Chapter 11

Multimedia Component Frameworks

Simon Gibbs

Abstract This chapter looks at the use of object-oriented technology, in particular class frameworks, in the domain of multimedia programming. After introducing digital media and multimedia programming, the central notion of multimedia frameworks is examined; an example of a multimedia framework and an application that uses the framework are presented. The example application demonstrates how object-oriented multimedia programming helps to insulate application developers from "volatility" in multimedia processing capabilities — this volatility and related uncertainty is currently one of the key factors hindering multimedia application development.

11.1 Digital Media and Multimedia

In discussing object-oriented multimedia, a convenient starting point is the notion of *media artefacts*. Here the term "media" is used in the sense of materials and forms of expression. This includes both *natural media*, such as inks and paints, and *digital media* made possible by computer technology. The latter either mimic natural media, as is the case with drawing and paint programs, or have no natural counterparts. Those things produced by working in or with a particular medium are what we call media artefacts.

The distinction between natural and digital media also applies to artefacts. Natural artefacts are those produced using natural media. Among natural artefacts are paintings, prints, sculptures, photographs, musical recordings, and video and film clips. Digital artefacts include both the artefacts of digital media, such as an image produced by a paint program, and the digitized artefacts of natural media, for instance an image produced by scanning a photograph.

Until fairly recently, artists and designers primarily worked with natural media and so produced what we have just described as natural artefacts (it should be noted, though, that

Simon Gibbs, "Multimedia Component Frameworks," *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis (Eds.), pp. 305-319, Prentice Hall, 1995.

we are including such technologies as film and video as "natural" media). But the tools of the trade are changing, and now, as a result of the increasing capabilities of the computer, high-quality digital artefacts are becoming easier, and less expensive, to produce. There are many advantages to digital, as opposed to natural, artefacts — digital artefacts can be easily modified, copied, stored or retrieved. They can be sent over communications networks and can be made interactive. Equally intriguing is the ease with which digital artefacts are combined. Because, ultimately, digital artefacts simply reduce to bits and bytes, there are no physical restrictions on combining artefacts of different digital media. Digital video can be placed in text, or, vice versa, text can be placed in video; similarly audio and graphics can be combined, speech and text can be combined, and so on.

The notion of media artefacts leads to a natural definition for *multimedia*. We consider multimedia to be broadly concerned with the creation, composition, presentation, recording, editing and, in general, manipulation, of artefacts from diverse media. Since multimedia is so free in style, an immense variety of techniques, and combinations of techniques, are available to the artist. This is reflected in the wealth of media manipulation, composition and transformational capabilities packaged in multimedia authoring tools.

11.2 Multimedia Systems and Multimedia Programming

A complex multimedia production, whether a video game, a multimedia encyclopaedia or a "location-based entertainment environment," often requires the concerted effort of large teams of people. Like film and video production, multimedia production calls upon the talents of artists, actors, musicians, script writers, editors and directors. These people, responsible for "content design" to use current terminology, create raw material and prepare it for presentation and interaction. In doing so they rely on multimedia authoring environments to edit and compose digital media.

The authoring environments used for multimedia production are examples of *multimedia systems* [9]. Some other examples are:

- *multimedia database systems* — used to store and retrieve, or better, to "play" and "record" digital media;
- *hypermedia systems* — used to navigate through interconnected multimedia material;
- *video-on-demand systems* — used to deliver interactive video services over wide-area networks.

The design and implementation of the above systems, and other systems dealing with digital media, forms the domain of *multimedia programming*.

Multimedia programming is based on the manipulation of media artefacts through software. One of the most important consequences arising from the digitization of media is that artefacts are released from the confines of studios and museums and can be brought into the realm of software. For instance, the ordinary spreadsheet or wordprocessor no longer need content itself with simple text and graphics, but can embellish its appearance

with high-resolution colour images and video sequences. (Although the example is intended somewhat facetiously, we should keep in mind that digital media offer many opportunities for abuse. Just as the inclusion of multiple fonts in document processing systems led to many "formatting excesses," so the ready availability of digital media can lead to their gratuitous use.)

