Surround Video Help

Welcome to the Surround Video SDK help.
Copyright Notice
Quick Takes

Surround Video Help

Welcome to the Surround Video SDK help.
Copyright Notice
Quick Takes

Surround Video Help

Welcome to the Surround Video SDK help.
Copyright Notice
Quick Takes

Surround Video Help

Welcome to the Surround Video SDK help.
Copyright Notice
Quick Takes

- **About Surround Video**
- **Producing 360° Panoramic Photographs**
- **Preparing Surround Video Images with SVEdit**
- **Programming with the Surround Video API**
  - Surround Video Image and Document Concepts
  - A Brief Overview of Using the Surround Video API
  - Surround Video Interfaces
  - Getting the ISurround Interface Pointer
    - PanoramicSurroundFromDIB
    - PanoramicSurroundFromFile
    - PanoramicSurroundFromStream
    - PanoramicSurroundFromPartialStream
    - UpdateStreamLength
  - ISurround::ForceValid()
  - ISurround::GetBits()
  - ISurround::GetColors()
  - ISurround::GetDepth()
  - ISurround::GetExtents()
  - ISurround::GetHorizon()
  - ISurround::GetMaxViewSize
  - ISurround::GetView()
  - ISurround::ReadMore()
  - ISurround::SetBits()
  - Surround::SetHorizon()
  - ISurroundView::Draw()
  - ISurroundView::GetColors()
  - ISurroundView::GetDepth()
  - ISurroundView::GetSize()
  - ISurroundView::GetViewRange()
  - ISurroundView::GetZoom()
  - ISurroundView::SphereToView()
  - ISurroundView::ViewToSphere()

Surround Video Help

Welcome to the Surround Video SDK help.
Copyright Notice
Quick Takes

- About Surround Video
- Producing 360° Panoramic Photographs
- Preparing Surround Video Images with SVEdit
- Preparing Surround Video Images with LinkEdit

- Surround Video APIs

Quick Takes

- Be sure to read the list of known bugs in the ReadMe.
- Using Surround Video means doing several things including making panoramic photos, prepping them, and writing an app that can display them with SURROUND.DLL. See The Short Story on What to Do.
- SURROUND.DLL is a standard OLE object with ISurround and ISurroundView interfaces, but you must use a non-COM way to get the inteface pointer. See Getting the ISurround Interface Pointer.
- You can redistribute SURROUND.DLL; your Setup must also register the Surround Video OLE object. See Redistributing SURROUND.DLL.

What is Surround Video?

The Surround Video™ SDK is a collection of tools that developers can use to add 360 panoramic images to an application, as well as a runtime Internet control to allow the use of Surround Video Images in HTML documents or WEB pages. The Surround Video SDK brings the latest innovative technology to multimedia authors and the Internet. Surround Video will help developers create dynamic desktop and Internet-ready multi-media titles with support for progressive rendering, hot-spotting with URL links, and development of Internet and native multimedia titles.

There are four components to Surround Video SDK: The Surround Video Editor, which provides for the authoring of Surround Video Images; the Surround Video API, the runtime component responsible for image display; the Surround Video Link Editor, which provides for the authoring of Surround Video Images suitable for use on the Internet; and the Surround Video Internet Control, which is an Internet-aware OLE control.

This powerful tool provides the professional developer with a wide range of options in the creation of desktop-based or Internet-ready business and edutainment software. The creation of web titles encouraging complete freedom of movement in photo-realistic backgrounds - while maintaining realistic perspective - engages users to interact with your products and services from every angle.

See Also

- What's In the Kit?
- Panoramic Images
- The Surround Video Image Editing Tool (SVEDIT.EXE)
- The Surround Video Link Editing Tool (LINKEDIT.EXE)
- The Surround Video Internet Control (SVDIEO.EXE)
- The Surround Video Component Object and Engine
- How Image Correction Works
- Playback System Requirements
- Surround Video API
- The Short Story on What to Do
- Redistributing SURROUND.DLL

What's In the Kit?

The Surround Video Toolkit consists of

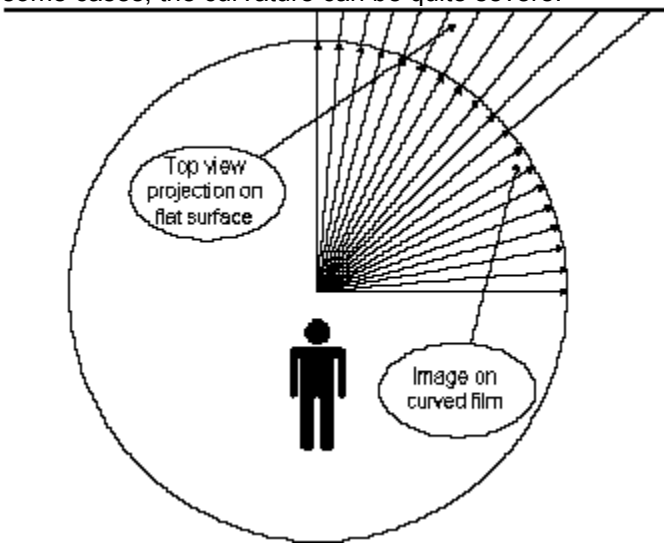
- Surround Video Prep tool (SVEdit.exe)
- Surround Video Component Object (SURROUND.DLL)
- Surround Video Library (SURROUND.LIB)
- Surround Video API Header file (SURROUND.H)
- Sample code (SVViewer and Hotspot)
- Sample Images
- Surround Video Internet control (SVIDEO.OCX)
- Sample HTML files and supporting images
- Surround Video Image Link Editor (LINKEDIT.EXE)

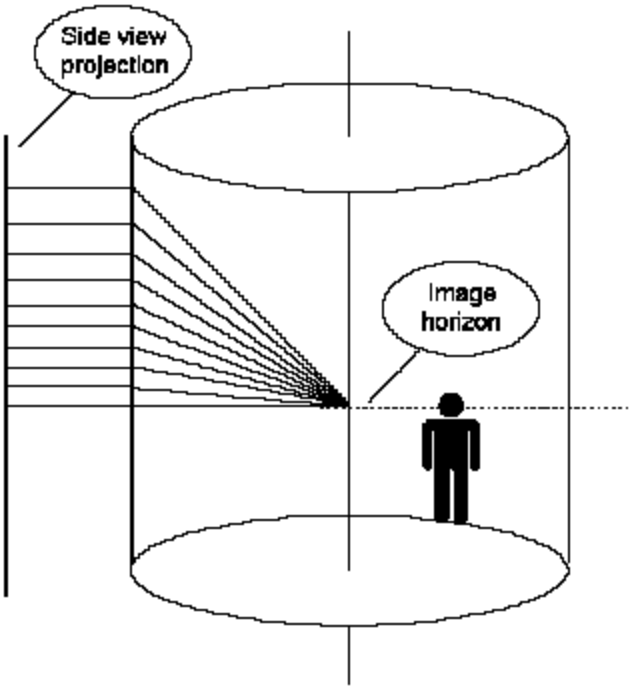
See Also

- Producing 360° Panoramic Photographs
- The Surround Video Image Editing Tool (SVEDIT.EXE)
- The Surround Video Link Editing Tool (LINKEDIT.EXE)
- The Surround Video Internet Control (SVIDEO.OCX)
- The Surround Video Component Object and Engine
- How Image Correction Works
- Playback System Requirements
- Surround Video API
- The Short Story on What to Do

Panoramic Images

Panoramic images have the drawback that they show the proper perspective only when viewed as a cylinder. When flattened out, features that would normally be seen as straight lines become warped; in some cases, the curvature can be quite severe.





Surround Video solves the image distortion problem through a very efficient mapping algorithm. The effect is similar to panning through a camera's viewfinder: the field of vision is narrower than that normally accorded to the human eye, but retains a strong sense of circular motion.

See Also

- Producing 360 Panoramic Photographs
- Surround Video Component Object and Engine
- How Image Correction Works
- Playback System Requirements

The Surround Video Image Editing Tool (SVEDIT.EXE)

Use SVEDIT.EXE to create Surround Video Images (.SVI files) from images taken with a 360-degree camera. See the Surround Video Image Editor help.

The Surround Video Link Editing Tool (LINKEDIT.EXE)

Use LINKEDIT.EXE to create Surround Video Link Images (.SVH files) from SVI files. See the Surround Video Link Editor help.

The Surround Video Internet Control (SVIDEO. OCX)

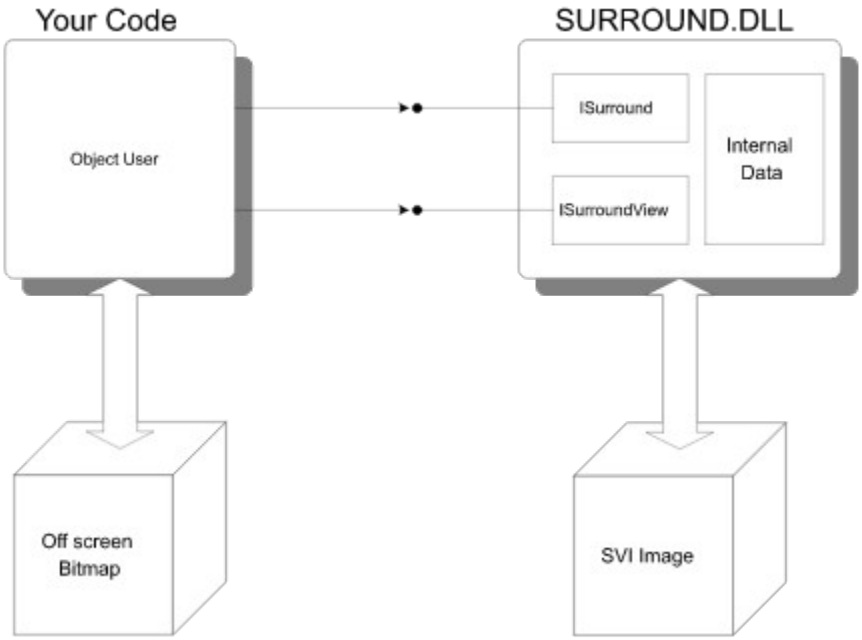
Use SVIDEO.OCX to view Surround Video Images (.SVH files) either off a local drive, or a URL internet site. See the Surround Video Internet Control help.

The Surround Video Component Object and Engine (SURROUND.DLL)

The Surround Video Engine is responsible for playing back Surround Video at run time. Its core technologies include:

- "On-the-fly" image correction. The user can turn correction on or off depending on the capabilities of machine on which the title is playing.
- Playing Windows-based digital video movies (AVI) anywhere on the current surround background while panning.
- Support for AVI digital video files that can contain "blue screen" backgrounds, that are painted transparently on the surround background.
- Opening a scene at any location within the surround background.
- Transition manager, responsible for ensuring quick transitions from scene to scene. This is especially useful when playing movies during scene transitions.
- Very efficient panning and custom blitting algorithms.

The Surround Video Component Object (SURROUND.DLL) contains the Surround Video Engine and is the heart of the Surround Video API. It is a single OLE object with two interfaces. Your application code uses these interfaces to interact with the Surround Video Image.



See Also

- What's in the Kit?
- Redistributing SURROUND.DLL

How Image Correction Works

Surround Video background images are taken with a panoramic (360 degree) camera. The geometry of a panoramic camera is significantly different from a standard camera in that all points along the length of the film are equidistant from the camera's focal point at the time of exposure. This, in effect, creates a cylindrical image that only appears correct when viewed from the exact center of the cylinder. When the film is "unrolled" onto a flat surface, such as a monitor, the image shows very noticeable distortion.
The specifics of the correction map are largely determined by the circumference of the surround image. However, there is another parameter affecting the correction map. The "horizon" of the image is the line that falls on the plane of the camera's rotation. In the original image this is in the exact vertical center of the film. However, after scanning and cropping, the horizon is unlikely to still fall in the center of the image. To correct for this, Surround Video lets the title developer adjust the horizon to indicate the original camera plane and produce the desired image correction.

