frame number extracted (in the `GetFrame()` function) and check the new frame number against the stored number:

```
        // Calculate which frame we need from the video stream
        switch (m_PlaybackMode)
       {
            case (EveryFrameAVI) :
               m_CurrentIndex++;
               break;
            case (RealTimeAVI)   :
            default              :
                m_CurrentIndex = (int)(m_SamplesPerSecond * current_time);
               break;
         }

          // Make sure we're not out of range (make the AVI loop)...
          m_CurrentIndex = m_CurrentIndex % m_lFrames;

          // Don't bother doing anything if the frame hasn't changed...
          if (m_CurrentIndex != m_LastIndex)
         {
             GetFrame (m_CurrentIndex);
         }
```

The work of extracting the frame information and transferring it down to the RealiMation libraries is done in the `GetFrame()` function. A simple call to `AVIStreamGetFrame()` gives you back a pointer to a `BITMAPINFOHEADER` structure :

```
  // Grab a pointer to a BITMAPINFOHEADER structure for the correct frame number
m_lpbi = (LPBITMAPINFOHEADER)AVIStreamGetFrame(m_gapgf, frame_number);
```

The actual bitmap data follows immediately after the header and palette information, and can be accessed with a little pointer arithmetic :

```
// The actual data is located after the BITMAPINFOHEADER structure and
// the (optional) RGBQUAD array which follows it...
LPBYTE lpBits = (LPBYTE)m_lpbi + (int)m_lpbi->biSize +
                (int)m_lpbi->biClrUsed*sizeof(RGBQUAD);
```

Now we have access to the bits, we can send them to the RealiMation library. If they are in the correct format, and don't need to be flipped, then the easiest option is just to point the `bitmap.pixel` member of the `rtImage` structure straight at the bits, and call `RTSetImage()` :

```
        m_ImageInfo.bitmap.pixel = lpBits;
        RTSetImage (m_VideoImage, &m_ImageInfo, TRUE);
```

The final 'TRUE' parameter is vital - it tells RealiMation that the actual pixel values have changed, and that the new image needs to be downloaded to the driver.

However, the image may need to be inverted before we can send it down to the driver, as described earlier. We cannot directly invert the bitmap returned to us by the `AVIStreamGetFrame()` function, since this is memory used by the AVI functions, and could affect further decoding of the AVI stream, so we need to copy the image into another block of memory before it is flipped.

```
        if (m_bFlipImage)
        {
            MMmemcpy(m_ImageInfo.bitmap.pixel, lpBits, Byte, m_ImageSize);
            InvertBitmap (&m_ImageInfo.bitmap, m_TempImageLine);
           RTSetImage (m_VideoImage, &m_ImageInfo, TRUE);
        }
```

Using the CAVIObject in an application

We can show a simple use of the `CAVIObject` by adding a single AVI texture to a RealiApp Wizard created application called 'RealiVideo'. The full source code for this is included on the latest version for the RealiMation CD, or is available for download from the RealiMation web-site.

We first need to add a member variable of type `CAVIObject *` to the class definition of `CChannel3D`, e.g. :

```
// Logo variables
BOOL m_bDrawLogo;
CRealiMationLogo *m_pLogo;

// Video object
CAVIObject *m_VideoObject;
```

and as part of the `CChannel3D` constructor, initialise the member to `NULL`.

As part of the `CChannel3D::RealiseChannelAndView()` function (after we have realised the channel and view), we look for a material with the name '*Video Material*', or an image with the name '*Video Image*'. (These names are entirely arbitrary, and have been chosen purely for the purpose of this demonstration. Any other method for identifying the material or image ID can be used in it's place.) If either of these is found, then the CAVIObject is created using the ID :

```
// Clean up and delete existing video object
if (m_VideoObject != NULL)
{
    m_VideoObject->FinishAVIfile();
    delete m_VideoObject;
    m_VideoObject = NULL;
}

// Look for video material
rtID id = RTGetIDFromName (“Video Material”);
if (id != RTNullID)
{
    m_VideoObject = new CAVIObject (id);
}
else
{
    id = RTGetIDFromName (“Video Image”);
    if (id != RTNullID)
        m_VideoObject = new CAVIObject (id);
}
```