With the appearance of media artefacts in software applications, programmers are faced with new issues and new problems. Although recent work in data encoding standards, operating system design and network design has identified a number of possible services for supporting multimedia applications, the application programmer must still be aware of the capabilities and limitations of these services. Issues influencing application design include:

- *Media composition* — digital media can be easily combined and merged. Among the composition mechanisms found in practice are: *spatial composition* (the document metaphor) which deals with the spatial layout of media elements; *temporal composition* (the movie metaphor) considers the relative positioning of media elements along a temporal dimension; *procedural composition* (the script metaphor) describes actions to be performed on media elements and how media elements react to events; and *semantic composition* (the web metaphor) establishes links between related media elements.
- *Media synchronisation* — media processing and presentation activities often have synchronisation constraints [10][13]. A familiar example is the simultaneous playback of audio and video material where the audio must be "lip synched" with the video. In general, synchronisation cannot be solved solely by the network or operating system and, at the very least, application developers must be aware of the synchronisation requirements of their applications and be capable of specifying these requirements to the operating system and network.
- *User-interfaces* — multimedia enriches the user-interface but complicates implementation since a greater number of design choices are available. For example, questions of "look-and-feel" and interface aesthetics must now take into account audio, video and other digital media, instead of just text and graphics. Multimodal interaction [2], where several "channels" can be used for information presentation, is another challenge in the design of multimedia user-interfaces.
- *Compression schemes* — many techniques are currently used, some standard and some proprietary, for the compression of digital audio and video data streams. Application developers need to be aware of the various performance and quality trade-offs among the numerous compression schemes.
- *Database services* — application programming interfaces (APIs) for multimedia databases are likely to differ considerably from the APIs of both traditional databases and the more recent object-oriented databases. For example, it has been argued that multimedia databases require asynchronous, multithreaded APIs [6] as opposed to the more common synchronous and single-threaded APIs (where the application

sends the database a request and then waits for the reply). The introduction of concurrency and asynchrony has a major impact on application architecture.

- *Operating system and network services* — recent work on operating system support for multimedia — see Tokuda [14] for an overview — proposes a number of new services such as real-time scheduling and stream operations for time-based media. Similarly, research on "multimedia networks" (e.g. [4], [12]) introduces new services such as multicasting and "quality of service" (QoS) guarantees. Developers must consider these new services and their impact on application architecture.
- *Platform heterogeneity* — cross-platform development, and the ability to easily port an application from one platform to another, are important for the commercial success of multimedia applications. It is also desirable that multimedia applications adapt to performance differences on a given platform (such as different processor speeds, device access times and display capabilities).

In summary, a rich set of data representation, user interface, application architecture, performance and portability issues face the developers of multimedia systems. What we seek from environments for multimedia programming are high-level software abstractions that help developers explore this wide design space.

11.3 Multimedia Frameworks

In identifying abstractions for multimedia programming one should consider the prevailing programming paradigms such as functional programming, rule-based programming and object-oriented programming. While discussion of this topic is beyond the scope of this chapter, our position is that each of these paradigms has something to offer to multimedia, but that object-oriented programming, because of its support for encapsulation and software extension, is perhaps the most natural.

The apparent affinity between multimedia and object-oriented programming is clearly evident if one looks at the short history of programming environments for multimedia applications. From the earliest multimedia toolkits, such as Muse [8] and Andrew [3], to recent commercial multimedia development environments (e.g. Apple [1], Microsoft [11]) one can see the influence of the object-oriented paradigm. Often these environments and toolkits, in addition to structuring interfaces into classes and class hierarchies, have the more ambitious goal of building class frameworks for multimedia programming.

Perhaps the main benefits of object-oriented technology to multimedia programming are its mechanisms for extending software environments. Many of the issues listed in the previous section (media composition techniques, compression schemes, etc.) are, at their core, questions of how best to cope with the uncertainties of evolving environments. Frameworks, or hierarchies of extensible and interworking classes, offer developers a way of coping with evolution (see chapter 1). In the case of multimedia programming, several "evolutionary processes" are of concern, in particular:

- *Platform evolution* — the hardware platforms for multimedia applications are rapidly evolving. Capabilities that were once considered exotic, such as video compression and digital signal processing, are now found on the desktop (and soon the "set top").
- *Performance evolution* — many of the operations of interest to multimedia programming have real-time constraints, consider audio or video playback as examples. Such temporal dependencies make multimedia applications particularly sensitive to platform performance. It may be necessary, for instance, to adapt to less than optimal processing capacity by reducing presentation "quality" (e.g. lowering frame rates or sample sizes).
- *Format evolution* — new data representations for image, audio, video and other media types are likely to appear as a result of on-going standardization activities and research in data compression and media composition.

Developers want to create applications that can adapt to and take advantage of changes in platform functionality, increases in platform performance and new data representations. Of course it is impossible to write applications that can fully anticipate future developments in multimedia technology, but frameworks at least offer a mechanism for incorporating these changes into the programming environment.