See Also

- Panoramic Images
- Surround Video Engine
- Playback System Requirements

Playback System Requirements

We recommend as a base platform requirements for Surround Video titles:

Platform(DLL):	Windows® 95 or Windows NT™ 3.5
Platform (OCX):	The Surround Video Internet Control runs on Windows® 95 or Windows NT™ 4.0 beta. It requires IE3 Beta 2 for usage on a Web Page. If planning to host the OCX within some in-house container application, that container application must to able to host a minimal control as specified in the Microsoft© document entitled "OLE Control and Control Container Guidelines". This document is available as part of the Microsoft© ActiveX™ SDK.
Computer:	486 or higher; minimum 33 MHz and higher recommended
Memory:	8MB or more
Video:	VGA 256 colors 640X480, fast video recommended
Sound:	Any MPC sound card

Notes

- The Surround Video DLL will run on Windows 3.1 (with Win32s), but will be much slower and less engaging.
- A Macintosh runtime version of Surround Video is planned.

See Also

- Surround Video Engine

The Short Story on What to Do

Basically, to publish a Surround Video title you'll need to complete 4 steps:

1. **Make panoramic images.**

   Shoot or contract for panoramic pictures taken with a panoramic camera.
   See the help topics on making panoramic photographic images.

2. **Prepare Surround Video images with SVEdit.**

   Once you have scanned your panoramic images into DIB (.BMP) files, prep them with SVEdit to turn the DIBs into Surround Video Image (.SVI) files. The .SVI files contain horizon and origin data for the images.
   See the Surround Video Image Editor help.

3. **Add Surround Video capabilities to your player application.**

   SURROUND.DLL is an OLE server for Surround Video (ISurround) objects, which have a simple API. Use the Surround Video API to instantiate an ISurround object and display Surround Video panoramic images in your application's window.
   See the Surround Video API help and the SVViewer sample application (located in the \Samples\Svviewer subdirectory of the Surround Video installation directory).

4. **Add SURROUND.DLL to your title's Setup.**

   Your title's Setup should install and register SURROUND.DLL. See Redistributing SURROUND.DLL. Remember to ensure that your playback system includes any codecs you need to decompress your .SVI images.

Redistributing SURROUND.DLL

You have a royalty-free right to redistribute SURROUND.DLL with your application. See the LICENSE.TXT installed in your kit.
For information on how to make a custom Windows Setup program, see the Setup Toolkit for Windows which is included in the Windows Software Development Kit
Remember that if SURROUND.DLL is not present in target system your setup program will have to install SURROUND.DLL and register it in the registry. See the SURROUND.REG file installed with the kit.

Related Documentation

See the following sources for more information:

- *Microsoft Win32 SDK Programmer's Reference* (for information about structure types and OLE controls)
- *Microsoft OLE Control Developer's Kit User's Guide and Reference*

Producing 360 Degree Panoramic Photographs for Surround Video

Shooting still 360 degree panoramic images is not a difficult process, but there are a few issues to consider. The most challenging aspect is making sure that you have a good range of foreground and background objects that give perspective and a sense of depth to the image.

Whether you are planning a very specific multimedia application, such as a tour of a location like a museum or golf course or city, or you are planning to use the image as a backdrop to "blue screen" video characters, complete storyboards and careful staging will save you much time and frustration when you actually shoot the images.

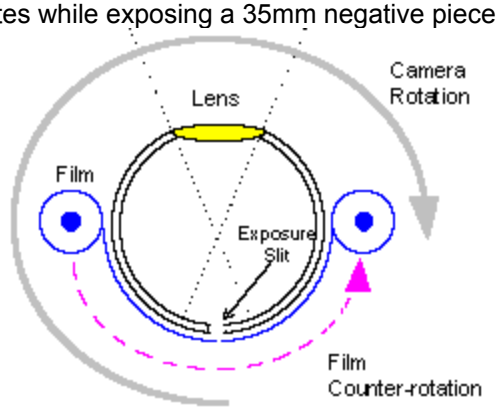
Information provided in this document is intended as general tips for producing 360-degree panoramic photographs, and assumes a basic understanding of still photography. This information is based on experience gained in producing a Surround Video demo at Microsoft and is intended only as guidelines.

See Also

- How Panoramic Cameras Work
- Creative Considerations
- Selecting an Approach
- Selecting the Film and Emulsion Type
- Setting Up to Shoot
- Shooting the Images
- Developing the Images
- Digitizing the Images
- Contacts for Panoramic Cameras and Photographers

How Panoramic Cameras Work

360-degree panoramic images are captured on film with cameras that rotate the lens "around" the film, resulting in a seamless image.
The Roundshot camera, which has a fixed aperture lens with a 38 degree vertical angle of exposure, rotates while exposing a 35mm negative piece of film that remains in relative position with the ground.



As the camera rotates clockwise, the film rotates counter-clockwise to stay perfectly steady with the ground. The film is exposed through a narrow vertical slit that results in a 10-inch wide continuous and steady exposure on the 35mm film.

Creative Considerations

It is always a good idea to carefully design and storyboard your application concept before you go out and shoot. When capturing images for Surround Video:

- Think in three dimensional space. Make sure you clearly define the point of view of the before you go out and shoot.
- Avoid panoramic scenes with lots of action. Remember, you are looking for "sets" for your title.
- If you are creating a visual map of a location, be consistent in the spacing between each shot (15-20 feet) and keep the height of the camera lens consistent.
- Consider action and blocking issues when you shoot the panoramic images. The panoramic images will largely provide a backdrop or environment for your main live action or animated characters to live within.

Selecting an Approach

Hire a Professional Panoramic Photographer

Many professional photographers specialize in shooting panoramas, and they have their own equipment and usually develop and print their own film.

When working with a professional photographer, be aware of copyright issues. Discuss up front complete ownership of the work for digital use. Most still photographers control and retain negatives and sell you the images for each additional type of use, even through you paid them to take the shots.

Not all panoramic photographers have full 360-degree cameras or have experience in shooting full 360-degree panoramic images. Make sure they have the right camera that will produce a single seamless piece of film covering 380 degrees (20 degrees minimum for the overlap).

Contact the International Association of Panoramic Photographers for referrals of professional 360-degree photographers, names of 360-degree cameras and list of book on panoramic photography.

Do It Yourself

To capture panoramic images yourself, you'll need these tools:

- A panoramic camera. These are available various price ranges, film formats, and degree capabilities. The cheaper ones are hand cranked and usually will not yield the most reliable images. Professional cameras are handmade and cost more than traditional single lens reflex cameras.
- A tripod. It should have quick release latches and the flexibility to shoot from about 2-3 feet above the ground to about 8 feet above the ground. The tripod should also have a swivel head, which will let you very quickly bubble-level the camera to make sure you always have a steady horizon line.
- A hand-held light meter. Avoid spot meters, which are expensive and designed for a telephoto lens or for large format photography with an accurate zone exposure system.

Selecting the Film and Emulsion Type

Other than standard photographic considerations for selecting the right film for various lighting conditions, there are several additional factors specific to shooting 360-degree panoramic images:

- Use 36 exposure film. You can then get five 380 degree panoramic images per roll of film.
- Use 100 ASA or lower. Do not use emulsion above 200 ASA, as you will need as much detail as possible once you digitize the image into the computer.
- Shoot slide film, which comes in 25, 64, and 200 ASA. Most digital film scanners are calibrated for positive rather than negative film, and slide film makes it easy to see your work immediately without having to get expensive contact sheets made.
- If you want high-resolution warm colors and you are not in a rush, you'll get better results with Kodachrome film. If you need to process the image in a day or in an hour, use Ektachrome film. For shooting interior with incandescent lighting, use tungsten 160 ASA slide film to correct for indoor lighting.

Setting Up to Shoot

Load the Film

Panoramic cameras have unique designs where the film often bends around a curved film plane. This can make loading film trickier than with regular cameras. It's a good idea to practice loading the film several times before you go out and shoot for the first time.
Take a short shot (120 degrees) immediately after loading the film to get rid of exposed film.

Level the Camera

It's very important to level the camera to insure a straight horizon line. (Most 360-degree cameras have built-in bubble levels.) Having a quick adjustment swivel head on the tripod makes leveling much easier.
Pay attention to the height of the camera. Eye level may not be the best point of view for your title. In fact, you might find that being close to the ground increases depth perception and reduces the amount of blue sky in the picture.
Also, use the view finder to see if you are cropping objects in the foreground unfavorably.

Shooting the Images

Aperture

When you shoot a 360-degree panoramic image, you have to deal with all lighting conditions available at the time of the shot. If you shoot a sunset, the sunset must look good and the shadows of the landscape behind you must also look good. Expose the film to take full advantage of its maximum F-stop latitude. There are two easy ways to accomplish this, depending of the film type you use.

- For slide film:
  Meter for *highlights* and close down the aperture by 1/2 a stop. Basically, you underexpose the film, giving you the most latitude on positive film.
- For negative film:
  Meter for *shadows* and open up the aperture by 1/2 a stop. Here you overexpose the film, providing the maximum range on negative film.

On overcast days, or for subjects with even lighting, you should either light for any regular picture or for the best middle gray.

Shutter Speed

Since the film is exposed gradually through a vertical slit, there is no actual shutter. By narrowing the slit you shorten the time of exposure on the film and therefore create an "effective" shutter speed.
Stick to 1/125 or 1/250 for most subjects to guarantee freezing any action that may occur during the 30 degree exposure
Use the slow speed to capture night scenes or event speed to expose objects or people multiple times on the same panorama. You can also create a continuous blur of someone walking in perfect sync with the lens motion at slow speed.

Focus and Depth of Field

You are dealing with a fixed focal wide-angle lens with no focusing, so there is no focus. However, you must be aware of depth of field and the minimum focal distance. You do not have to worry much about distant focus.

Selecting the Number of Degrees to Shoot

The 360-degree camera usually gives you control over how many degrees you expose.
Always overshoot by one segment, or 90 degrees, to make sure you have a good overlap to select your wrap around point once the image is digitized on the computer.
Shoot two segments at the beginning of each roll after loading the film to get rid of exposed film.

Developing the Images

The film is processed normally at any place of your choice, but you should make sure the film is NOT cut when it's processed. Protecting the film in a dust- and scratch-free environment is very important. Request your UNCUT rolls of film to be sleeved. If the service bureau cannot provide the sleeving service, most professional photographic supply stores sell clear uncut film sleeves that can accommodate an entire 36 exposure roll of film.
If you shot negative film, you will probably need to have custom contact proofs made. Cut your negative at the right places into 10 or 11 inch strips and give them to a professional photo finisher lab.

Digitizing the Images

Thirty-five mm images need to be scanned at very high pixel resolution to work effectively with the Surround Video technology. Because of the odd shape of the image on the film (9 to 11 inches wide) it's not easy to scan on traditional flatbed scanners, and it provides mediocre results.
To digitize the image yourself, you will spend from $20,000 to $75,000 to buy a high quality film scanner. Unless you are planning to generate 20 scans a week for several months, it's a good idea to use a service bureau.

Working with a Service Bureau

Two types of scanners are available that will deal effectively with the odd size image: drum scanners and large format transparency flatbed scanners. In both cases, the scanner should be capable of at least 1000 dots per inch (DPI) resolution.
Make sure the service bureau clearly understands how the images are going to be used, and specify exactly what you need:

- For best results, images should be scanned at 600 or 1000 DPI (height of positive or negative image should equal 600 DPI).
- Images should have 24-bit or 8-bit color depth. 24-bit gives you the flexibility to work with the image once it's scanned, but it means that each image is also going to be a very large file: 20-45 MB.
- Images should be delivered in TIFF format (most common among image processing software).
- Images should be oil-mounted on the scanner to provide for the best focus and remove minor scratches or dust particles.
- Image must be delivered on specific media. For example, a 240 MB floptical 3.5" disc. It's very important to agree in advance on a disc media that both you and the bureau can exchange images on.
- Ask them if you (or the photographer) can be there to supervise the color saturation during the scanning process.