Once the `m_VideoObject` has been created, all that remains is to use the `SetAVIFilename()` function to tell it which AVI file to use, and initialise everything. Here, we have added a `CString` member variable called `m_AVIfilename` to the `CRealiVideo` application class definition, holding the name of the AVI file to play, initialised to a suitable value in the `CRealiVideo` constructor. (The full source sample provides user-interface code to allow the AVI file to be changed during run-time.) :

```
if (m_VideoObject != NULL)
{
    if (!m_VideoObject->SetAVIFilename (theApp.m_AVIfilename))
    {
        delete m_VideoObject;
        m_VideoObject = NULL;
    }
}
```

With the `m_VideoObject` correctly setup, in order to make the AVI play, all we have to do is remember to update the object with the new time before we display the view :

```
// Draw the view on this channel.
if (m_bChannelOpen)
{
    if (m_VideoObject != NULL)
        m_VideoObject->Update (RTInqViewTime (m_ViewID));

    // Ensure there is a view to draw
```

```
        ASSERT(m_ViewID != RTNullID);

        // Draw a RealiMation view object
        RTDisplayView (m_ViewID);

       ...
```

The final stage is to make sure we clean everything up when every the channel is closed. There are two places this can happen - this first is the `CChannel3D::Close()` member function :

```
    void CChannel3D::Close(BOOL deleteChannel)
   {
       if (m_VideoObject != NULL)
      {
          m_VideoObject->FinishAVIfile();
          delete m_VideoObject;
          m_VideoObject = NULL;
      }
      ...
```

The second is when the user decides to load in a new RBS file - in this case, the `CChannel3D::SetView()` is called with `RTNullID` as the view parameter. When this happens, we also need to clean up the `m_VideoObject` object.

Finally, since the CAVIObject is using the Windows AVI functions, we need to include the `"vfw32.lib"` in the in the ‘Object/library modules’ field on the ‘Project | Settings | Link’ dialog.

The full source of the ‘RealiVideo’ demonstration can be found on the RealiMation CD, or is available for download from the RealiMation web-site. It also includes additional coding for allowing the user to alter which AVI file is played, using standard Windows/MFC user-interface code.

The sample code can (currently) only deal with video streams recorded at 24bpp (8-8-8 format). However, it should be possible to deal with streams recorded at other bit depths by creating a RealiMation image in the correct format.

Optimising Playback

Each display driver has it’s own ‘best format’ for textures. If you can arrange the video stream to match this format, then performance will improve.

- Take account of texture limitations of the driver - resizing bitmaps can significantly increase processing requirements, reducing frame rates. Many drivers want the texture dimensions to be a power of 2 - images which do not match these size requirements will be resized as they are downloaded. Since this will be done every time, it can represent a significant overhead. So, record the AVI (or resize it) to the best size for the driver.
- The `AVIStreamGetFrameOpen()` allows us to extract the frame information in a pre-defined pixel format. If we know what format the display driver is expecting, we could extract the image directly into the correct format.
- Mip-mapping can slow everything down (all levels will need to be regenerated for every frame of the AVI - the larger the image size, the more levels needed, and the greater the slow down). It is advisable to turn off mip-mapping for the image, as the example source code does when it creates an image.
- If the AVI is small enough, it might be worth reading individual images in during initialisation, and storing them separately in the correct format for the display driver you are using. Then, instead of extracting a frame from the AVI file each time, you would just need to point `m_ImageInfo.bitmap.pixel` to the correct block of memory for the relevant frame.
- If an application knows that particular AVI files will be used by particular object, then it should be possible to detect when these object move out of the visible view, and so not extract the frame data when it is not required.

Summary

The paper has shown how the video stream from a .AVI file can be used as a RealiMation texture image. The basic steps have been discussed, along with implementation details, and sample code has been supplied.

The techniques discussed here are not limited to AVI files, or to using the images for textures - you can easily use similar techniques to stream images from other sources, and use the images for other purposes such as 2D overlays or background images.