11.4 A Multimedia Framework Example — Components

We now look at a particular multimedia framework — one that provides explicit support for component-oriented software development. This framework is described more fully elsewhere [5]. In essence it consists of four main class hierarchies: media classes, transform classes, format classes and component* classes (see figure 11.1):

- *Media classes* correspond to audio, video and the other media types. Instances of these classes are particular media values — what were called media artefacts earlier in the chapter.
- *Transform classes* represent media operations in a flexible and extensible manner. For example, many image editing programs provide a large number of filter operations with which to transform images. These operations could be represented by methods of an image class; however, this makes the image class overly complicated and adding new filter operations would require modifying this class. These problems are avoided by using separate transform classes to represent filter operations.
- *Format classes* encapsulate information about external representations of media values. Format classes can be defined for both file formats (such as GIF and TIFF, two

* The term "component" appears throughout this book, here the term is used in the specific sense of a software interface encapsulating software and/or hardware processes that produce, consume or transform media streams. Some examples are video codecs and audio players.

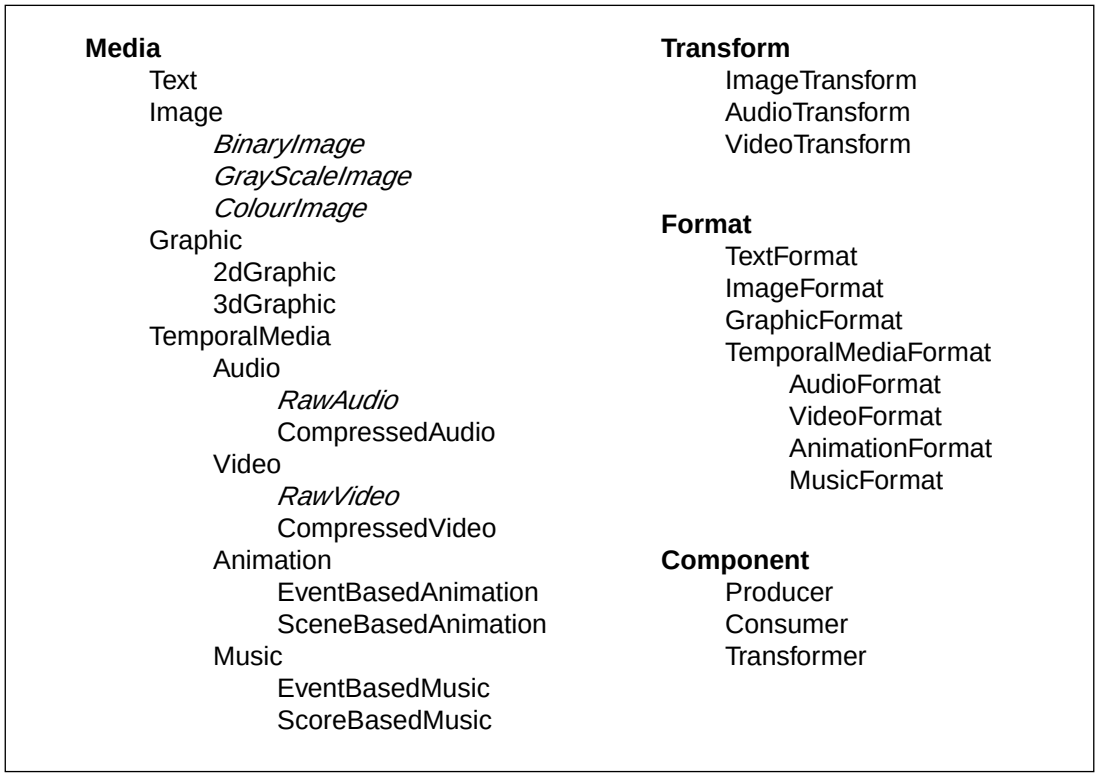

Figure 11.1 *Four class hierarchies of a multimedia framework: the Media, Format, Transform and Component classes and examples of their immediate subclasses. The classes shown are abstract (with the exception of those in italics) — concrete classes appear deeper in the hierarchies.*

image file formats) and for "stream" formats (for instance, CCIR 601 4:2:2, a stream format for uncompressed digital video).

- *Component classes* represent hardware and software resources that produce, consume and transform media streams. For instance, a CD-DA player is a component that produces a digital audio stream (specifically, stereo 16 bit PCM samples at 44.1 kHz).

Components are central to the framework for two reasons. First, the framework is adapted to a particular platform by implementing component classes that encapsulate the media processing services found on the platform. Second, applications are constructed by instantiating and connecting components. The remainder of this section looks at components in more detail.