Each image can vary from 6 MB to 45 MB in size, depending on the color depth and resolution you requested. Even though the images are very large, it's a good idea to ask for the best resolution so that you can archive the image and bring it down to the size you need on your own. However, make sure you have the right machine to be able to load and work with image easily. A 45 MB image can be very difficult to deal with on a low-end computer with small amount or RAM.
If the service bureau uses the drum scanner, you may have to trim the film to 9.5 inches wide to fit on the drum.
If you are going to give the service bureau negative film, be sure to ask in advance whether their scanners are calibrated for negative films. Most scanners are optimized for transparency/positive film and require the bureau to perform time-consuming calibration on the scanner to handle negative film.
Although programs like Adobe™ Photoshop™ let you change colors, it's very difficult to add colors that are not in the digital image in the first place. Most service bureaus either have a very good technical staff who could deliver high resolution sharp images (but poor color saturation), or an excellent colorist who delivers beautifully colored images that were technically poor in resolution quality or sharpness.

Contacts for Panoramic Cameras and Photographers

Interactive Association of Panoramic Photographers

Richard Fowler, 1995 Exec. Secretary/Treasurer, (407) 293-8003 (IAPPMAN@AOL.COM)
Chett Hanchett, 1995 President, (314) 781-3600 (V_PAN@AOL.COM)

360 Degree Panoramic Cameras

RoundShot by Seitz of Switzerland. 360 35mm slit camera available in several film sizes: 110, 35mm, 70mm, 220 and 5 inch. Motorized and battery powered. High quality, precision cameras, some with interchangeable lenses and variable rotation speeds.

Hulcherama by Charles Hulcher Co, of Hampton, VA. Model 120 slit camera is battery powered and accepts 120 or 220 roll film.

Alpha Roto 70 by Alpa Pignons S.A. of Switzerland. 360 slit camera battery powered by an electronically governed motor. Can take 70mm or 220 roll film.

Cirkut Camera by Folmer and Schwing Division of the Eastman Kodak Co. of Rochester NY. 360 slit cameras can handle several large format films.

Globuscope by Globuscope Inc. New York, NY. 360 35mm slit camera with fluid drive spring motor mechanism.

Corrales by Corrales Camera Whittier, CA. 360 35mm slit camera powered by mechanical spring. (Also sold as Spinshot 35mm Panoramic Camera.)

Contacts for Purchasing 360 Panoramic Cameras

Brooklyn Camera Exchange (718) 462-2892 (Cirkut)
Custom Panoramic Lab (407) 361-0031 (Roundshot)
Hulcherama (804) 245-6190
Ken Henson Imaging (407) 832-4844 (Roundshot)
Pro Photo (800) 732-6361
Skip Baldwin (617) 893-5611 (Roundshot)

Preparing Surround Video Images with SVEdit

You can find SVEdit in the \BIN directory of your Surround Video installation. You can use SVEdit to convert bitmap images (.BMP files) to the Surround Video image format (.SVI files) in both 8 and 24 bit formats, and also crop the image, correct distortion, and add compression.

Here are the basics of how to prepare an .SVI file with SVEdit. See the SVEdit help for details on how to use SVEdit.

Steps for Preparing the Surround Video Image (.SVI)

1. Open the panorama image (.BMP, .DIBs or .SVI).
2. Set the Image Field of View. Enter the number of degrees (0 to 360) for the image field of view.
3. Crop the image height as necessary.
4. Join 360-degree image ends by adjusting the image origin and cropping the image ends as necessary.
5. Correct image distortion as necessary.
6. Add compression if you want.
7. Save the image in the Surround Video image (.SVI) format.

Preparing Surround Video Images with LINKEDIT

You can find LinkEdit in the \BIN directory of your Surround Video installation. You can use LinkEdit to convert SVI files into the Surround Video Linked Image format (.SVH files).

Here are the basics of how to prepare an .SVH file with LinkEdit. See the LinkEdit help for details on how to use LinkEdit.

Steps for Preparing the Surround Video Linked Image (.SVH)

1. Open a Surround Video Image (.SVI).
2. Create links on the image with either drawing tool. Enter information about that link in the dialog.
3. Use “SaveAs” to save the .SVI file as a .SVH file.

Surround Video Image and Document Concepts

The Surround Video application programming interface (API) allows application programmers to control one or more viewports onto a Surround Video Image. This image is encapsulated in a Surround Video "document" that represents some portion of a spherical surface as it would be viewed from the center of a sphere.
The Microsoft® Surround Video Developer's Kit contains an authoring tool, SVEdit, for creating Surround Video images from digitized images (.BMP files) made from photographs taken by 360-degree panoramic cameras.
Click on the See Also button to jump to help on a specific function. You may also want to take a look at the A Brief Overview of Using the Surround Video API and the Surround Video Interfaces help topics.

Surround Video Document

A Surround Video Document, along with its runtime component, represents some portion of a spherical image. This image could be made from actual image data within the document, or by rendering engines that create the exposed surface at runtime. The final rendering of the image is independent of the API and is solely the responsibility of the runtime component that supports the document. When flattened, features in panoramic images can become severely warped. A Surround Video Document contains tables for correcting such distortion in the "compiled views."

Coordinate Space

The coordinate space in a Surround Video Document is expressed in latitude and longitude (measured from the center of the sphere), which enables the application programmer to create a viewport anywhere within the sphere.

A Brief Overview of Using the Surround Video API

Following is a brief once-through of how to create an Isurround object, initialize it, render an image, and display it using the Surround Video Interfaces. Please also look at the SVViewer sample code which shows how to open a file (.SVI or .BMP) and create an ISurround using the appropriate functions.

Create an ISurround Object and Initialize it

Before you can do anything with a Surround Video Image you must initialize OLE and instantiate an ISurround Object.

The first step is to initialize the OLE libraries. For example:

```
// Initialize OLE libraries
HRESULT hresult;

if((hresult = OleInitialize()) != NOERROR)
      return hresult;
```

In a C++ app this usually happens in ***CWinApp::InitInstance()***. The SVViewer sample app's ***CSVViewerApp::InitInstance*** calls ***AfxOleInit()***:

```
// Initialize OLE libraries
if (!AfxOleInit())
{
    AfxMessageBox(IDP_OLE_INIT_FAILED);
    return FALSE;
}
```

Next we instantiate an ISurround object using one of three C functions contained in the file SURROUND.LIB. Please look at the ***SVViewer*** sample application for an example of how to use the following functions.

- PanoramicSurroundFromDIB
- PanoramicSurroundFromFile
- PanoramicSurroundFromStream
- PanoramicSurroundFromPartialStream

Click on a name in the list to see help on that function.
You ***must*** use one of these functions to get the initial ISurround interface pointer. Applications cannot call CoCreateInstance to get an initial ISurround. See Getting the ISurround Interface Pointer.

For example, from the SVDOC.CPP source file of the SVVIEWER sample:

```
TRY
{
  // Make sure we at least have a Compound File
  hr = StgIsStorageFile( lpszPathName );
  if( hr == STG_E_FILENOTFOUND )
  {
    AfxThrowFileException(0);
  }
  else if( hr != S_OK )
  {
```

```
     pDib = (BITMAPINFOHEADER*)GlobalAllocPtr( GHND,
sizeof(BITMAPINFOHEADER) +
                            256*sizeof(RGBQUAD) );
     if( pDib == NULL )
       AfxThrowMemoryException();

     // See if it's a Dib we can treat as an Surround Video Image
     if( ReadDib( lpszPathName, (BITMAPINFO*)pDib, &pvBits ) != DIB_OK )
       AfxThrowFileException(0);

     // Add code here to
     hr = PanoramicSurroundFromDIB( pDib,
                    DibColors(pDib),
                    pvBits,
                    DibHeight(pDib) / 2,
                    MAX_ARCSECONDS,
                    &m_pISurround);

     if( FAILED(hr) ) AfxThrowFileException(0);

     return m_pISurround != NULL;
   }

   // open the storage with the image filename
   hr = StgOpenStorage(lpszPathName,
             NULL,
             STGM_READ | STGM_SHARE_EXCLUSIVE,
             0,
             0,
             &pIStorage);

   if( FAILED(hr) ) AfxThrowFileException(0);

   hr = pIStorage->OpenStream ( (LPOLESTR)CSVDOC_SURROUNDIMAGE,
                NULL,
                STGM_READ | STGM_SHARE_EXCLUSIVE,
                0,
                &pStream );

   // okay, so maybe we're using unicode.
   if( hr == STG_E_FILENOTFOUND )
   {
     wchar_t  wcstr[MAX_PATH * sizeof(OLECHAR)];
     size_t   count = mbstowcs( wcstr, CSVDOC_SURROUNDIMAGE, MAX_PATH);

     hr = pIStorage->OpenStream (   (LPOLESTR)wcstr,
                    NULL,
                    STGM_READ | STGM_SHARE_EXCLUSIVE,
                    0,
                    &pStream );
   }

   if( FAILED(hr) ) AfxThrowFileException(0);

   hr = PanoramicSurroundFromStream(pStream, 24, &m_pISurround);
```

```
    if (hr == SV_E_INCOMPATIBLE_SURFACE)
    {
     hr = PanoramicSurroundFromStream(pStream, 8, &m_pISurround);
    }

    if( FAILED(hr) ) AfxThrowFileException(0);
   }
   CATCH_ALL(e)
   {
    if( pDib != NULL )
     GlobalFreePtr( pDib);

    if( pvBits != NULL )
     GlobalFreePtr( pvBits );

    AfxMessageBox("The file could not be read.");
   }
   END_CATCH_ALL
```

We now have a valid ISurround Object which has been initialized. The next step is to get a view on it so we can look around.

Create an ISurroundView Object and Display it

Here we instantiate an ISurroundView object. The *Svviewer* sample app uses ***ISurround::GetMaxViewSize()*** determine the maximum size it can use to create a view.

```
SIZE maxViewSize;
float fZoom = 1.0f;                    // Zoom factor of the view (1:1)
int iViewQuality = 100;                // Quality setting of view (0 to 100)
DWORD dwFlags = SV_TOTAL_CORRECTION;   // Vertical and Horizontal correction

// Get the maximum size view we can create with this zoom factor
m_pISurround->GetMaxViewSize( fZoom, &maxViewSize );
```

Now we create the view.

```
HRESULT hr;

hr = pISurround->GetView( &maxViewSize, fZoom, iViewQuality,
                          dwFlags, &pISurroundView );
if( FAILED(hr) )
    ....error....
```

Now we determine what range of locations are valid for this view. The ***ISurroundView::Draw()*** function takes a location relative to the center of the viewport when rendering to your off screen Bitmap so we would like to know what locations are valid. We also need to know what the latitude at the center of the image.
Note that because many images are cropped after scanning Surround Video cannot assume that the center of each image will be exactly at the equator (0). This value for the center is set when the image is prepped with SVEdit.exe and is derived from the image's horizon.

```
SPHERE_RECT viewExtents;
SPHERE_POINT location;
HRESULT hr;

result = pISurroundView->GetViewRange( &viewExtents, &location.latitude );
```

Render the Image

At this point we have an ISurroundView object and we know the range of possible locations. All that is left to do is create an off screen Bitmap that is compatible with the view (it has the same size and color depth) and use ***ISurroundView::Draw()*** to render the portion of the image you wish to see. You will also need to create a palette based on the colors in the image.

```
SPHERE_POINT location;
BITMAPINFOHEADER* pDib;
VOID* pBits;
RECT viewRect;
int iDrawQuality = 100;    // Draw with maximum quality

hr = pISurroundView->Draw( &location, pDib, pBits, &viewRect, iDrawQuality );
```

Notice the iDrawQuality parameter. It is sets the draw quality of the image and ranges from 0 (lowest) to 100 (highest). It can be used to perform faster rendering while panning.