Another possible extension would be to make use of any audio stream in the AVI file, if present. The Win32 API also provides functions for extracting and playing these audio streams. Once playing, it would be possible to synchronise the video stream to the audio stream by inquiring the current time value from the audio stream as it is playing, then setting the video stream to match.

Appendix A : Source code for CAVIObject

```
////////////////////////////////////////////////////////////////////////////////
//
// Copyright © Datapath Ltd. 1998
//
// DESCRIPTION    :
//
//     CAVIObject - A class which controls playback of an AVI file to be
//                  used as a RealiMation texture
//

#include “stdafx.h”
#include “rfu.h”
#include “rmdebug.h”
#include “RMVideo.h”

/*————————————————————————————————————————*/
//
// Flips a bitmap vertically -
//    The temp_storage_cache points to a block which must be frame->pitch bytes
//    wide. If this is NULL, then the routine will allocate and deallocate
//    a large enough block of memory. However, using the temp_storage_cache
//    parameter can reduce the number of memory operations.
//

static void InvertBitmap (rtBitmap *frame, Byte *temp_storage_cache = NULL)
{
    // Inverts bitmap so AVI file appears the right way up
    Byte *temp_storage = temp_storage_cache;

    if (temp_storage == NULL)
        temp_storage = MMNewArray (Byte, frame->pitch);

    DWord half_frame_height = frame->height / 2;

    // Initialise to first and last line of the image
    Byte *pixels_top         = frame->pixel;
    Byte *pixels_bottom      = frame->pixel + (frame->height - 1) * frame->pitch;

    for (DWord j = 0; j < half_frame_height; j++)
    {
        // Place the top pixels in temp storage
        MMmemcpy (temp_storage,  pixels_top,    Byte, frame->pitch);
        MMmemcpy (pixels_top,    pixels_bottom, Byte, frame->pitch);
        MMmemcpy (pixels_bottom, temp_storage,  Byte, frame->pitch);

        pixels_top    += frame->pitch;
        pixels_bottom -= frame->pitch;
    }

    if ((temp_storage_cache == NULL) && (temp_storage != NULL))
        MMfree (temp_storage);
}

/*————————————————————————————————————————*/

////////////////////////////////////////////////////////////////////////////////
//
// A CAVIObject can be based on either a material or image.
//
CAVIObject::CAVIObject (rtID id)
{
    //  Make sure everything starts of nice an clean -
    //     no AVI file, everything initialised

    m_bInitialised = FALSE;
    m_pavi         = NULL;
    m_gapgf        = NULL;
    m_TempPixel    = NULL;

    // Default mode is to sync. the AVI
    m_PlaybackMode    = RealTimeAVI;
    m_VideoImage       = RTNullID;
    m_VideoMaterial    = RTNullID;
    m_PreviousImageID  = RTNullID;
    m_bCreatedImage    = FALSE;

    m_ImageSize = 0;  // Size, in bytes, of the image

    m_lFrames           = 0;
    m_CurrentIndex      = 0;
    m_LastIndex         = -1;
    m_SamplesPerSecond  = 1;
    m_lpSrcFmt          = NULL;
    m_bFlipImage        = FALSE;
    m_TempImageLine     = NULL;

    if (RTInqIDKind (id) == RTMaterialID)
    {
        // Record the fact we're using a material which
        // already exists, but are replacing the image it
        // uses as a texture
        m_VideoMaterial = id;

        rtMaterial mat_data;
        RTInqMaterial (id, &mat_data);

        // Store the previous image used by the material, so we
        // can replace it later
        if (mat_data.nimage > 0)
            m_PreviousImageID = mat_data.image[0];
        else
            m_PreviousImageID = RTNullID;

        // Create a new image to use as a video texture
        m_VideoImage = RTCreateID (RTImageID);

        mat_data.image[0] = m_VideoImage;
        if (mat_data.nimage == 0) mat_data.nimage = 1;

        RTSetMaterial (id, &mat_data);

        rtImage image_info;
        RTInqImage (m_VideoImage, &image_info);

        // Turn off mip-mapping - this can significantly slow
        // everything down, as the mip-maps will need to be
        // regenerated every frame