11.4.1 Producers, Consumers and Transformers

The structure of a component is depicted graphically in figure 11.2. Of central importance are the *ports* through which media streams enter and leave. Components can be divided

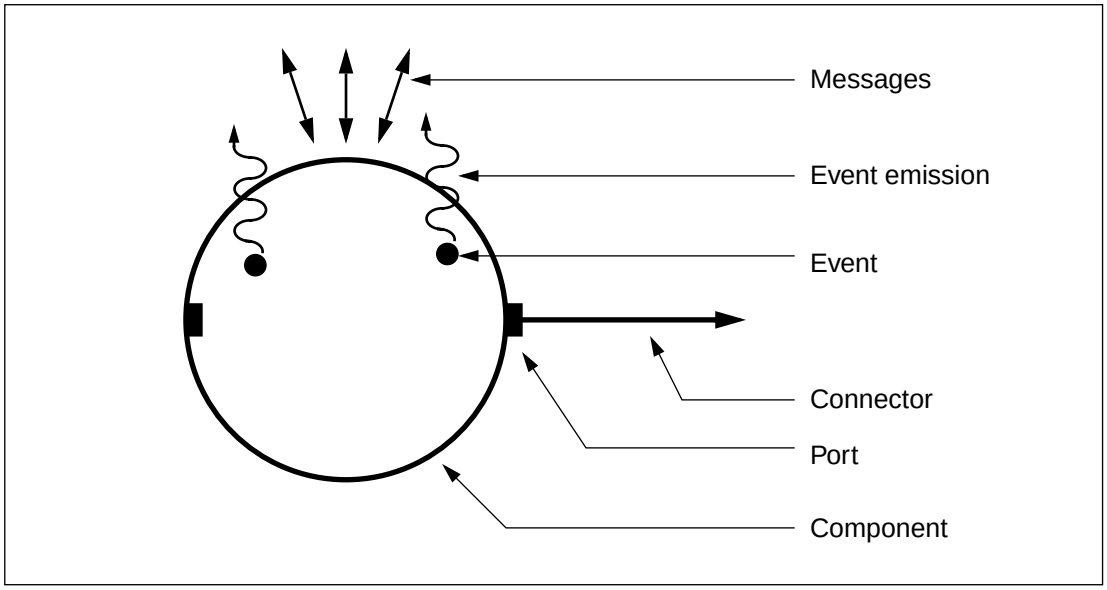

Figure 11.2 *Structure of a component. Three interfaces are available to the programmer: a synchronous interface based on message passing, an asynchronous interface based on events, and an isochronous interface based on streams. Streams enter and leave components through their ports and flow over the connectors joining components.*

into three broad categories based on the directionality of their ports: *producers* have only output ports, *consumers* have only input ports, and *transformers* have both input and output ports.

11.4.2 Component Interfaces

Components communicate with other components, and with other objects, via three interfaces:

- *Synchronous interface* — components, since they are objects, have a method interface describing messages that can be sent to the component and the associated replies. This interface is intended to allow external control over the component. For example, methods might include starting and stopping the component and querying or modifying operational parameters.
- *Asynchronous interface* — components emit events that can be caught by other objects (including other components, although building in dependencies between components is not recommended). As an example of event generation, a video player component might emit a "frame completed" event each time it produces a new frame on its output port. Generally the asynchronous interface is intended for monitoring and coordinating component behaviour.

- *Isochronous interface* — finally the input and output ports provide a third form of interface. Streams of media data (such as audio samples, video frames or animation events) enter and leave through ports. If congestion (or starvation) is to be avoided, connected components must operate at the same rate — in other words, connected components are *isochronous*.

11.4.3 Plug Compatibility

Several conditions must be satisfied before a pair of ports can be connected. In particular:

- One port must be an output port, the other an input port.
- The ports must be *plug compatible.*
- Creating the connection cannot exceed either port's *fan-limit* (the number of simultaneous incoming or outgoing connections a port may accept).
- The ports must accept the same form of connector. Generally connectors come in a variety of "forms" such as shared memory connectors, network connectors and connectors using a hardware bus.

Plug compatibility is related to type compatibility. Each port is associated with a set of stream format classes; these are the *supported types* of the port. When a port is to be connected, a specific member of this set is specified and is called the *activated type* of the port. An input and output port are then said to be plug compatible when the activated type of the output port is either identical to or a subtype of the activated type of the input port.

Plug compatibility rules out such errors as connecting a video output to an audio input. Of more interest though, are the situations involving subtyping. It is best to think of a port type as specifying the form of elements in the stream that flows through the port. Note that streams need not be homogeneous, one could have a stream containing both "circular" elements and "square" elements. Plug compatibility then says that an output port producing, for instance, only "circular" elements, can be connected to an input port that accepts streams containing both "circular" and "square" elements. In practice this means that we can connect a source to a sink provided the "vocabulary" of the source is included in that of the sink.

