

Building a Smart Online Video Application

by Robin Rowe

Dr. Dobb's Journal, December 1997

Many of today's applications involve Internet protocols, MPEG-1 video, JPEG images, ActiveX components, Netscape plug-ins, and other hard-to-master programming paradigms. Each of these technologies is complex, making it a challenge to build real-time systems that integrate all (or even part) of them.

In this article, we'll describe a technology we call a *Smart VCR* (SVCR) that integrates these disparate technologies. SVCR is a video-recording technology that enables real-time searches of broadcast television, thereby combining the ubiquity of existing TV channels with the convenient interactive interface of a Web browser. SVCR watches TV for you and can send you an e-mail notification when it captures something interesting to watch.

SVCR captures the closed-captioned text that is encoded in most U.S. television broadcasts and converts that text into program transcripts formatted in HTML. As it scans the closed-captioned text, SVCR searches for clues as to when video segments start and end. In the process, it stores the closed-captioning text into a real-time database and snaps JPEG thumbnail images of the streaming video to become icons and filmstrips. Finally, it captures MPEG-1 video clips at a nominal data rate of about 10 MB/minute (1.1 mbs).

The result is a Web page that lists TV news stories in chronological order, with links to the full transcripts, filmstrips of the video images, and hyperlinks to the actual video clips. A Netscape plug-in or cgi-bin program provides a browser interface to the transcript database, enabling users to keyword search the transcripts, retrieving associated images and video with any matches.



SVCR Overview

While a SVCR could be implemented in many ways, our design has four major hardware components:

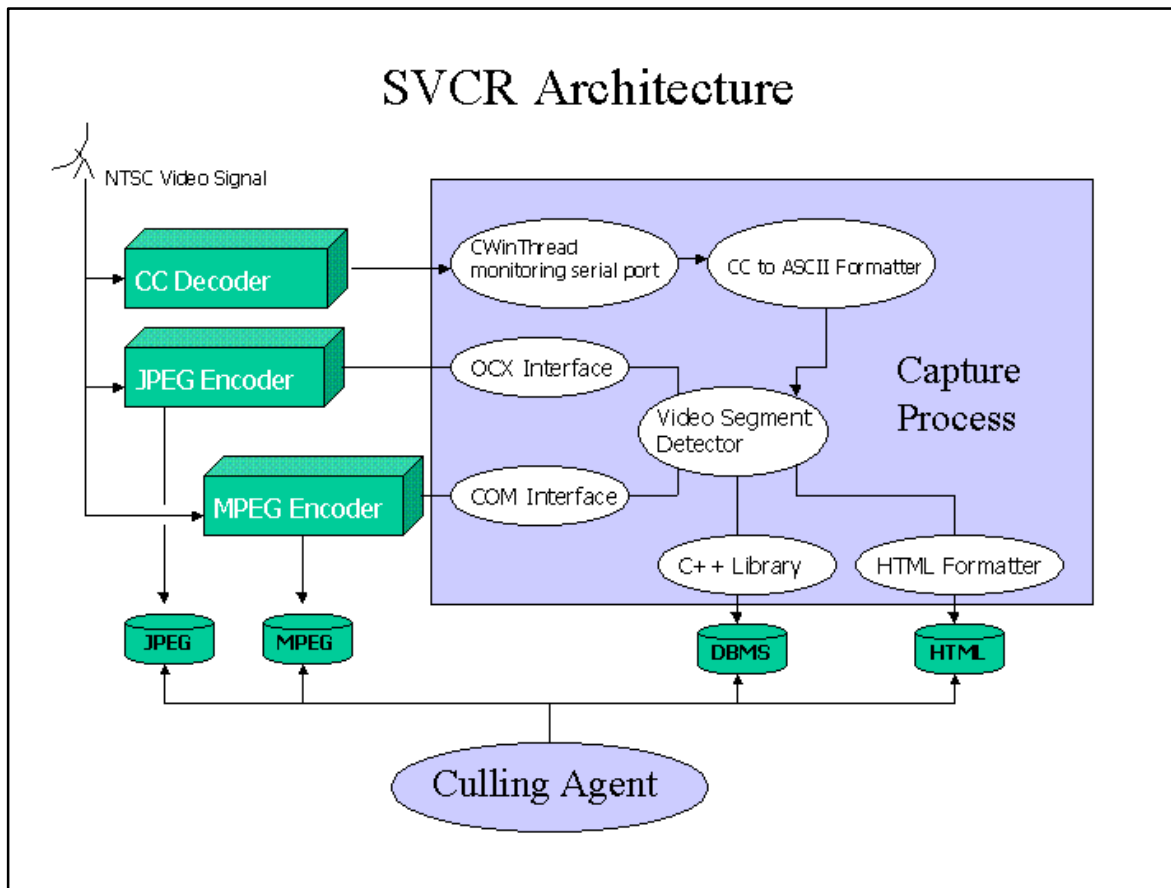
- A Pentium-based PC running Windows 95
- An MPEG-1 encoder card
- A JPEG encoder device
- A closed-captioning decoder box