Surround Video Interfaces

The Surround Video API is based on Component Object Model (COM) interfaces, as defined in the *Microsoft Win32 SDK Programmer's Reference*.

ISurround

The **ISurround** interface provides detailed control of the Surround Video Document.

ISurroundView

The **ISurroundView** interface provides control over viewing the Surround Video Document.

Click on the See Also button to jump to help on a specific function. You may also want to take a look at A Brief Overview of Using the Surround Video API.

Getting the ISurround Interface Pointer

Getting a pointer to the ISurround interface is the starting point for every Surround Video application. To instantiate an ISurround object use one of these three C functions:

- PanoramicSurroundFromDIB
- PanoramicSurroundFromFile
- PanoramicSurroundFromStream
- PanoramicSurroundFrom PartialStream

Click on a name in the list to see help on that function.

These three functions will work from C, C++, and any other language that supports OLE, and are contained in the SURROUND.LIB static link library. The functions provide a simple non-COM way for an application to create a Surround Video object from an existing DIB (.BMP), Surround Video image (.SVI), or a Surround Video OLE stream. Each of these functions returns a pointer to the ISurround interface of the object created (in the ppISurround parameter).

You ***must*** use one of these functions to get the initial ISurround interface pointer. Applications cannot call CoCreateInstance to get an initial ISurround.

PanoramicSurroundFromDIB

HRESULT PanoramicSurroundFromDIB(LPBITMAPINFOHEADER lpbmi**,**
LPRGBQUAD lpColors**, LPVOID** lpBits**, int** iHorizon**, ARCSECONDS** extent**,**
Isurround ****ppISurround);**

The PanoramicSurroundFromDIB function creates an ISurround object, using the image data from the given DIB to initialize the object.

Parameters

lpbmi	Pointer to the DIB's BITMAPINFOHEADER.
lpColors	Pointer to the DIB's color table, an array of RGBQUADs. This parameter can be NULL for DIBs that do not have a color table. (e.g. 24-bit DIBs).
lpBits	Pointer to the DIB's data.
iHorizon	Horizon of image. The horizon controls Surround Video's image correction for cylindrical images. This is normally the center of the image (1/2 the image height) but may be some other value if the image has been cropped.
extent	Horizontal extent of the image in arcseconds. This should be set to MAX_ARCSECONDS (defined in surround.h) for 360-degree images.
ppISurround	Pointer to returned ISurround interface.

Return Value

Value	Meaning
S_OK	Success.
SV_E_INCOMPATIBLE_SURFACE	The image format is not supported.

Any HRESULT returned by QueryInterface() or CoCreateInstance().

See Also

- Getting the ISurround Interface Pointer
- PanoramicSurroundFromFile
- PanoramicSurroundFromStream
- PanoramicSurroundFrom PartialStream

PanoramicSurroundFromFile

HRESULT PanoramicSurroundFromFile(LPCTSTR lpszFilename**,**
UINT iDepthRequested**, Isurround ****ppISurround**);**
The PanoramicSurroundFromFile function creates an ISurround object based on a file. PanoramicSurroundFromFile expects the file to have been created by SVEdit or another application that supports the Storage/Stream naming convention used by SVEdit .

Parameters

lpszFilename	Filename (with path) of a Surround Video Image file (.SVI file).
iDepthRequested	Either 8 or 24, to request a bit depth of 8 or 24 bits respectively, that PanoramicSurroundFromFile is to open the SVI file with (Surround Video Images can be created to support both 8- and 24-bit-depth Isurround objects). If the requested bit depth is not available, PanoramicSurroundFromFile returns **SV_E_INCOMPATIBLE_SURFACE**.
ppISurround	Pointer to returned ISurround interface.

Return Value

Value	Meaning
S_OK	Success.
SV_E_INCOMPATIBLE_SURFACE	The image format is not supported.

Any HRESULT returned by QueryInterface() or CoCreateInstance().

See Also

- Getting the ISurround Interface Pointer
- PanoramicSurroundFromDIB
- PanoramicSurroundFromStream
- PanoramicSurroundFrom PartialStream

PanoramicSurroundFromStream

HRESULT PanoramicSurroundFromStream(IStream __RPC_FAR *pStream,
UINT iDepthRequested, Isurround **ppISurround);

The **PanoramicSurroundFromStream** function creates an ISurround object based on a stream. PanoramicSurroundFromStream expects the stream to contain data created by SVEdit or another application that supports the Storage/Stream naming convention used by SVEdit .

Parameters

pStream	Stream pointer as returned from IStorage::OpenStream(); the stream should contain Surround Video Image data.
iDepthRequested	Either 8 or 24, to request a bit depth of 8 or 24 bits respectively, that PanoramicSurroundFromFile is to open the SVI file with (Surround Video Images can be created to support both 8- and 24-bit-depth Isurround objects). If the requested bit depth is not available, PanoramicSurroundFromFile returns **SV_E_INCOMPATIBLE_SURFACE**.
ppISurround	Pointer to returned ISurround interface.

Return Value

Value	Meaning
S_OK	Success.
SV_E_INCOMPATIBLE_SURFACE	The image format is not supported.

Any HRESULT returned by QueryInterface() or CoCreateInstance().

See Also

- Getting the ISurround Interface Pointer
- PanoramicSurroundFromDIB
- PanoramicSurroundFromFile
- PanoramicSurroundFrom PartialStream

PanoramicSurroundFromPartialStream

HRESULT PanoramicSurroundFromStream(IStream __RPC_FAR *pStream,
UINT iDepthRequested, Isurround **ppISurround, UINT dwValidBytes, UINT dwOrigin);
The **PanoramicSurroundFromPartialStream** function creates an ISurround object based on a stream. PanoramicSurroundFromPartialStream expects the stream to contain data created by SVEdit or another application that supports the Storage/Stream naming convention used by SVEdit .

Parameters

pStream	Stream pointer as returned from IStorage::OpenStream(); the stream should contain Surround Video Image data.
iDepthRequested	Either 8 or 24, to request a bit depth of 8 or 24 bits respectively, that PanoramicSurroundFromFile is to open the SVI file with (Surround Video Images can be created to support both 8- and 24-bit-depth Isurround objects). If the requested bit depth is not available, PanoramicSurroundFromFile returns **SV_E_INCOMPATIBLE_SURFACE**.
ppISurround	Pointer to returned ISurround interface.
dwValidBytes	Number of valid bytes in the stream
dwOrigin	Offset of SVI data in stream

Return Value

Value	Meaning
S_OK	Success.
SV_E_INCOMPATIBLE_SURFACE	The image format is not supported.
SV_E_INSUFFICIENTBYTES	There is not enough data to render any part of image.

Any HRESULT returned by QueryInterface() or CoCreateInstance().

See Also

- Getting the ISurround Interface Pointer
- PanoramicSurroundFromDIB
- PanoramicSurroundFromFile
- PanoramicSurroundFromStream
- UpdateStreamLength

UpdateStreamLength()

HRESULT UpdateStreamLength(DWORD dwValidBytes**,**
BOOL FAR * pbUpdate**);**

The **UpdateStreamLength** function reads dwValidBytes data from the specified stream, and updates its internal stream which it uses to hold the SVI data. This call is made after a successful return from **PanoramicSurroundFromStream()**, and until all data has been retrieved for the current image.

Parameters

dwValidBytes	Number of valid bytes in the stream
pbUpdate	*pbUpdate is set to TRUE if new data has been read and needs to be painted to the screen (if NULL, nothing is returned)

Return Value

Value	Meaning
S_OK	Success.

Any HRESULT returned by QueryInterface() or CoCreateInstance().

See Also

Getting the ISurround Interface Pointer
PanoramicSurroundFrom PartialStream

See Also

ISurround::ForceValid()
ISurround::GetBits()
ISurround::GetColors()
ISurround::GetDepth()
ISurround::GetExtents()
ISurround::GetHorizon()
ISurround::GetMaxViewSize
ISurround::GetView()
ISurround::ReadMore()
ISurround::SetBits()
Surround::SetHorizon()
ISurroundView::Draw()
ISurroundView::GetColors()
ISurroundView::GetDepth()
ISurroundView::GetSize()
ISurroundView::GetViewRange()
ISurroundView::GetZoom()
ISurroundView::SphereToView()
ISurroundView::ViewToSphere()
ISurround Interface Pointer

ISurround::GetBits()

HRESULT GetBits(SPHERE_POINT *pPt, SIZE *pSize,
BITMAPINFOHEADER *pbmi, VOID *pvBits);

The **GetBits** function returns a rectangular bitmap of raw bits that make up the image at the specified location.

Parameters

pPt	Points to a **SPHERE_POINT** structure that defines the image location at the center of the viewport.
pSize	Points to a **SIZE** structure that defines the size of the viewport in pixels.
pbmi	Specifies the address of a **BITMAPINFOHEADER** structure containing bitmap size, format, and color data.
pvBits	Specifies a pointer to the destination bits.

Return Values

Value	Meaning
S_OK	Success.
SV_E_INCOMPATIBLE_SURFACE	Bit depth does not match in pbmi.
SV_E_INVALID_PARAMETER	Invalid parameter.
SV_E_RANGE	*pPt* lies outside the image.

Example

The following code example uses ***ISurround::GetBits()*** to create a DIB from a section of the image. At the present time, 256-color images (8 bit) have a full color table of 256 entries.

```
// CreateDibFromImage() - Create a DIB from a section of an ISurround image
BITMAPINFO* CreateDibFromImage( ISurround* pISurround, SPHERE_POINT*
pLocation, SIZE* pSize )
{
    BITMAPINFOHEADER* pDib;
    DWORD memSize;
    UINT depth;
    HRESULT hr;

    // Sanity check
    if( pISurround == NULL )
        return NULL;

    // Get the image depth
    depth = pISurround->GetDepth();

    // Calculate memory size needed the DIB
    memSize = sizeof( BITMAPINFOHEADER );

    // ...add in the size of the color table
    if( depth == 8 )
        memSize += 256 * sizeof( RGBQUAD );
```

```
    // ...and the size of the image
    memSize +=  WIDTHBYTES( pSize->cx * depth ) * pSize->cy;

    // Allocate memory for Header + Colors(if any) + bits
    // Note: GMEM_DDESHARE makes it sutible for passing to the clipboard
    pDib = (BITMAPINFOHEADER *)GlobalAllocPtr( GMEM_MOVEABLE|GMEM_DDESHARE,
memSize );

    // Create the header
    pDib->biSize = sizeof( BITMAPINFOHEADER );
    pDib->biWidth = pSize->cy;
    pDib->biHeight = pSize->cx;
    pDib->biPlanes = 1;
    pDib->biBitCount = depth;
    pDib->biCompression = BI_RGB;
    pDib->biSizeImage = WIDTHBYTES( pSize->cx * depth ) * pSize->cy;
    pDib->biXPelsPerMeter = 0;
    pDib->biYPelsPerMeter = 0;
    pDib->biClrUsed = ( depth == 8 ) ? 256 : 0;
    pDib->biClrImportant = pDib->biClrUsed;

    // Get color table
    if( depth == 8 )
    {
        hr = pISurround->GetColors( 0, 256, DibColors(pDib) );
        if( FAILED(hr) )
        {
            GlobalFreePtr( pDib );
            return NULL;
        }
    }

    // Get the bits
    hr = pISurround->GetBits( pLocation, pSize, pDib, DibPtr(pDib) );
    if( FAILED(hr) )
    {
        GlobalFreePtr( pDib );
        return NULL;
    }

    return (BITMAPINFO*)pDib;

}
```