11.4.4 Component Networks

Groups of connected components are called *component networks*. A component network resembles a dataflow machine — streams of media data flow from producers, through transformers, and finally to consumer components. Applications are responsible for building component networks — in other words, applications build the virtual machine on which they run. This involves:

- *Instantiation* — the instantiation of a component results in resources being allocated for its operation. Resources include such things as memory, bus and network bandwidth, processor cycles, and hardware devices.
- *Initialization* — after creating a component object it must be initialized, i.e. operational parameters such as "speed" or "volume" must be set. The component's method interface is used for this purpose.
- *Connection* — after instantiating and initializing components, they can then be connected. Depending on the application, all connections may be made statically when the application begins (e.g. a two-party desktop conferencing application) or dynamically as the application runs (e.g. a multi-party desktop conferencing application where users have the ability to enter and leave conferences as they are running). An example of a tool that can be adapted to allow the visual configuration of media processing components is described in chapter 10.
- *Synchronisation* — components are subject to real-time constraints. In particular, media values enter and leave their ports at specific rates. If for some reason components are no longer able to process streams at the proper rates, then synchronisation errors start to appear (such as video lagging behind audio). When a component network falls "out-of-sync" it may be necessary for the application to specify corrective action (such as shutting down components, reducing quality, or acquiring more resources).
- *Event-handling* — during operation, components generate a variety of events. Applications can register interest in events and must then provide appropriate event handlers.

11.4.5 Media Processing Platforms and Component Kits

Finally, two important notions related to components are *media processing platforms* and *component kits*. A media processing platform is simply a set of hardware and software resources. Some examples would be a CD-i player, a MIDI network, a PC with a sound board, a video editing suite, a digital signal processor, and a network of "multimedia workstations" (workstations with audio and video capabilities).

Given a media processing platform, a component kit is the set of components offered by the platform. Clearly applications can only use available components. However, it should be possible for applications to adapt themselves, at least to some extent, to different platforms and different component kits. For instance, consider an application that plays multiple audio, video and MIDI tracks. If the application finds itself on a platform with no MIDI components, it might select simply to ignore any MIDI tracks during playback.

11.5 Video Widgets — A Programming Example

The preceding section contained a short overview of a proposal for an object-oriented framework for multimedia programming. To give a better idea of how such frameworks

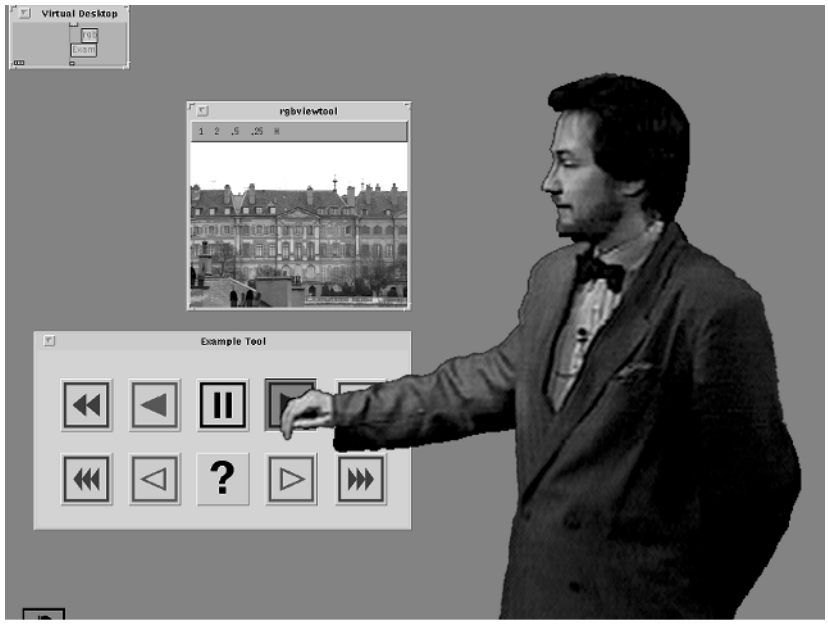

Figure 11.3 *A video widget and application windows.*

can be used, and how they can help shield applications from changes in platform architecture, we will look at a programming example based on an existing prototype.

The programming example we have chosen is the implementation of "video widgets" [7]. Video widgets, like graphics widgets (menus, buttons, icons and so on) are user-interface elements encapsulating both visual and behavioural information. Video widgets are rendered (i.e. displayed) by compositing video sequences (stored either in analog or digital form) over application graphics.

An example of a video widget is shown in figure 11.3. This widget is the basis of a simple "video assistant" for explaining and demonstrating the use of buttons belonging to some application. Such a video widget could be of use in multimedia kiosks or other situations where users may not be familiar with the operation of the application.