Although its name suggests otherwise, a SVCR isn't necessarily a VCR. For several years the video post-production industry has been moving away from video tape to non-linear editing systems such as the Avid (<http://www.avid.com>) and Media 100 (<http://www.media100.com>). One advantage of non-linear is the ability to randomly access video clips. It's like the difference between finding a song on a CD and on a cassette tape--you never wait for non-linear to rewind or fast forward. In building an SVCR, our approach was to go digital, recording video directly to hard disk.

In the interest of real-time performance, all encoding and decoding operations are hardware assisted. Using commercial off-the-shelf (COTS) hardware, we wrote C++ code to interface with vendor-supplied Windows COM and OCX objects manipulating the hardware encoders. A separate Windows thread listens to the closed-caption text as it is delivered on the RS-232 port.

To distribute the load of actually serving the video, parts of SVCR were ported to DEC Alpha and Sun Sparc systems and compiled with GNU g++. The supported client platforms for viewing captured video includes Windows 95/NT, Macintosh, and the UNIX systems of DEC, Sun, HP, and SGI.

Additionally, we wrote a database system to optimize for minimum latency on writes. The goal was to maintain real-time capture performance on a Pentium 133. For improved performance on reads (that is, user queries), the data may be optionally saved across an Ethernet network using SAMBA to a UNIX box (such as a DEC AlphaStation running OSF) rather than to the local disk. (SAMBA is a suite of programs that lets clients access a server's file space and printers via SMB protocol; for more information, see [http://samba.canberra.edu.au/pub/samba/.](http://samba.canberra.edu.au/pub/samba/))



SVCR runs in real-time, either standalone in Windows 95 or in a distributed networked heterogeneous environment. It is both multi-process and multithreaded. Because Windows 95 supports a variety of commercial off-the-shelf graphics hardware, the code to handle the digitalization of video content was written for that operating system using MFC with components built in both Symantec C++ and Microsoft Visual C++.

Enter the Culling Agent

Because digital video can consume massive amounts of disk space in a short amount of time (MPEG-1 consumes about 2 GB every three hours), a *culling agent* periodically removes video that isn't of interest. Without a culling agent, you must create a large central data warehouse with row upon row of RAID storage. Even if you are willing to store content that way, delivering full

quality MPEG-1 video requires more network bandwidth than is available on the Internet today. Collecting the video content through the existing broadcast channels and saving it locally sidesteps this bandwidth bottleneck. However, the video warehouse approach is being pursued through the DARPA-sponsored Digital Library Initiative <http://www.aero.hq.nasa.gov/hpcc/cdrom/content/reports/annrpt96/iita/JRL.htm>.

Why not limit the recording of video segments at the time of capture, instead of culling later? Because this would return us to the same problem a human has when deciding whether or not to record a program. It doesn't usually become apparent that it is an interesting segment until it is already half over. Postponing culling leaves a window for users to change their mind about what's worth saving without consuming too much space.