Copy

Close
Print

ISurround::GetBits() Example

```
// CreateDibFromImage() - Create a DIB from a section of an ISurround image
BITMAPINFO* CreateDibFromImage( ISurround* pISurround, SPHERE_POINT*
pLocation, SIZE* pSize )
{
    BITMAPINFOHEADER* pDib;
    DWORD memSize;
    UINT depth;
    HRESULT hr;

    // Sanity check
    if( pISurround == NULL )
        return NULL;

    // Get the image depth
    depth = pISurround->GetDepth();

    // Calculate memory size needed the DIB
    memSize = sizeof( BITMAPINFOHEADER );

    // ...add in the size of the color table
    if( depth == 8 )
        memSize += 256 * sizeof( RGBQUAD );

    // ...and the size of the image
    memSize +=  WIDTHBYTES( pSize->cx * depth ) * pSize->cy;

    // Allocate memory for Header + Colors(if any) + bits
    // Note: GMEM_DDESHARE makes it sutible for passing to the clipboard
    pDib = (BITMAPINFOHEADER *)GlobalAllocPtr( GMEM_MOVEABLE|GMEM_DDESHARE,
memSize );

    // Create the header
    pDib->biSize = sizeof( BITMAPINFOHEADER );
    pDib->biWidth = pSize->cy;
    pDib->biHeight = pSize->cx;
    pDib->biPlanes = 1;
    pDib->biBitCount = depth;
    pDib->biCompression = BI_RGB;
    pDib->biSizeImage = WIDTHBYTES( pSize->cx * depth ) * pSize->cy;
    pDib->biXPelsPerMeter = 0;
    pDib->biYPelsPerMeter = 0;
    pDib->biClrUsed = ( depth == 8 ) ? 256 : 0;
```

```
    pDib->biClrImportant = pDib->biClrUsed;

    // Get color table
    if( depth == 8 )
    {
        hr = pISurround->GetColors( 0, 256, DibColors(pDib) );
        if( FAILED(hr) )
        {
            GlobalFreePtr( pDib );
            return NULL;
        }
    }

    // Get the bits
    hr = pISurround->GetBits( pLocation, pSize, pDib, DibPtr(pDib) );
    if( FAILED(hr) )
    {
        GlobalFreePtr( pDib );
        return NULL;
    }

    return (BITMAPINFO*)pDib;

}
```

See Also

SPHERE_POINT
HRESULT

ISurround::GetColors()

HRESULT GetColors(UINT iFirstEntry**, UINT** iNumEntries**,**
RGBQUAD FAR* pColors**);**
The **GetColors** function retrieves the color table for the Surround Video Document.

Parameters

iFirstEntry	Specifies the index of first color entry to return.
iNumEntries	Specifies the number of color entries to return.
pColors	Specifies a pointer to an array of **RGBQUAD** color entries.

Return Value

Value	Meaning
S_OK	Success.
SV_E_INCOMPATIBLE_SURFACE	Image does not have a palette.
SV_E_INVALID_PARAMETER	Invalid parameter.
SV_E_RANGE	*iNumEntries* specifies more colors than are present in the image.

Example

The following code example uses ***ISurround::GetColors()*** to create a DIB from a section of the image. At the present time, 256-color images (8 bit) have a full color table of 256 entries.

```
// CreateDibFromImage() - Create a DIB from a section of an ISurround image
BITMAPINFO* CreateDibFromImage( ISurround* pISurround, SPHERE_POINT*
pLocation, SIZE* pSize )
{
    BITMAPINFOHEADER* pDib;
    DWORD memSize;
    UINT depth;
    HRESULT hr;

    // Sanity check
    if( pISurround == NULL )
        return NULL;

    // Get the image depth
    depth = pISurround->GetDepth();

    // Calculate memory size needed the DIB
    memSize = sizeof( BITMAPINFOHEADER );

    // ...add in the size of the color table
    if( depth == 8 )
        memSize += 256 * sizeof( RGBQUAD );

    // ...and the size of the image
    memSize +=  WIDTHBYTES( pSize->cx * depth ) * pSize->cy;
```

```
    // Allocate memory for Header + Colors(if any) + bits
    // Note: GMEM_DDESHARE makes it sutible for passing to the clipboard
    pDib = (BITMAPINFOHEADER *)GlobalAllocPtr( GMEM_MOVEABLE|GMEM_DDESHARE,
memSize );

    // Create the header
    pDib->biSize = sizeof( BITMAPINFOHEADER );
    pDib->biWidth = pSize->cy;
    pDib->biHeight = pSize->cx;
    pDib->biPlanes = 1;
    pDib->biBitCount = depth;
    pDib->biCompression = BI_RGB;
    pDib->biSizeImage = WIDTHBYTES( pSize->cx * depth ) * pSize->cy;
    pDib->biXPelsPerMeter = 0;
    pDib->biYPelsPerMeter = 0;
    pDib->biClrUsed = ( depth == 8 ) ? 256 : 0;
    pDib->biClrImportant = pDib->biClrUsed;

    // Get color table
    if( depth == 8 )
    {
        hr = pISurround->GetColors( 0, 256, DibColors(pDib) );
        if( FAILED(hr) )
        {
            GlobalFreePtr( pDib );
            return NULL;
        }
    }

    // Get the bits
    hr = pISurround->GetBits( pLocation, pSize, pDib, DibPtr(pDib) );
    if( FAILED(hr) )
    {
        GlobalFreePtr( pDib );
        return NULL;
    }
```

Copy

Close
Print

ISurround::GetColors() Example

```
// CreateDibFromImage() - Create a DIB from a section of an ISurround image
BITMAPINFO* CreateDibFromImage( ISurround* pISurround, SPHERE_POINT*
pLocation, SIZE* pSize )
{
    BITMAPINFOHEADER* pDib;
    DWORD memSize;
    UINT depth;
    HRESULT hr;

    // Sanity check
    if( pISurround == NULL )
        return NULL;

    // Get the image depth
    depth = pISurround->GetDepth();

    // Calculate memory size needed the DIB
    memSize = sizeof( BITMAPINFOHEADER );

    // ...add in the size of the color table
    if( depth == 8 )
        memSize += 256 * sizeof( RGBQUAD );

    // ...and the size of the image
    memSize +=  WIDTHBYTES( pSize->cx * depth ) * pSize->cy;

    // Allocate memory for Header + Colors(if any) + bits
    // Note: GMEM_DDESHARE makes it sutible for passing to the clipboard
    pDib = (BITMAPINFOHEADER *)GlobalAllocPtr( GMEM_MOVEABLE|GMEM_DDESHARE,
memSize );

    // Create the header
    pDib->biSize = sizeof( BITMAPINFOHEADER );
    pDib->biWidth = pSize->cy;
    pDib->biHeight = pSize->cx;
    pDib->biPlanes = 1;
    pDib->biBitCount = depth;
    pDib->biCompression = BI_RGB;
    pDib->biSizeImage = WIDTHBYTES( pSize->cx * depth ) * pSize->cy;
    pDib->biXPelsPerMeter = 0;
    pDib->biYPelsPerMeter = 0;
    pDib->biClrUsed = ( depth == 8 ) ? 256 : 0;
```

```
pDib->biClrImportant = pDib->biClrUsed;

// Get color table
if( depth == 8 )
{
    hr = pISurround->GetColors( 0, 256, DibColors(pDib) );
    if( FAILED(hr) )
    {
        GlobalFreePtr( pDib );
        return NULL;
    }
}

// Get the bits
hr = pISurround->GetBits( pLocation, pSize, pDib, DibPtr(pDib) );
if( FAILED(hr) )
{
    GlobalFreePtr( pDib );
    return NULL;
}
```

See Also

HRESULT

ISurround::GetDepth()

UINT GetDepth( );

The **GetDepth** function returns the bit depth of the image, in bits pep pixel.

Return Value

The bit depth of the image, in bits per pixel.

Example

The following code example uses ***ISurround::GetDepth()*** to create a DIB from a section of the image. At the present time, 256-color images (8 bit) have a full color table of 256 entries.

```
// CreateDibFromImage() - Create a DIB from a section of an ISurround image
BITMAPINFO* CreateDibFromImage( ISurround* pISurround, SPHERE_POINT*
pLocation, SIZE* pSize )
{
    BITMAPINFOHEADER* pDib;
    DWORD memSize;
    UINT depth;
    HRESULT hr;

    // Sanity check
    if( pISurround == NULL )
        return NULL;

    // Get the image depth
    depth = pISurround->GetDepth();

    // Calculate memory size needed the DIB
    memSize = sizeof( BITMAPINFOHEADER );

    // ...add in the size of the color table
    if( depth == 8 )
        memSize += 256 * sizeof( RGBQUAD );

    // ...and the size of the image
    memSize +=  WIDTHBYTES( pSize->cx * depth ) * pSize->cy;

    // Allocate memory for Header + Colors(if any) + bits
    // Note: GMEM_DDESHARE makes it sutible for passing to the clipboard
    pDib = (BITMAPINFOHEADER *)GlobalAllocPtr( GMEM_MOVEABLE|GMEM_DDESHARE,
memSize );

    // Create the header
    pDib->biSize = sizeof( BITMAPINFOHEADER );
    pDib->biWidth = pSize->cy;
    pDib->biHeight = pSize->cx;
    pDib->biPlanes = 1;
```

```
    pDib->biBitCount = depth;
    pDib->biCompression = BI_RGB;
    pDib->biSizeImage = WIDTHBYTES( pSize->cx * depth ) * pSize->cy;
    pDib->biXPelsPerMeter = 0;
    pDib->biYPelsPerMeter = 0;
    pDib->biClrUsed = ( depth == 8 ) ? 256 : 0;
    pDib->biClrImportant = pDib->biClrUsed;

    // Get color table
    if( depth == 8 )
    {
        hr = pISurround->GetColors( 0, 256, DibColors(pDib) );
        if( FAILED(hr) )
        {
            GlobalFreePtr( pDib );
            return NULL;
        }
    }

    // Get the bits
    hr = pISurround->GetBits( pLocation, pSize, pDib, DibPtr(pDib) );
    if( FAILED(hr) )
    {
        GlobalFreePtr( pDib );
        return NULL;
    }
```

Copy

Close
Print

ISurround::GetDepth() Example

```
// CreateDibFromImage() - Create a DIB from a section of an ISurround image
BITMAPINFO* CreateDibFromImage( ISurround* pISurround, SPHERE_POINT*
pLocation, SIZE* pSize )
{
    BITMAPINFOHEADER* pDib;
    DWORD memSize;
    UINT depth;
    HRESULT hr;

    // Sanity check
    if( pISurround == NULL )
        return NULL;

    // Get the image depth
    depth = pISurround->GetDepth();

    // Calculate memory size needed the DIB
    memSize = sizeof( BITMAPINFOHEADER );

    // ...add in the size of the color table
    if( depth == 8 )
        memSize += 256 * sizeof( RGBQUAD );

    // ...and the size of the image
    memSize +=  WIDTHBYTES( pSize->cx * depth ) * pSize->cy;

    // Allocate memory for Header + Colors(if any) + bits
    // Note: GMEM_DDESHARE makes it sutible for passing to the clipboard
    pDib = (BITMAPINFOHEADER *)GlobalAllocPtr( GMEM_MOVEABLE|GMEM_DDESHARE,
memSize );

    // Create the header
    pDib->biSize = sizeof( BITMAPINFOHEADER );
    pDib->biWidth = pSize->cy;
    pDib->biHeight = pSize->cx;
    pDib->biPlanes = 1;
    pDib->biBitCount = depth;
    pDib->biCompression = BI_RGB;
    pDib->biSizeImage = WIDTHBYTES( pSize->cx * depth ) * pSize->cy;
    pDib->biXPelsPerMeter = 0;
    pDib->biYPelsPerMeter = 0;
    pDib->biClrUsed = ( depth == 8 ) ? 256 : 0;
```

```
pDib->biClrImportant = pDib->biClrUsed;

// Get color table
if( depth == 8 )
{
    hr = pISurround->GetColors( 0, 256, DibColors(pDib) );
    if( FAILED(hr) )
    {
        GlobalFreePtr( pDib );
        return NULL;
    }
}

// Get the bits
hr = pISurround->GetBits( pLocation, pSize, pDib, DibPtr(pDib) );
if( FAILED(hr) )
{
    GlobalFreePtr( pDib );
    return NULL;
}
```

See Also

ISurroundView::GetDepth()

ISurround::GetExtents()

HRESULT GetExtents(SPHERE_RECT *pExtent **);**

The **GetExtents** function returns the extents of the document.