The implementation of video widgets involves components for playing, mixing and displaying video — these are producers, transformers and consumers respectively. The instantiation and connection of these components is performed by a class called VideoWidget, this class also provides application programmers methods for controlling widget behaviour. A partial class definition for VideoWidget is as follows:

```
class VideoWidget {
private:
    VideoPlayer*    player;     // a Component object (a Producer)
    VideoMixer*     mixer;      // a Component object (a Transformer)
    WindowServer*   wserver;    // a Component object (a Producer)
    Display*        display;    // a Component object (a Consumer)
    ActionTable*    atab;       // identifies widget actions
```

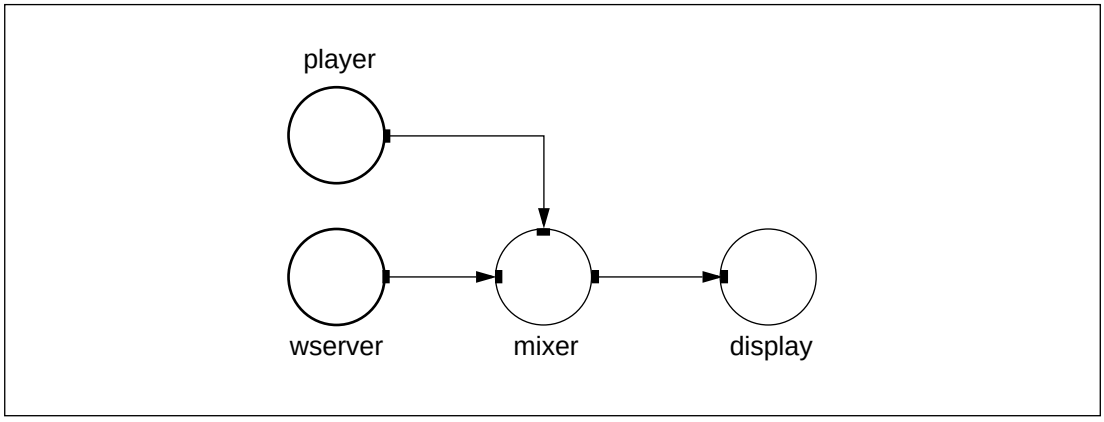

Figure 11.4 *A component network for a video widget.*

```
public:
                    // create a video widget
                    VideoWidget(WindowServer* w, Display* d,
                          Video* v, ActionTable* a, ChromaKey k);
                    // have widget perform some action
                    // this may generate events that can be
                    // caught by the application
        void        Perform(ActionId  aid, float  speed, bool  blockFlag);
};
```

The VideoWidget class includes instance variables that refer to the component objects used to build the "virtual machine" (i.e. component network) on which a video widget runs. The classes of these components are:

- VideoPlayer — an abstract class for components that playback video values (either analog or digital). Some specializations could include: VideoTapePlayer, VideoDiscPlayer, MpegPlayer and JpegPlayer. Methods declared by VideoPlayer (and implemented by the subclasses) include Load, Cue, Play and Pause.
- VideoMixer — a class for components that mix video using techniques such as chroma-keying. Methods include SetChromaKey, EnableKeying, BypassKeying.
- WindowServer — a class used to encapsulate window server functionality. A window server is represented by a producer component with a video-valued output port.
- Display — a class used for display devices. A particular display is represented by a consumer component with a video-valued input port.

Using the framework's notion of components and connections, a typical graphics application would consist of a WindowServer component connected to a Display component. Video widgets can then be implemented by "splicing" a video mixer and a video player into this connection. The resulting component network is shown in figure 11.4.

Configuration of the component network takes place in the constructor for VideoWidget. An outline of this method is:

```
VideoWidget::VideoWidget(WindowServer* w, Display* d,
                         Video* v, ActionTable* a, ChromaKey k)
{
    player = new VideoPlayer(v->Format( ));
    mixer = new VideoMixer;
    wserver = w;
    display = d;
    atab = a;

    // connect player and wserver outputs to mixer inputs
    // connect mixer output to display input

    // initialize components
    player->Load(v);
    mixer->SetChromaKey(k);
    mixer->EnableKeying( );
}
```

In addition to making component connections, the constructor loads a video value onto the video player and configures the mixer for chroma-keying. The constructor also takes an ActionTable argument; this is a data structure identifying offsets within the video value for particular "actions" that can be performed by the widget. A particular action is played back by using the Perform method:

```
VideoWidget::Perform(ActionId  aid, float  speed, bool  blockFlag)
{
    player->Cue(atab[aid]);                  // cue at start frame of action aid
    player->Play(speed, blockFlag);          // start playing, this method blocks
                                             // if blockFlag is TRUE
}
```

The VideoWidget class can be expanded in many ways to include such things as audio capabilities, multi-layer mixing and video effects (e.g. fading in or out a video widget). However, our purpose here is not really to discuss the use of video widgets or their design requirements, but rather to provide a non-trivial example of how component networks are mapped to media processing platforms.

Two possible, but radically different, platforms for video widgets are shown in figures 11.5 and 11.6. The first is based on analog video and external devices for mixing and switching. The second assumes a fast internal bus and hardware components for processing high data rate uncompressed digital video.