        image_info.hints &= ~RTMipmapIH;

        // Turn off the chroma colour
        image_info.chroma_red =
        image_info.chroma_green =
        image_info.chroma_blue = 256;

        RTSetImage (m_VideoImage, &image_info, True);
    }
    else if (RTInqIDKind (id) == RTImageID)
    {
        m_VideoMaterial = RTNullID;
        m_VideoImage = id;
    }
    else
    {
        RMASSERT (FALSE, "Trying to create CAVIObject with invalid type of id");
    }
}


/*________________________________________*/
// Actually pulls the frame information from the current AVI file.
// Everything must be initialised correctly before calling this
//
void CAVIObject::GetFrame (LONG frame_number)
{
    // Grab a pointer to a BITMAPINFOHEADER structure for the
    // correct frame number
    m_lpbi = (LPBITMAPINFOHEADER)AVIStreamGetFrame(m_gapgf, frame_number);

    // If we were successful
    if (m_lpbi != NULL)
    {
        // The actual data is located after the BITMAPINFOHEADER structure and
        // the (optional) RGBQUAD array which follows it...
        LPBYTE lpBits = (LPBYTE)m_lpbi + (int)m_lpbi->biSize +
                            (int)m_lpbi->biClrUsed*sizeof(RGBQUAD);

        // If we don't need to flip the image, then we can just point the
        // bitmap straight at the block of memory
        if (!m_bFlipImage)
        {
            m_ImageInfo.bitmap.pixel = lpBits;
        }
        else
        {
            // Otherwise, we copy the block of memory, then flip the image
            MMmemcpy(m_ImageInfo.bitmap.pixel, lpBits, Byte, m_ImageSize);
            InvertBitmap (&m_ImageInfo.bitmap, m_TempImageLine);
        }

        // Lastly, tell RealiMation we have new image data - the last 'True'
        // parameter says actually bitmap has changed, rather than just
        // the image properties
        RTSetImage (m_VideoImage, &m_ImageInfo, True);

        // Store the frame number we've just extracted - no point in extracting
        // it again next time...
        m_LastIndex = frame_number;
    }
}
/*----------------------------------------*/

//////////////////////////////////////////////////////////////////////////////
// Initialise the CAVIObject with the AVI filename. Sets up everything we
// need to read from the video stream, and reads the first frame.
//

BOOL CAVIObject::SetAVIFilename (CString &avi_filename)
{
    // If we already have an AVI file, then get rid of it...
    //   this will call AVIFileExit().
    FinishAVIfile ();

    // Initialise the AVI File library
    AVIFileInit();

    // Open a video stream from the AVI file (the files can also contain
    // other types of steams, eg audio, but we're currently not interested
    // in them).
    if (AVIStreamOpenFromFile(&m_pavi, avi_filename, streamtypeVIDEO,
                              0, OF_READ | OF_SHARE_EXCLUSIVE, NULL) != 0)
    {
        // Unable to open the video stream
        TRACE ("Error opening video stream from file : '%s' \n", avi_filename);

        // Close down cleanly, and set the initialised flag to FALSE
        AVIFileExit();
        m_bInitialised = FALSE;
    }
    else
    {
        // Video stream was successfully opened - now we need to get some
        // information from it about length, image format, etc...
        AVIStreamInfo(m_pavi, &m_avis, sizeof(m_avis));

        m_LastIndex    = -1;
        m_CurrentIndex = 0;
        m_lFrames      = AVIStreamLength(m_pavi);

        RMASSERT (m_avis.dwScale > Epsilon,
                  " Incorrect time scale in AVI file!");
```

```
     m_SamplesPerSecond = (Ord)m_avis.dwRate / (Ord)m_avis.dwScale;

    // Get the source format - currently can only deal with 24bpp 8-8-8
    // video streams. However, should be possible to construct images at
    // different bitdepths/pixel formats if we need to...
    LONG lFmtLength;
     AVIStreamFormatSize(m_pavi, 0, &lFmtLength);

     m_lpSrcFmt = (LPBITMAPINFOHEADER)MMmalloc(lFmtLength);
    memset (m_lpSrcFmt, 0, lFmtLength);

     AVIStreamReadFormat(m_pavi, 0, m_lpSrcFmt, &lFmtLength);

