Scan-converting to a Bitmap

How to use bitmaps as the destination of the scan-conversion process, and how to avoid artefacts when using and rendering the bitmaps

Summary

Cambridge Pixel's SPx radar software can send scan-converted radar video in PPI and B-Scan formats to a graphical display or, alternatively, to a bitmap for use by application software.

Scan-conversion to a bitmap is often an effective generic method for receiving the image into an application, and can be used with normal Windows, X11, Open GL or any other graphics library.

However, it is important to handle the bitmap in a way that prevents conflict between the scan-conversion and rendering processes and the resulting image artefacts.

This application note briefly discusses the issues and their resolution, with references to example code.

The SPxScDestBitmap class

As shown in Figure 1, a scan conversion object such as `SPxScSourceLocal` or `SPxScSourceNet` can use the software class `SPxScDestBitmap` as the destination of the scan conversion process. This class uses a graphics-independent bitmap to hold the scan-converted video. The user's application can use the contents of the bitmap in a number of ways. These may include saving a sequence of scan-converted images to a file, converting them to an encoded video stream, performing image processing, or simply combining with other graphics sources to produce a composite image for display, potentially using a non-standard graphics display library.

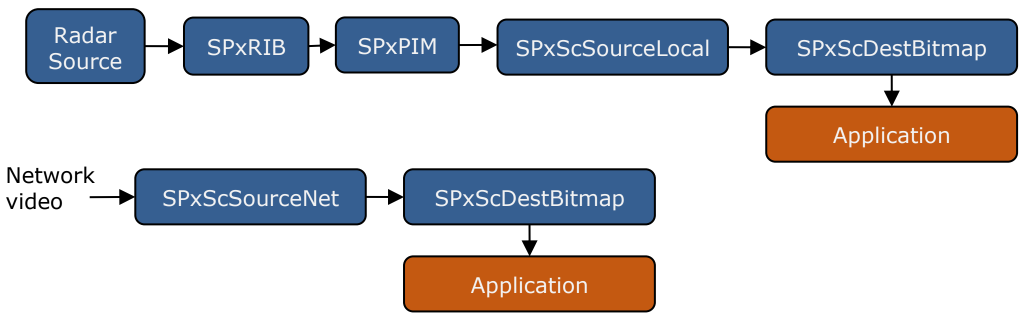

Figure 1 – Two typical configurations of SPx objects for radar acquisition and display application

The principal elements of the `SPxScDestBitmap` object are as follows.

Bitmap creation. `SPxScDestBitmap` can use either 8-bit or 32-bit bitmaps. Since the output from the scan converter is typically 8-bits per pixel, using an 8-bit bitmap as the destination means that the destination bitmap contains only intensity information, without any colour values. Adding colour to the radar image would be achieved within the user's application. However, if a 24-bit bitmap is used as the destination, each pixel is represented in ARGB format (alpha value and red/green/blue), and the functions `SetRadarColour()` and `SetRadarBright()` can be used to control the radar colour and brightness, with the function `SetRadarLUT()` providing full flexibility in mapping 8-bit to 32-bit values.

`SPxScDestBitmap` can allocate its own memory for holding the bitmap. However, it is also possible to supply the class with a pre-allocated bitmap. This can be useful, for example, if the use of a platform-specific format such as a Windows HBITMAP would make application programming more straightforward. In this case, the user can either create an HBITMAP directly or use the `SPxBitmapWin` class as a wrapper around an HBITMAP. A pointer to the memory allocated is then passed into `SPxScDestBitmap`.

Updates and notifications. The scan converter providing data to `SPxScDestBitmap` generates updates to the scan-converted view at a rate of approximately 50Hz (once every 20msec). Every time an update occurs, `SPxScDestBitmap` can notify its client, the application software, so that appropriate action can be taken. For a normal rotating radar, the updates will take form of a set of small patches clustered along the leading edge of the radar's sweep. The notification handler is provided with the dimensions and location of the bounding box that encloses the full set of patches generated by the scan converter since the last acknowledged update (the 'dirty box'). This information can be used by the application to copy only that part of the bitmap that has been changed.