Parameters

pExtent	Points to a **SPHERE_RECT** structure, that will receive the viewable extents of the document.

Return Value

Value	Meaning
S_OK	Success.
SV_E_INVALID_PARAMETER	*pExtent* is an invalid pointer.

See Also

SPHERE_RECT
HRESULT

ISurround::GetHorizon()

HRESULT GetHorizon(int *piHorizon **);**

The **GetHorizon** function returns the location of the image horizon.

Parameters

piHorizon Points to an int that will receive the horizon of the image.

Return Value

Value	Meaning
S_OK	Success.

See Also

HRESULT

ISurround::GetMaxViewSize()

HRESULT GetMaxViewSize(float fZoom**, SIZE** *pSize **);**

The **GetMaxViewSize** function is used to calculate the largest view that can be created for a given zoom factor.

Parameters

fZoom	Specifies the zoom factor of the image. For example, a value of 1.0 maps to a 1:1 zoom; a value of 2.0 maps to a 2:1 zoom.
pSize	Points to a **SIZE** structure that defines the size of the viewport in pixels.

Return Value

Value	Meaning
S_OK	An instance of the specified object class was successfully created.
SV_E_INVALID_PARAMETER	*pSize* is an invalid pointer.

Example

The following code example (derived from the SVViewer program example) uses ***ISurround::GetMaxViewSize()*** to show the initialization of ISurroundView.
Please note that variables starting with "m_" are data members of the CView class.

```
// InitSurroundView() - Initialize surround view
HRESULT CView::GetSurroundView()
{
    SIZE maxSize;
    SIZE viewSize;
    HDC hdc;
    CRect clientRect;
    HRESULT hr;

    if( pISurround == NULL )
        return FALSE;

    // Get client rect
    this->GetClientRect( clientRect );

    // Get the maximum view size we can get (zoom = 1:1)
    m_pISurround->GetMaxViewSize( 1.0f, &maxSize );

    // Calculate the view size we will create
    viewSize.cx = min( clientRect.Size().cx, maxSize.cx );
    viewSize.cy = min( clientRect.Size().cy, maxSize.cy );

    // Create the view
    hr = m_pISurround->GetView( &viewSize, 1.0f, 100, &m_pISurroundView );

    if( FAILED(hr) )
        return hr;
```

```
    // Get the view range
    m_pISurroundView->GetViewRange( &m_viewExtents, &m_location.latitude );

    // see if it's a 360 degree image...
    m_b360Image = ( m_viewExtents.left == 0  &&
                    m_viewExtents.right == MAX_ARCSECONDS ) ? TRUE : FALSE;

    // ...if not, then make sure we are in a valid location
    if( !m_b360Image )
    {
        if( m_location.longitude < m_viewExtents.left )
            m_location.longitude = m_viewExtents.left;
        else if( m_location.longitude > m_viewExtents.right )
            m_location.longitude = m_viewExtents.right;
    }

    // Delete any previous bitmap
    if( m_hOffscreenBitmap != NULL )
        DeleteObject( m_hOffscreenBitmap );

    // Create off screen bitmap and palette
    m_offscreenInfo.header.biBitCount = m_pISurroundView->GetDepth();

    hdc = ::GetDC( m_hWnd );

    if( m_offscreenInfo.header.biBitCount == 8 )
        m_hPalette = CreateIdentityPalette( 256 );

    m_offscreenInfo.header.biHeight = viewSize.cy;
    m_offscreenInfo.header.biWidth = viewSize.cx;
    m_hOffscreenBitmap = CreateDIBSection( hdc,
(BITMAPINFO*)&m_offscreenInfo,
                                           DIB_RGB_COLORS, &m_pOffscreenBits,
NULL, 0 );
    ::ReleaseDC( NULL, hdc );

    return S_OK;
}
```

ISurround::GetMaxViewSize() Example

```
// InitSurroundView() - Initialize surround view
HRESULT CView::GetSurroundView()
{
    SIZE maxSize;
    SIZE viewSize;
    HDC hdc;
    CRect clientRect;
    HRESULT hr;

    if( pISurround == NULL )
        return FALSE;

    // Get client rect
    this->GetClientRect( clientRect );

    // Get the maximum view size we can get (zoom = 1:1)
    m_pISurround->GetMaxViewSize( 1.0f, &maxSize );

    // Calculate the view size we will create
    viewSize.cx = min( clientRect.Size().cx, maxSize.cx );
    viewSize.cy = min( clientRect.Size().cy, maxSize.cy );

    // Create the view
    hr = m_pISurround->GetView( &viewSize, 1.0f, 100, &m_pISurroundView );

    if( FAILED(hr) )
        return hr;

    // Get the view range
    m_pISurroundView->GetViewRange( &m_viewExtents, &m_location.latitude );

    // see if it's a 360 degree image...
    m_b360Image = ( m_viewExtents.left == 0  &&
                    m_viewExtents.right == MAX_ARCSECONDS ) ? TRUE : FALSE;

    // ...if not, then make sure we are in a valid location
    if( !m_b360Image )
    {
        if( m_location.longitude < m_viewExtents.left )
            m_location.longitude = m_viewExtents.left;
        else if( m_location.longitude > m_viewExtents.right )
            m_location.longitude = m_viewExtents.right;
```

```
    }

    // Delete any previous bitmap
    if( m_hOffscreenBitmap != NULL )
        DeleteObject( m_hOffscreenBitmap );

    // Create off screen bitmap and palette
    m_offscreenInfo.header.biBitCount = m_pISurroundView->GetDepth();

    hdc = ::GetDC( m_hWnd );

    if( m_offscreenInfo.header.biBitCount == 8 )
        m_hPalette = CreateIdentityPalette( 256 );

    m_offscreenInfo.header.biHeight = viewSize.cy;
    m_offscreenInfo.header.biWidth = viewSize.cx;
    m_hOffscreenBitmap = CreateDIBSection( hdc,
(BITMAPINFO*)&m_offscreenInfo,
                                           DIB_RGB_COLORS, &m_pOffscreenBits,
NULL, 0 );
    ::ReleaseDC( NULL, hdc );

    return S_OK;
}
```

ISurround::GetView()

HRESULT GetView(SIZE *pSize**, float** fZoom**, int** iViewQuality**, DWORD** dwFlags**,**
ISurroundView FAR* FAR* ppView **);**
The **GetView** function creates an instance of ISurroundView and returns a pointer to it.

Parameters

pSize	Points to a **SIZE** structure that defines the size of the viewport in pixels.	
fZoom	Specifies the zoom factor of the image. For example, a value of 1.0 maps to a 1:1 zoom; a value of 2.0 maps to a 2:1 zoom.	
iViewQuality	An integer, from 0 to 100, that represents the desired image quality of the viewport.	
dwFlags	Correction flags that specify the method of image correction to use when rendering the image with ***ISurroundView::Draw()***.	
	SV_NO_CORRECTION	No image correction.
	SV_HORIZONTAL_CORRECTION	Image correction along the horizontal axis only
	SV_VERTICAL_CORRECTION	Image correction along the vertical axis only
	SV_TOTAL_CORRECTION	Image correction along both horizontal and vertical axes.
ppView	Specifies where to place the pointer for the requested interface. ppView will be NULL on error.	

Return Values

Value	Meaning
S_OK	An instance of the specified object class was successfully created.
SV_E_OUTOFMEMORY	Out of memory.
SV_E_INVALIDARG	One or more arguments are invalid.
SV_E_UNEXPECTED	An unexpected error occurred.

Example

The following code example (derived from the SVViewer program example) uses ***ISurround::GetView()*** to show the initialization of ISurroundView.
Please note that variables starting with "m_" are data members of the CView class.

```
// InitSurroundView() - Initialize surround view
HRESULT CView::GetSurroundView()
{
    SIZE maxSize;
    SIZE viewSize;
    HDC hdc;
    CRect clientRect;
    HRESULT hr;

    if( pISurround == NULL )
        return FALSE;

    // Get client rect
    this->GetClientRect( clientRect );
```

```
    // Get the maximum view size we can get (zoom = 1:1)
    m_pISurround->GetMaxViewSize( 1.0f, &maxSize );

    // Calculate the view size we will create
    viewSize.cx = min( clientRect.Size().cx, maxSize.cx );
    viewSize.cy = min( clientRect.Size().cy, maxSize.cy );

    // Create the view
    hr = m_pISurround->GetView( &viewSize, 1.0f, 100, SV_TOTAL_CORRECTION,
&m_pISurroundView );

    if( FAILED(hr) )
        return hr;

    // Get the view range
    m_pISurroundView->GetViewRange( &m_viewExtents, &m_location.latitude );

    // see if it's a 360 degree image...
    m_b360Image = ( m_viewExtents.left == 0  &&
                    m_viewExtents.right == MAX_ARCSECONDS ) ? TRUE : FALSE;

    // ...if not, then make sure we are in a valid location
    if( !m_b360Image )
    {
        if( m_location.longitude < m_viewExtents.left )
            m_location.longitude = m_viewExtents.left;
        else if( m_location.longitude > m_viewExtents.right )
            m_location.longitude = m_viewExtents.right;
    }

    // Delete any previous bitmap
    if( m_hOffscreenBitmap != NULL )
        DeleteObject( m_hOffscreenBitmap );

    // Create off screen bitmap and palette
    m_offscreenInfo.header.biBitCount = m_pISurroundView->GetDepth();

    hdc = ::GetDC( m_hWnd );

    if( m_offscreenInfo.header.biBitCount == 8 )
        m_hPalette = CreateIdentityPalette( 256 );

    m_offscreenInfo.header.biHeight = viewSize.cy;
    m_offscreenInfo.header.biWidth = viewSize.cx;
    m_hOffscreenBitmap = CreateDIBSection( hdc,
(BITMAPINFO*)&m_offscreenInfo,
```

```
                                                DIB_RGB_COLORS, &m_pOffscreenBits,
NULL, 0 );
    ::ReleaseDC( NULL, hdc );

    return S_OK;
}
```

ISurround::GetView() Example

```
// InitSurroundView() - Initialize surround view
HRESULT CView::GetSurroundView()
{
    SIZE maxSize;
    SIZE viewSize;
    HDC hdc;
    CRect clientRect;
    HRESULT hr;

    if( pISurround == NULL )
        return FALSE;

    // Get client rect
    this->GetClientRect( clientRect );

    // Get the maximum view size we can get (zoom = 1:1)
    m_pISurround->GetMaxViewSize( 1.0f, &maxSize );

    // Calculate the view size we will create
    viewSize.cx = min( clientRect.Size().cx, maxSize.cx );
    viewSize.cy = min( clientRect.Size().cy, maxSize.cy );

    // Create the view
    hr = m_pISurround->GetView( &viewSize, 1.0f, 100, SV_TOTAL_CORRECTION,
&m_pISurroundView );

    if( FAILED(hr) )
        return hr;

    // Get the view range
    m_pISurroundView->GetViewRange( &m_viewExtents, &m_location.latitude );

    // see if it's a 360 degree image...
    m_b360Image = ( m_viewExtents.left == 0  &&
                    m_viewExtents.right == MAX_ARCSECONDS ) ? TRUE : FALSE;

    // ...if not, then make sure we are in a valid location
    if( !m_b360Image )
    {
        if( m_location.longitude < m_viewExtents.left )
            m_location.longitude = m_viewExtents.left;
        else if( m_location.longitude > m_viewExtents.right )
```

```
            m_location.longitude = m_viewExtents.right;
    }

    // Delete any previous bitmap
    if( m_hOffscreenBitmap != NULL )
        DeleteObject( m_hOffscreenBitmap );

    // Create off screen bitmap and palette
    m_offscreenInfo.header.biBitCount = m_pISurroundView->GetDepth();

    hdc = ::GetDC( m_hWnd );

    if( m_offscreenInfo.header.biBitCount == 8 )
        m_hPalette = CreateIdentityPalette( 256 );

    m_offscreenInfo.header.biHeight = viewSize.cy;
    m_offscreenInfo.header.biWidth = viewSize.cx;
    m_hOffscreenBitmap = CreateDIBSection( hdc,
(BITMAPINFO*)&m_offscreenInfo,
                                           DIB_RGB_COLORS, &m_pOffscreenBits,
NULL, 0 );
    ::ReleaseDC( NULL, hdc );

    return S_OK;
}
```

ISurround::ForceValid()

HRESULT ForceValid(SPHERE_RECT *pExtent **)**

The **ForceValid** function is used to force a particular section of the image into memory.
When an ISurround image is opened, it is not read into memory until it is needed. This is normally handled automatically by the API, but **ISurround::ForceValid** can be used to force a section of image into memory. This "pre-loading" of the image can substantially improve performance, especially on CD-ROM based applications.