    // Need to construct bitmap at correct bit-depth for decoded image -
    //   at the moment, stick to 24bpp 8-8-8 images, and
    //   15/16 bpp 5-5-5,  nothing else will work!
    if ((m_lpSrcFmt->biBitCount == 24) ||
         (m_lpSrcFmt->biBitCount == 16) ||
         (l_lpSrcFmt->biBitCount == 15))
   {
       // Grab the current image information
       RTInqImage (m_VideoImage, &m_ImageInfo);

       // If there is already an image here, then we need to destroy it...
       RFDestroyBitmap (&m_ImageInfo.bitmap);

       // Construct a bitmap the same size as the source,
       // at 24bpp 8-8-8 format, or 15/16bpp 5-5-5 format, no alpha channel
       if ((m_lpSrcFmt->biBitCount == 16) || (m_lpSrcFmt->biBitCount == 15))
      {
           RFConstructBitmap (&m_ImageInfo.bitmap, m_lpSrcFmt->biWidth,
                              RMFabs(m_lpSrcFmt->biHeight),
                              m_lpSrcFmt->biBitCount,
                              0x7c00, 0x3e0, 0x1F, 0x00, False, True);
      }
      else
      {
           RFConstructBitmap (&m_ImageInfo.bitmap, m_lpSrcFmt->biWidth,
                              RMFabs(m_lpSrcFmt->biHeight),
                              m_lpSrcFmt->biBitCount,
                              0xff, 0xff00, 0xff0000, 0x00, False, True);
      }
       // Calculate the size (in bytes) of the image for later
        m_ImageSize = m_ImageInfo.bitmap.pitch * m_ImageInfo.bitmap.height;

       // Images returned can either be bottom-up (ie, origin is at
       // lower left corner of screen), or top-down (origin at top
      // left corner).
       // Whether the biHeight field is negative or positive tells
       // use which way up the image is (negative is top-down).

       // RealiMation expects textures to be top-down. So, if we get
       // an image which is bottom up, we need to take account of this.
       // There are a couple of methods which we can use :
      //
       // 1. After we have extracted the image from the AVI stream, we
       //    copy into a second block of memory and invert the image before
       //    calling RTSetImage(). This has the advantage of working
       //    correctly  with any faces in the RealiBase, but of course
       //    involves extra copying and processing of the image.
      //
       // 2. If we know exactly which faces are going to take video textures,
       //    we can alter the texture coordinates of those faces if required.
       //    This removes the extra processing.
      //
       // Of course, the best solution would be to have the image in the
       // AVI the correct way up for RealiMation - if you are recording the
       // AVI yourself, this may be possible.
      //

       m_bFlipImage = FALSE;
       m_TempImageLine = NULL;
       if (m_lpSrcFmt->biHeight > 0)
      {
          // Image is wrong way up for RealiMation
```

```
                m_bFlipImage = TRUE;
                m_TempImageLine = MMNewArray (Byte, m_ImageInfo.bitmap.pitch);
            }
            else
            {
                // Image is OK for RealiMation
                m_bFlipImage = FALSE;
            }

            // Store the pixel information we've just allocated -
            m_TempPixel = m_ImageInfo.bitmap.pixel;

        // Initialize video stream for getting decompressed frames
            m_gapgf = AVIStreamGetFrameOpen(m_pavi, NULL);
        if (m_gapgf != NULL)
            {
                // Fill in with information from the first frame - this
                // also calls RTSetImage with the new information
                GetFrame (0);
                m_LastIndex = -1;
                m_bInitialised = TRUE;
            }
            else
            {
                // Something happened to prevent us from opening the stream
                // so clean everything up, and report a failure
                if (m_TempImageLine != NULL)
                    MMfree (m_TempImageLine);
                m_TempImageLine = NULL;
                if (m_lpSrcFmt != NULL) MMfree (m_lpSrcFmt);
                m_lpSrcFmt = NULL;