A Serial Port Thread in MFC

Each frame of video that is closed-captioned contains two text characters even if they are just nulls (padding). The Telescriber closed-captioned decoder box is an off-the-shelf device primarily intended for the hearing impaired (see <http://www.viewcomtech.com/>). The Telescriber reads the closed-captioning hidden in the retrace interval of line 21 of the video signal and outputs it into a standard serial port. The first step in developing SVCR was to integrate this device by writing our own program to listen to it.

The close-captioned text transmitted in a TV signal isn't ASCII, but its own code that is set down by FCC rule 91-119, mod 92-157. (It can be found in the Code of Federal Regulations Title 47, Part 15, Section 119.) Stripping off the top bit of close-captioned text converts it to a rough approximation of ASCII. It's only a first approximation because the close-captioned text contains formatting codes in addition to data. It also has a somewhat different symbol set; for example a musical note to represent singing.

To monitor the serial port in Windows, you start a separate thread of execution devoted to this task. (Code to control the serial port in Windows is widely available. See, for instance, *Programming Windows 95 Unleashed*, edited by Randall A. Tamura, Sams, 1995.) The fundamental approach is to use a *CWinThread* in conjunction with Windows OVERLAPPED event objects.

In Windows 95, the serial port is opened using *CreateFile()* and treated as an asynchronous file. Windows NT supports asynchronous files (which read or write in the background), but Windows 95 does not. Windows 95 only supports them if the "file" is actually a port. The serial port thread fills a buffer with data while the main thread empties it. The serial thread notifies the main thread that data is waiting by using *view->PostMessage()*.

Using COM, OLE, and OCX

SVCR talks to two other hardware devices--a Snappy JPEG image capture device (<http://play.com>) that hangs off the parallel port and a Darim MPEGator MPEG-1 encoder card (<http://www.darim.com>) installed in a PCI slot. The Snappy includes an API controlled through an OCX. The MPEGator is controlled through a COM interface. Although an OCX is a type of COM, programming to these two interfaces is quite different.

COM is an underlying protocol that allows Windows applications to interface with objects that are external to a program. To a C++ programmer this means that you get a pointer to an object in an external program, with all the power that implies. ActiveX and OCX controls are standard interfaces that use COM. Similar in concept to abstract base classes, these interfaces are the methods that a particular type of COM class must provide.

ActiveX was formerly known as OLE, and you will still find references in documentation and elsewhere of OLE. The OCX interface evolved from the VBX control. It's a COM interface intended for use by Visual Basic, but available to other languages too since it is just another kind of COM. ActiveX is a more lightweight interface than OCX (and is consequently more popular for writing ActiveX controls for downloading via the internet).

Look Ma, No Libs!

Not long ago, to communicate with a vendor's proprietary hardware you made some calls into their programming API and linked your code with their C library that did the low level control of their device. More recently, an enhancement came in the form of Windows DLL's (Dynamic Link Libraries). You don't have to link statically anymore because you can load the (vendor's) DLL at runtime. The Windows COM interface, of which ActiveX and OCX are examples, goes a step further. The COM mechanism lets you retrieve the vendor's library as though it was a C++ object. The COM object is actually an executable or DLL, but that detail is hidden.

You can retrieve a COM object either by its CLSID number (it's unique Windows ID number generated at compile time) or by name (that looks up the ID in the Windows registry). The compiler's class wizard generates MFC code to make this operation invisible to the programmer.

Since ActiveX executables are smaller than OCX it would be preferable in a real time program to use ActiveX. Unfortunately, the Snappy API only supports using their OCX for capturing JPEG images. Snappy offers a COM interface, but that only handles raw bitmaps. It was also a disappointment that SC++ 7.5 (the Windows C++ compiler we use most) doesn't support controlling an OCX, although it does support creating an OCX. As a workaround we took the Snappy OCX and wrapped it inside a new ActiveX control using Visual

C++. This, in turn, was controlled easily from our main program compiled in SC++.