Parameters

pExtent — Points to a **SPHERE_RECT** structure, that is to be forced valid.

Return Values

Value	Meaning
S_OK	Success.
SV_E_INVALID_PARAMETER	*pExtent* is an invalid pointer.
SV_E_RANGE	*pExtent* does not lie within the image.

See Also

SPHERE_RECT
HRESULT

ISurround::ReadMore()

BOOL ReadMore(UINT nAmount **);**

The **ReadMore** function is used to allow background preprocessing of the image.
The ReadMore function is used to load more of the image into memory. The ReadMore function anticipates the next area of the image that may be needed. This function should be called from a thread in Win32 or in the message loop of a Win16/32s application.

Parameters

nAmount	Specifies the amount of processing to do before returning. The value, a percent of remaining image, ranges from 1 (lowest) to 100 (highest).

Return Value

If there is more image to be loaded, the return value is non-zero; otherwise it is zero.
At the present time, this function always returns TRUE.

ISurround::SetBits()

HRESULT SetBits(SPHERE_POINT *pPt, **SIZE** *pSize,
BITMAPINFOHEADER *pbmi, **VOID** *pvBits **);**

The **SetBits** function writes raw bits from a source bitmap to the document at the specified location.

Parameters

pPt	Points to a **SPHERE_POINT** structure that defines the image location at the center of the viewport.
pSize	Points to a **SIZE** structure that defines the size of the viewport in pixels.
pbmi	Specifies the address of a structure containing bitmap size, format, and color data.
pvBits	Specifies a pointer to the source bits.

Return Value

Value	Meaning
S_OK	Success.
SV_E_INCOMPATIBLE_SURFACE	pbmi is incompatible with the document image.
SV_E_INVALID_PARAMETER	Invalid parameter.
SV_E_RANGE	One or more arguments are out of range.

Comments

The parameters should be the same as a previous call to **ISurround::GetBits()** for the image to be correctly written to the document.

Example

```
void HiLight( SPHERE_POINT *pLocation, SIZE *pSize, ... )
{
    BITMAPINFOHEADER bmi;
    VOID *pBits;
    Isurround *pISurround;
    ....

    // See how big a DIB we need (fill in bmi)
    pISurround->GetBits(pLocation, pSize, &bmi, NULL );

    // Create the DIB (allocate the bits)
    CreateDIBSection( pWnd->GetDC(), &bmi, DIB_RGB_COLORS,
                          &pBits, 0, 0 );

    // Actually get the bits
    pISurround->GetBits(pLocation, pSize, &bmi, &pBits );

    //... CHANGE THE BITS ...

    // Put them back
    pISurround->SetBits(pLocation, pSize, &bmi, &pBits );

    ...
```

}

ISurround::SetBits() Example

```
void HiLight( SPHERE_POINT *pLocation, SIZE *pSize, ... )
{
    BITMAPINFOHEADER bmi;
    VOID *pBits;
    Isurround *pISurround;
    ....

    // See how big a DIB we need (fill in bmi)
    pISurround->GetBits(pLocation, pSize, &bmi, NULL );

    // Create the DIB (allocate the bits)
    CreateDIBSection( pWnd->GetDC(), &bmi, DIB_RGB_COLORS,
                          &pBits, 0, 0 );

    // Actually get the bits
    pISurround->GetBits(pLocation, pSize, &bmi, &pBits );

    ... CHANGE THE BITS ...

    // Put them back
    pISurround->SetBits(pLocation, pSize, &bmi, &pBits );

    ...

}
```

ISurround::SetHorizon()

HRESULT SetHorizon**(int** iHorizon **);**
The **GetHorizon** function sets the horizon of the image.

Parameters

iHorizon Specifies the horizon of the image.

Return Value

Value	Meaning
S_OK	Success.
SV_E_RANGE	iHorizon is out of range.

See Also

HRESULT

ISurroundView::Draw()

HRESULT Draw(SPHERE_POINT *pPt,
BITMAPINFOHEADER *pbmi, **VOID** *pvBits, **RECT** *pRect, **int** iDrawQuality**);**
The **Draw** function draws an image centered at the location specified by pPt on the surface defined by pbmi and pvBits. The size of the surface specified by pbmi must match the current size of the viewport.

Parameters

pPt	Points to a **SPHERE_POINT** structure that defines the image location at the center of the viewport.
pbmi	Specifies the address of a **BITMAPINFOHEADER** structure containing bitmap size, format, and color data of the surface.
pvBits	Specifies a pointer to the destination bits.
pRect	Points to a **RECT** structure that defines the area of the bitmap to be drawn. If NULL, the entire surface is assumed.
iDrawQuality	Specifies the draw quality. The value ranges from 0 (lowest) to 100 (highest) and sets the quality of the image that is returned. This value is normally set to 100, but lower quality images can be rendered faster. The draw quality can be changed to optimize drawing time when high quality images are not needed (such as, the panning of the background image).

Return Value

Value	Meaning
S_OK	Success.
SV_E_INCOMPATIBLE_SURFACE	*pbmi* is incompatible with the document image.
SV_E_INVALID_PARAMETER	Invalid parameter.
SV_E_RANGE	One or more arguments are out of range.

Example

The following code example uses ***ISurroundView::Draw()*** to draw the image into an off-screen bitmap and then to the screen.

```
void CView::OnDraw( CDC* pDC )
{
    HDC hdc;
    CRect imageRect;
    CRect clientRect;
    HRESULT hr;

    // Only do this if we have a view
    if( m_pISurroundView == NULL )
        return;

    // Get the HDC
    hdc = pDC->GetSafeHdc();

    // Select the palette
    HPALETTE hOldPalette = ::SelectPalette( hdc, m_hPalette, FALSE );
```

```
    ::RealizePalette(hdc);

    // imageRect is the size of the offscreen bitmap
    imageRect.SetRect( 0, 0,
                       m_offscreenInfo.header.biWidth,
                       m_offscreenInfo.header.biHeight );

    // Tell the view to draw into our offscreen bitmap
    hr = m_pISurroundView->Draw( &m_location, &m_offscreenInfo.header,
                                 m_pOffscreenBits, &imageRect, iDrawQuality);
    if( FAILED(hr) )
        return;

    // Get the size of the client rect
    GetClientRect( &clientRect );

    // NOTE: in practice, imageRect and clientRect should be the same.

    // Paint it to the screen
    StretchDIBits( hdc,
                   imageRect.left, imageRect.top,
                   imageRect.Width(), imageRect.Height(),
                   clientRect.left, clientRect.top,
                   clientRect.Width(), clientRect.Height(),
                   m_pOffscreenBits, (BITMAPINFO *)&m_offscreenInfo,
                   DIB_RGB_COLORS, SRCCOPY );
}
```

ISurroundView::Draw() Example

The following code example uses ***ISurroundView::Draw()*** to draw the image into an off-screen bitmap and then to the screen.

```
void CView::OnDraw( CDC* pDC )
{
    HDC hdc;
    CRect imageRect;
    CRect clientRect;
    HRESULT hr;

    // Only do this if we have a view
    if( m_pISurroundView == NULL )
        return;

    // Get the HDC
    hdc = pDC->GetSafeHdc();

    // Select the palette
    HPALETTE hOldPalette = ::SelectPalette( hdc, m_hPalette, FALSE );
    ::RealizePalette(hdc);

    // imageRect is the size of the offscreen bitmap
    imageRect.SetRect( 0, 0,
                       m_offscreenInfo.header.biWidth,
                       m_offscreenInfo.header.biHeight );

    // Tell the view to draw into our offscreen bitmap
    hr = m_pISurroundView->Draw( &m_location, &m_offscreenInfo.header,
                                 m_pOffscreenBits, &imageRect, iDrawQuality);
    if( FAILED(hr) )
        return;

    // Get the size of the client rect
    GetClientRect( &clientRect );

    // NOTE: in practice, imageRect and clientRect should be the same.

    // Paint it to the screen
    StretchDIBits( hdc,
                   imageRect.left, imageRect.top,
                   imageRect.Width(), imageRect.Height(),
                   clientRect.left, clientRect.top,
```

```
                    clientRect.Width(), clientRect.Height(),
                    m_pOffscreenBits, (BITMAPINFO *)&m_offscreenInfo,
                    DIB_RGB_COLORS, SRCCOPY );
}
```

See Also

SPHERE_POINT
HRESULT

ISurroundView::GetColors()

HRESULT GetColors(UINT iFirstEntry**, UINT** iNumEntries**, RGBQUAD FAR*** pColors **);**

The **GetColors** function retrieves color entries for the current viewport.

Parameters

iFirstEntry	Specifies the index of the first color entry to return.
iNumEntries	Specifies the number of color entries to return.
pColors	Specifies a pointer to an array of **RGBQUAD** color entries.

Return Values

Value	Meaning
S_OK	Success.
SV_E_INCOMPATIBLE_SURFACE	Image does not have a palette.
SV_E_INVALID_PARAMETER	Invalid parameter.
SV_E_RANGE	*iNumEntries* specifies more colors than are present in the image.

See Also

HRESULT

ISurroundView::GetDepth()

UINT GetDepth( );

The **GetDepth** function returns the bit depth of the image, in bits per pixel.

Return Value

The bit depth of the view in bits per pixel.