                AVIFileExit();
                m_bInitialised = FALSE;
            }
        }
        else
        {
            if (m_lpSrcFmt != NULL) MMfree (m_lpSrcFmt);
            m_lpSrcFmt = NULL;
            AVIFileExit();
            m_bInitialised = FALSE;
        }
    }

    return m_bInitialised;
}
/*-----------------------------------------------------------------*/

/////////////////////////////////////////////////////////////////////////////
// We've finished using the AVI - close everything down, clean it up, and
// reset to the state we were in before,
//
void CAVIObject::FinishAVIfile ( )
{
    if (m_bInitialised)
    {
        // Close down the AVI stream and AVI file library
        AVIStreamGetFrameClose(m_gapgf);

        AVIStreamRelease(m_pavi);
        AVIFileExit();

        // Free up memory allocated
        if (m_TempImageLine != NULL)
            MMfree (m_TempImageLine);
        m_TempImageLine = NULL;
        if (m_lpSrcFmt != NULL) MMfree (m_lpSrcFmt);
        m_lpSrcFmt = NULL;
        m_bInitialised = FALSE;

        // Reset the image_info.bitmap.pixel back to the block of memory
        // we actually allocated with RFConstructBitmap - if it's been
        // changed to the area used for decompressing the AVI, then this
        // area will no longer be valid, and when RealiMation comes to
        // delete this ID, we will get an error...
        m_ImageInfo.bitmap.pixel = m_TempPixel;
        RTSetImage (m_VideoImage, &m_ImageInfo, True);


    }
}
/*————————————————————————————————————————*/

////////////////////////////////////////////////////////////////////////////////
// Destroy the CAVIObject, making sure everything is closed down first
//
CAVIObject::~CAVIObject ()
{
    // Make sure everything is closed down nicely
    FinishAVIfile();

    // Clean up any temporary ID’s we’ve created
    if (m_VideoMaterial != RTNullID)
    {
        rtMaterial mat_data;
        RTInqMaterial (m_VideoMaterial, &mat_data);

        if (m_PreviousImageID != RTNullID)
        {
            // Restore the original image used
            mat_data.image[0] = m_PreviousImageID;
        }
        else
        {
            mat_data.image[0] = RTNullID;
            mat_data.nimage   = 0;
        }
        RTSetMaterial (m_VideoMaterial, &mat_data);

        if (m_bCreatedImage)
            RTDeleteID (m_VideoImage);
    }

}
/*————————————————————————————————————————*/

////////////////////////////////////////////////////////////////////////////////
// Update the CAVIObject to the current time - reads image from the video
// stream.
void CAVIObject::Update (Ord current_time)
{
    // Update the image to match the current time

    if (m_bInitialised)
    {
        // Calculate which frame we need from the video stream
        switch (m_PlaybackMode)
        {
            case (EveryFrameAVI) : m_CurrentIndex++;
                                   break;
            case (RealTimeAVI)   :
            default              : m_CurrentIndex = (int)(m_SamplesPerSecond *
current_time);
                                   break;
        }

        // Make sure we’re not out of range (make the AVI loop)...
        m_CurrentIndex = m_CurrentIndex % m_lFrames;

        // Don’t bother doing anything if the frame hasn’t changed...
        if (m_CurrentIndex != m_LastIndex)
        {
            GetFrame (m_CurrentIndex);
        }
    }
}
/*————————————————————————————————————————*/

////////////////////////////////////////////////////////////////////////////////
// Update the CAVIObject to use a specific frame - reads image from the video
// stream.
void CAVIObject::SetCurrentFrame (LONG frame_number)
{
    // Update the current frame
    if (m_bInitialised)
    {
        m_CurrentIndex = frame_number;

        // Make sure we're not out of range (make the AVI loop)...
        m_CurrentIndex = m_CurrentIndex % m_lFrames;

        // Don't bother doing anything if the frame hasn't changed...
        if (m_CurrentIndex != m_LastIndex)
        {
            GetFrame (m_CurrentIndex);
        }
    }
}
```