The important point of this example is that the differences between the platforms need not be visible to the user of video widgets. More specifically, it is possible to have a single implementation of the VideoWidget class for both platforms. The code for methods such as Perform remains the same; what changes between platforms are the implementations of the components used by VideoWidget. However, as long as implementations of VideoMixer, VideoPlayer, etc., provide the same interfaces, there is no reason to change the VideoWidget class.

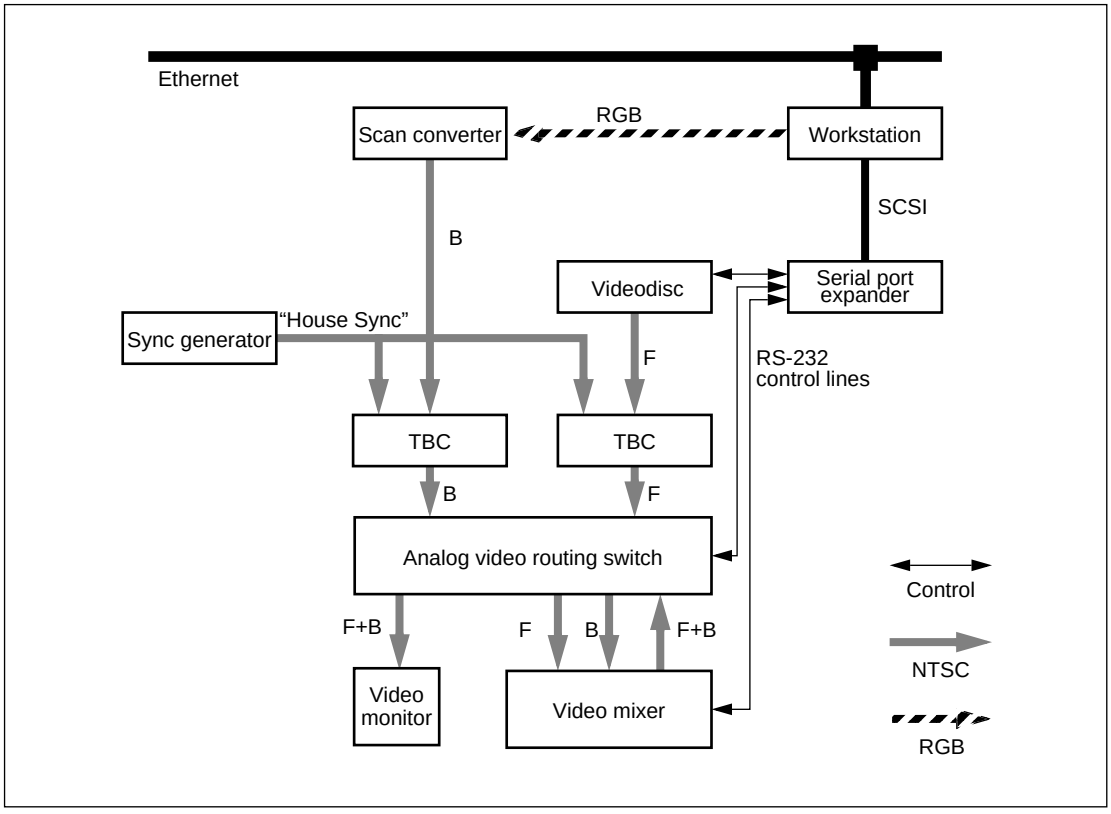

Figure 11.5 *An analog video platform for "video widgets". The two video signals F (front) and B (back) come from the video widget and the application respectively. The central part of this layout is an analog video routing switch allowing video equipment to be connected under computer control. The TBCs (time-base correctors) synchronize video signals against some reference signal (coming here from a sync generator) and are needed when video signals are mixed.*

11.6 Summary

Multimedia raises a host of new design issues for application developers. Questions of media composition, media synchronisation, data formats, user interfaces and database interfaces must be re-examined in the light of the capabilities of multimedia platforms. To take one example, advances in video compression techniques now make it possible to construct "video servers." These digital video storage and delivery systems are the basis of the new family of video-on-demand services and lead us to question the nature of the interface between applications and database systems.