Working with an OCX or ActiveX control is almost trivial with C++ compiler support. Using the class wizard, the compiler generates a class with the code to handle the COM communication. Using an object of this class you can drive the external control as you would if it was an object compiled into your own code.

The MPEGator is a popular MPEG-1 video capture card. A high-quality MPEG-1 datastream consumes 10 MB/min, a low-resolution stream about 2 MB/min. The MPEGator also supports AVI, but that consumes more space for the same quality.

Database Design

We considered using a COTS database such as Excite (<http://www.excite.com>) that is designed for indexing Web pages. However, none of the Web database engines seemed suitable for a real-time system. Without a real-time database, segments captured would not become immediately available. They would be held up waiting for the database to index them in a batch process. The benefit of a custom real-time database is that access to stories is only limited by the latency of system buffers in saving the data. Stories become available for search and retrieval within seconds, even while still in the process of being captured. The indices are maintained in real time.

Commercial databases are designed for generic uses, not for optimal speed on writes.

Rather than use complicated B-trees or object databases, the closed-caption transcripts database is based on simple flat files-- it appends text and stores the length of each story. Conceptually, this is a simple design, but the tradeoff is it's more difficult to edit that data later when culling. Since the culling process has no real-time constraints this limitation is no significant problem. The obvious alternative, of using a myriad of tiny files to contain each story, would have degraded performance when doing keyword searches.

To go with the custom database we built a custom search engine to scan through the closed-captioned text looking for the word or phrase specified. Since the stories are arbitrarily numbered (1,2,3, and so on) as they are captured, the search function returns the number of the matching story. From that the story itself is easily retrieved. The search engine was designed using cgi-bin protocol, that is, a program that returns data through a Web server.

Creating a C++ cgi-bin

Although there is considerable mystique about cgi-bins and a common misconception that C++ is ill-suited for that as a language, it is straightforward to write a cgi-bin in C++.

For a cgi-bin, you simply output what you want the Web browser to see using standard out (that is, `cout`).

Of course, it's a little harder than it sounds because your output must be formatted as HTML code. You must also remember to output the suitable `Content-Type` at the beginning of the document, followed by two carriage returns. Failing to do so may cause unpredictable results with some Web servers.

It's slightly more difficult to interact with an HTML form to get input from the user. The Web server takes action by executing our cgi-bin program `form.cgi` in response to a user pressing Enter, and the output of that program is returned to user (via `cout`) to be displayed in his or her Web browser. You must, of course, have the `form.cgi` executable installed in the cgi-bin directory of the Web server.

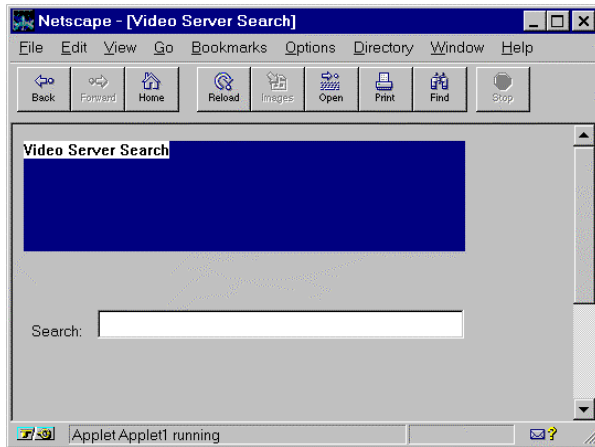
For a single field input you can use HTTP GET data. The field data is magically provided to you by the Web server in the `QUERY_STRING` environment variable, easily retrieved with a call to `getenv()`. Complex forms require HTTP POST data that is read from `cin`.

Porting cgi-bin to Netscape Plug-in

Setting up a Web server can be tricky, especially if you've never done it before. Each server has its own quirks and there are hundreds of different servers to choose from on the various platforms. And, don't forget that you must have TCP/IP networking set up properly first. That, of course, requires that your networking hardware be correctly installed.