Example

The following example code uses ***ISurroundView::GetDepth()***. It is derived from the SVViewer sample code and shows how to initialize an ISurroundView.
Please note that variables starting with "m_" are data members of the CView class.

```
// InitSurroundView() - Initialize surround view
HRESULT CView::GetSurroundView()
{
    SIZE maxSize;
    SIZE viewSize;
    HDC hdc;
    CRect clientRect;
    HRESULT hr;

    if( pISurround == NULL )
        return FALSE;

    // Get client rect
    this->GetClientRect( clientRect );

    // Get the maximum view size we can get (zoom = 1:1)
    m_pISurround->GetMaxViewSize( 1.0f, &maxSize );

    // Calculate the view size we will create
    viewSize.cx = min( clientRect.Size().cx, maxSize.cx );
    viewSize.cy = min( clientRect.Size().cy, maxSize.cy );

    // Create the view
    hr = m_pISurround->GetView( &viewSize, 1.0f, 100, &m_pISurroundView );

    if( FAILED(hr) )
        return hr;

    // Get the view range
    m_pISurroundView->GetViewRange( &m_viewExtents, &m_location.latitude );

    // see if it's a 360 degree image...
    m_b360Image = ( m_viewExtents.left == 0  &&
                    m_viewExtents.right == MAX_ARCSECONDS ) ? TRUE : FALSE;
```

```
    // ...if not, then make sure we are in a valid location
    if( !m_b360Image )
    {
        if( m_location.longitude < m_viewExtents.left )
            m_location.longitude = m_viewExtents.left;
        else if( m_location.longitude > m_viewExtents.right )
            m_location.longitude = m_viewExtents.right;
    }

    // Delete any previous bitmap
    if( m_hOffscreenBitmap != NULL )
        DeleteObject( m_hOffscreenBitmap );

    // Create off screen bitmap and palette
    m_offscreenInfo.header.biBitCount = m_pISurroundView->GetDepth();

    hdc = ::GetDC( m_hWnd );

    if( m_offscreenInfo.header.biBitCount == 8 )
        m_hPalette = CreateIdentityPalette( 256 );

    m_offscreenInfo.header.biHeight = viewSize.cy;
    m_offscreenInfo.header.biWidth = viewSize.cx;
    m_hOffscreenBitmap = CreateDIBSection( hdc,
(BITMAPINFO*)&m_offscreenInfo,
                                           DIB_RGB_COLORS, &m_pOffscreenBits,
NULL, 0 );
    ::ReleaseDC( NULL, hdc );

    return S_OK;
}
```

ISurroundView::GetDepth() and ISurroundView::GetViewRange() Example

```
// InitSurroundView() - Initialize surround view
HRESULT CView::GetSurroundView()
{
    SIZE maxSize;
    SIZE viewSize;
    HDC hdc;
    CRect clientRect;
    HRESULT hr;

    if( pISurround == NULL )
        return FALSE;

    // Get client rect
    this->GetClientRect( clientRect );

    // Get the maximum view size we can get (zoom = 1:1)
    m_pISurround->GetMaxViewSize( 1.0f, &maxSize );

    // Calculate the view size we will create
    viewSize.cx = min( clientRect.Size().cx, maxSize.cx );
    viewSize.cy = min( clientRect.Size().cy, maxSize.cy );

    // Create the view
    hr = m_pISurround->GetView( &viewSize, 1.0f, 100, &m_pISurroundView );

    if( FAILED(hr) )
        return hr;

    // Get the view range
    m_pISurroundView->GetViewRange( &m_viewExtents, &m_location.latitude );

    // see if it's a 360 degree image...
    m_b360Image = ( m_viewExtents.left == 0  &&
                    m_viewExtents.right == MAX_ARCSECONDS ) ? TRUE : FALSE;

    // ...if not, then make sure we are in a valid location
    if( !m_b360Image )
    {
        if( m_location.longitude < m_viewExtents.left )
            m_location.longitude = m_viewExtents.left;
        else if( m_location.longitude > m_viewExtents.right )
            m_location.longitude = m_viewExtents.right;
```

```
    }

    // Delete any previous bitmap
    if( m_hOffscreenBitmap != NULL )
        DeleteObject( m_hOffscreenBitmap );

    // Create off screen bitmap and palette
    m_offscreenInfo.header.biBitCount = m_pISurroundView->GetDepth();

    hdc = ::GetDC( m_hWnd );

    if( m_offscreenInfo.header.biBitCount == 8 )
        m_hPalette = CreateIdentityPalette( 256 );

    m_offscreenInfo.header.biHeight = viewSize.cy;
    m_offscreenInfo.header.biWidth = viewSize.cx;
    m_hOffscreenBitmap = CreateDIBSection( hdc,
(BITMAPINFO*)&m_offscreenInfo,
                                           DIB_RGB_COLORS, &m_pOffscreenBits,
NULL, 0 );
    ::ReleaseDC( NULL, hdc );

    return S_OK;
}
```

See Also

ISurround::GetDepth()

ISurroundView::GetSize()

HRESULT GetSize(SIZE *_pSize_);

The **GetSize** function is used to return the size of the viewport in pixels. This is the viewport size that was created with ***ISurround::GetView()***.

Parameters

pSize	Address of a SIZE structure to receive the size of the view.

Return Values

Value	Meaning
S_OK	Success.
SV_E_INVALID_PARAMETER	*pSize* is not a valid pointer.

See Also

ISurround::GetView()

ISurroundView::GetViewRange()

HRESULT GetViewRange(SPHERE_RECT *pExtent,
ARCSECONDS FAR *pLatitudeCenter **);**
The **GetViewRange** function is used to set the range of the viewport.

Parameters

pExtent	Points to a **SPHERE_RECT** extents of the view.
pLatitudeCenter	Points to long (**ARCSECONDS**) that will receive the latitude of the image center.

Return Value

Value	Meaning
S_OK	Success.

Example

The following example code uses ***ISurroundView::GetViewRange()***. It is derived from the SVViewer sample code and shows how to initialize an ISurroundView.
Please note that variables starting with "m_" are data members of the CView class.

```
// InitSurroundView() - Initialize surround view
HRESULT CView::GetSurroundView()
{
    SIZE maxSize;
    SIZE viewSize;
    HDC hdc;
    CRect clientRect;
    HRESULT hr;

    if( pISurround == NULL )
        return FALSE;

    // Get client rect
    this->GetClientRect( clientRect );

    // Get the maximum view size we can get (zoom = 1:1)
    m_pISurround->GetMaxViewSize( 1.0f, &maxSize );

    // Calculate the view size we will create
    viewSize.cx = min( clientRect.Size().cx, maxSize.cx );
    viewSize.cy = min( clientRect.Size().cy, maxSize.cy );

    // Create the view
    hr = m_pISurround->GetView( &viewSize, 1.0f, 100, &m_pISurroundView );

    if( FAILED(hr) )
        return hr;

    // Get the view range
```

```
    m_pISurroundView->GetViewRange( &m_viewExtents, &m_location.latitude );

    // see if it's a 360 degree image...
    m_b360Image = ( m_viewExtents.left == 0  &&
                    m_viewExtents.right == MAX_ARCSECONDS ) ? TRUE : FALSE;

    // ...if not, then make sure we are in a valid location
    if( !m_b360Image )
    {
        if( m_location.longitude < m_viewExtents.left )
            m_location.longitude = m_viewExtents.left;
        else if( m_location.longitude > m_viewExtents.right )
            m_location.longitude = m_viewExtents.right;
    }

    // Delete any previous bitmap
    if( m_hOffscreenBitmap != NULL )
        DeleteObject( m_hOffscreenBitmap );

    // Create off screen bitmap and palette
    m_offscreenInfo.header.biBitCount = m_pISurroundView->GetDepth();

    hdc = ::GetDC( m_hWnd );

    if( m_offscreenInfo.header.biBitCount == 8 )
        m_hPalette = CreateIdentityPalette( 256 );

    m_offscreenInfo.header.biHeight = viewSize.cy;
    m_offscreenInfo.header.biWidth = viewSize.cx;
    m_hOffscreenBitmap = CreateDIBSection( hdc,
(BITMAPINFO*)&m_offscreenInfo,
                                           DIB_RGB_COLORS, &m_pOffscreenBits,
NULL, 0 );
    ::ReleaseDC( NULL, hdc );

    return S_OK;
}
```

See Also

ARCSECONDS
SPHERE_RECT
HRESULT

ISurroundView::GetZoom()

float GetZoom( );

The **GetZoom** function returns the zoom factor of the viewport.

Return Value

The zoom factor of the viewport.

ISurroundView::SphereToView()

HRESULT SphereToView(SPHERE_POINT *pSpt,
SPHERE_POINT *pSpTangent, POINT *pPt)

The **SphereToView** function converts spherical coordinates of the document to rectangular coordinates of the viewport.

Parameters

pSpt	Points to a **SPHERE_POINT** structure that specifies a location in the document.
pSpTangent	Points to a **SPHERE_POINT** structure that specifies the location where the plane of the viewport touches (is tangent to) the sphere of view.
pPt	Points to a **POINT** structure that will receive the converted coordinate.

Return Value

Value	Meaning
S_OK	Success.
SV_E_INVALID_PARAMETER	Invalid parameter.
SV_E_RANGE	One or more parameters are out of range.

ISurroundView::ViewToSphere()

HRESULT ViewToSphere(POINT *pPt, SPHERE_POINT *pSpTangent, SPHERE_POINT *pSpt)

The **ViewToSphere** function converts rectangular coordinates of the viewport to spherical coordinates of the document.

Parameters

pPt	Points to a **POINT** structure that specifies a location relative to the viewport.
pSpTangent	Points to **SPHERE_POINT** structure that specifies the location where the plane of the viewport touches (is tangent to) the sphere of view.
pSpt	Points to a **SPHERE_POINT** structure that will receive the converted coordinate.

Return Value

Value	Meaning
S_OK	Success.
SV_E_INVALID_PARAMETER	Invalid parameter.
SV_E_RANGE	One or more parameters are out of range.

See Also

SPHERE_POINT
HRESULT

ARCSECONDS

```
typedef long ARCSECONDS;
```

ARCSECONDS is a measurement of "Seconds of Arc" in polar coordinates. There are 360 degrees in a circle. Each degree is 60 minutes and each minute has 60 seconds. For example, the measurement of 20 16'50" would be equal to (20*60*60) + (16*60) + 50 = 73010 seconds.
The coordinate system for the document is as follows:
Latitude ranges from -90 0'0" (looking up) to +90
0'0" (looking down), or from -324000 to +324000 in **ARCSECONDS**.
Longitude ranges from 0 0'0" to 360
0'0", or from a value of **ARCSECONDS** of 0 to 1296000.

See Also

MAX_ARCSECONDS

MAX_ARCSECONDS

```
MAX_ARCSECONDS = 1296000
```

MAX_ARCSECONDS is the number of **ARCSECONDS** in a circle (360 degrees).

SPHERE_POINT

```
typedef struct _tagSPHERE_POINT
{
    ARCSECONDS latitude;
    ARCSECONDS longitude;
} SPHERE_POINT;
```

A **SPHERE_POINT** structure is a collection of **ARCSECONDS** for latitude and longitude. It defines a point in spherical coordinates that lie in the inside surface of the document.

See Also

ARCSECONDS

SPHERE_RECT

```
typedef struct _tagSPHERE_RECT
{
    ARCSECONDS left;
    ARCSECONDS top;
    ARCSECONDS right;
    ARCSECONDS bottom;
} SPHERE_RECT;
```

A **SPHERE_RECT** structure is a definition of a size of a viewport in "Seconds of Arc."

HRESULT

Error codes returned by the **ISurround** and **ISurroundView** interfaces:

Value	Meaning
SV_E_INVALID_MESSAGE	Invalid message parameter.
SV_E_INVALID_FLAG	Invalid flag specified.
SV_E_INVALID_ZOOM	Invalid zoom specified.
SV_E_INVALID_LOCATION	The specified location was not contained within the document.
SV_E_INVALID_PARAMETER	Invalid parameter specified.
SV_E_RANGE	Parameter specified was out of range.
SV_E_INCOMPATIBLE_SURFACE	The surface supplied was incompatible with the desired operation.
SV_E_BUG	Internal bug.

extent

An extent is the amount you can look around. In a 360-degree image, you have a 0 to 360 degree horizontal extent, but a much more limited vertical extent. With spherical images, you have a much broader vertical extent (360 degrees).

viewport

A sizable "window" (or view) of an image. A viewport is an instance of the **ISurroundView** interface.

SVViewer Program Example

The C++ SVViewer program example is located in \Samples\Svviewer of the Surround Video directory.