One of the more severe practical difficulties facing developers of multimedia applications is the lack of stable target platforms. What can be called "platform volatility" results from the rapid pace of additions to the functionality of multimedia hardware, improve-

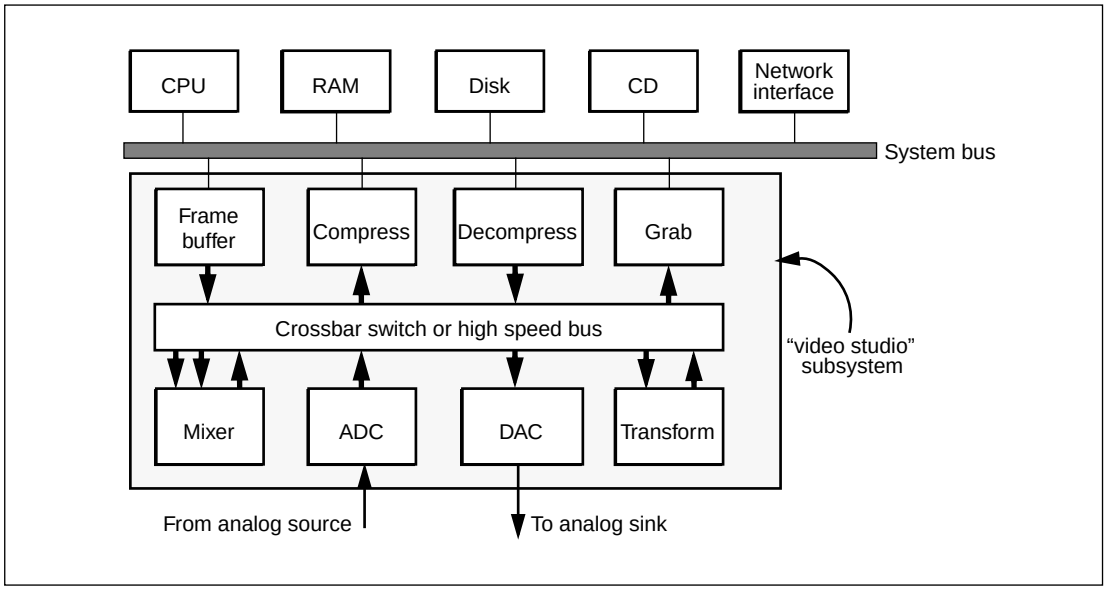

Figure 11.6 *A digital video platform for "video widgets". Heavy lines indicate high data rate streams. The analog-to-digital converter (ADC) and digital-to-analog converter (DAC) connect the "video studio subsystem" to external analog sources (e.g. video cameras) and sinks (e.g. monitors).*

ments in performance and quality characteristics, and the introduction of new media formats. In order to simplify cross-platform development, multimedia programming environments must address the issue of platform volatility. This chapter has argued, through a concrete example, that object-oriented programming, class frameworks and component-based software allow us to cope with platform evolution — that constructing applications from connectable and "swappable" components helps protect developers from even radical changes in target platforms.

References

[1] Apple Computer Inc., *QuickTime 1.5 Developer's Kit*, 1992.

[2] Meera Blattner and Roger Dannenberg (eds.), *Multimedia Interface Design*, ACM Press, Reading, Mass., 1992.

[3] Nathaniel Borenstein, *Multimedia Applications Development with the Andrew Toolkit*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[4] Dominico Ferrari, Anindo Banerjea and Hui Zhang, "Network Support for Multimedia: A Discussion of the Tenet Approach," Technical Report TR-92-072, International Computer Science Institute, University of California at Berkeley, 1992.

[5] Simon Gibbs and Dennis Tsichritzis, *Multimedia Programming: Objects, Environments and Frameworks*, Addison-Wesley / ACM Press, Wokingham, England, 1994.

[6] Simon Gibbs, Christian Breiteneder and Dennis Tsichritzis, "Audio/Video Databases: An Object-Oriented Approach," in *Proceedings IEEE Data Engineering Conference*, Vienna, 1993, pp. 381–390.

[7] Simon Gibbs, Christian Breiteneder, Vicki de Mey and Michael Papathomas, "Video Widgets and Video Actors," in *Symposium on User Interface Software and Technology* (UIST '93), 1993, pp. 179–185.

[8] Matthew Hodges, Russel Sasnett and Mark Ackerman, "A Construction Set for Multimedia Applications," *IEEE Software*, vol. 6, no. 1, 1989, pp. 37–43.

[9] John Koegel Buford (ed.), *Multimedia Systems*, Addison-Wesley, Reading, Mass., 1994.

[10] Thomas D.C. Little *et al.*, "Multimedia Synchronization, *IEEE Data Engineering Bulletin*, vol. 14, no. 3, 1991, pp. 26–35.

[11] Microsoft Corporation, *Microsoft Windows Multimedia Programmer's Reference*, Microsoft Press, 1991.

[12] Doug Shepherd and Michale Salmony, "Extending OSI to Support Synchronization Required by Multimedia Applications," *Computer Communications*, vol. 13, no. 7, 1990, pp. 399–406.

[13] Ralf Steinmetz, "Synchronization Properties in Multimedia Systems," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 3, 1990, pp. 401–412.

[14] Hideyuki Tokuda, "Operating System Support for Continuous Media Applications," in *Multimedia Systems*, ed. John Koegel Buford, Addison-Wesley, Reading, Mass., 1994, pp. 201–220.