Fading. The process of fading scan-converted radar video involves replacing each pixel in the bitmap with a smaller value. Three types of fading are supported by the SPx software. Sweep-based fading updates the entire scan-converted bitmap once per radar rotation. In replace mode, no fading is performed, and pixels are simply overwritten by the new radar video as each segment of the bitmap is updated. When real-time fading is selected, however, every pixel in the bitmap has to be updated at regular intervals so that the entire bitmap contents are faded correctly. In this mode, the application is responsible for calling the `FadeBitmap()` function on a timed basis to ensure that fading happens correctly. This should be not be done within the update notification handler, so that fading continues even in the absence of new radar video being supplied to the scan converter.

Handling the scan-converted bitmap

As discussed above, the contents of the scan-converted bitmap are regularly updated. The application software needs to make use of this data for downstream processing or rendering. The two methods discussed here are denoted synchronous and asynchronous.

Asynchronous. With this approach, the application code copies or processes scan-converted video from the bitmap without reference to the notification process that indicates to the application code that new data is available. This approach can yield acceptable results. However, a potential problem lies in the fact that the scan-converted video may be changed by the scan converter at the same time as it is being copied from the bitmap by the application. This is made even more likely since the asynchronous method does not have access to the dirty box indicating the changed part of the bitmap, so the entire bitmap must be copied each time. The asynchronous approach can result in visible artefacts in the radar video image when it is eventually rendered to a display, typically appearing as tearing in the image.

Synchronous. Within the notification handler registered with `SPxScDestBitmap`, the application code copies the scan-converted video from the bitmap into a temporary buffer, thereby ensuring that the video is not being updated during the copy. The dirty box can be used to reduce and thus optimise the transfer size. Independently of these updates, the application can retrieve the contents of the buffer for processing or rendering. A mutex lock is used to ensure that there is no conflict between the notification handler writing into the buffer and the application code that reads and uses the contents of the buffer. The application code keeps its own notion of a dirty box whose extent is increased each time the buffer is updated, allowing the application to select for processing only those parts of the buffer that have been updated.

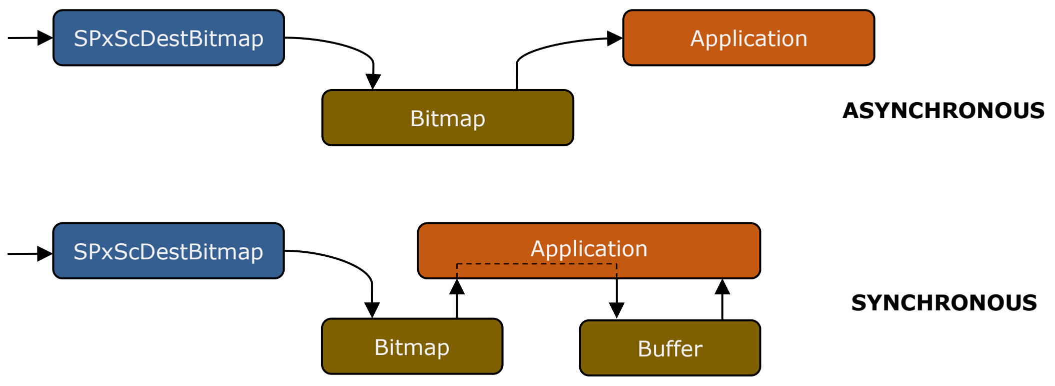

Figure 2 – Two approaches to handling the scan-converted bitmap

These two approaches are illustrated in the example solution `SPxWinBitmap`, which is supplied as part of the standard SPx developer's release. This example allows the user to select either asynchronous or synchronous operation and to observe the difference in behavior. The file **SPxWinBitmapDisplay.cpp** contains the primary routines for copying and using the bitmap. In asynchronous mode, the function `redrawWindowFromBitmap()` is used, and in synchronous mode the functions `CopyBitmapToOffscreen()` and `redrawWindowFromOffscreen()` are used.

< End of document >