A plug-in lets you extend Netscape, usually for the purpose of writing an inline viewer of a new file format. This can serve as an alternative to using a Web server. The Netscape plug-in API supports Windows, Macintosh, and UNIX platforms. In Windows, plug-ins are DLLs. The power of a plug-in can be awesome. They can be called by a Java method. Once inside the C++ code of your plug-in you are no longer under the normal restrictions of the Java applet security model. You are running in native code in Windows. Further, you can call back from C++ into Java, manipulating its GUI or data. You get the performance and power of C++ along with the portable GUI of Java. The Netscape plug-in developer's kit (at <http://www.netscape.com>) includes the `javah` compiler and example code for some simple plug-ins.

Netscape plug-ins are a bit strange. The code has a passing similarity to MFC, since plug-ins are a kind of framework. However, that's where the similarity to any normal C++ code ends. The `javah` compiler writes C/C++ wrappers for the methods in Java libraries. It isn't a compiler really, but a code generator. It works for both standard Java classes and for classes you create. The boilerplate code it writes is reminiscent of what a class wizard does for you with MFC. Although the implementation is completely different, using `javah` is conceptually similar to working with a COM. It gives you a way to get a pointer to a Java object and manipulate it as though it was a C++ object.



In this picture is a simple Web page with both a plug-in and Java applet. The plug-in is embedded in the blue area, where it says "Video Search." That string is actually being displayed by the plug-in DLL using a Windows call. You can display whatever you want in the plug-in's window using the normal Windows API calls.

The "Search:" string and its data entry field are a Java applet, not an HTML form. What the user enters as a search string is passed by the applet into the plug-in, that actually does the search and writes an HTML page (a file on the local file system). Finally, the applet then loads that page into the browser using the Java method *ShowDocument* ().

The HTML code in Listing Seven is virtually the same for any HTML code that uses a Java-aware plug-in that you might come across surfing the Web. There are two differences from simple HTML that allow Java to talk to the plug-in: One, the plug-in is actually embedded into the page by *EMBED*. You can't just *new* the Java class that has the native methods implemented in a plug-in. It has to be embedded in the page. The second significant point is Netscape's magic *MAYSCRIPT* tag.

Since you can't *new* a plug-in object, there must be some other way to call it out of the page. The method is to ask JavaScript for a handle to the plug-in. Even though there isn't any JavaScript code in the page, the applet needs the *MAYSCRIPT* tag to be allowed to call the JavaScript interpreter and use the Java *JSObject*.

Instead of *new*, you use code to fetch the JavaScript window object, then the document in the window, and finally the plug-in in the document. The name of the plug-in is the same as what it was called in the *EMBED* clause. Once the plug-in is retrieved it works like any other Java object. Calling its native *Search* method

hands off all the real work to the DLL. All of this code is just the normal plumbing necessary to use Java and C++ together in Netscape. Nothing special.


Conclusion

Our SVCR monitors broadcast television news identifying and recording stories of interest to the user. In this case it is a real-time system and uses a standard Web browser interface for maximum portability.

Anyone who wants to be able to produce a quick analysis of breaking news is a potential user. Early adopters include a military command center. Future enhancements will evolve the system beyond U.S. television news monitoring to indexing other types of video information, both with and without closed-captioning.

Epilog

Since publication of this article in 1997 this technology was further developed with funding support from DARPA (the research arm of the U.S. Department of Defense) under a contract that ended in 1999. As part of the DARPA effort it was successfully tested at sea by the author on the aircraft carrier USS Lincoln and on the Third Fleet flagship USS Coronado. The author won follow-on support from DISA to integrate it with the Global Broadcast System for deployment in the Pentagon. It is also a component of Project Genoa, a DARPA project to create an interactive daily briefing system for the President.

	<p>Robin Rowe is the CEO/COO of MovieEditor.com. During his tenure as chief technologist at SAIC, a Fortune 500 IT company, he designed BrowseCutter and VANA. His broadcast experience includes technical director of broadcast news at WICD-TV, a mid-market NBC station in central Illinois. He helped build the robotic studios at Chicago NBC station WMAQ-TV. He can be reached at Robin.Rowe@MovieEditor.com</p>
---	--