An Adaptive Framework for Developing Multimedia Software Components *

Edward J. Posnak, **R. Greg Lavender**, and **Harrick M. Vin**

Distributed Multimedia Computing Laboratory
Department of Computer Sciences, University of Texas at Austin
Taylor Hall 2.124, Austin, Texas 78712-1188
E-mail: {ejp,lavender,vin@cs.utexas.edu}, Telephone: (512) 471-9732, Fax:(512) 471-8885

1 Introduction

Recent improvements in microprocessor performance have made possible the migration of continuous media processing from specialized hardware, such as decompression and digital signal processing boards, to software. The extensibility and configurability of software libraries allows multimedia applications to access a wider range of multimedia objects, stored in a variety of compressed formats, and to employ an extensible set of tools for processing these objects. Furthermore, configurable software libraries enable applications to take advantage of new audio and video compression standards as they emerge, rather than becoming obsolete. Despite these advantages, there are two fundamental problems that have limited the success of software libraries for processing digital audio and video:

- *Difficulty of developing software components.* The task of developing software components (e.g., media players, Netscape plug-ins, ActiveX controls) that decode and process digital audio and video remains time-consuming and costly, due to the inability to reuse code, design patterns, or domain expertise. Media processing component implementations achieve high performance by tightly coupling the media processing code with the environment-specific code. However, this lack of modularity makes it difficult to factor out reusable code and often results in a "cut and paste" form of code reuse. Insufficient modularity and information hiding requires each new component to be designed from scratch, and forces the component developer to understand, in detail, the implementation of the reusable code as well as the underlying digital representation of the audio and video objects.
- *Variation of resources during presentation.* Whereas hardware decompression boards provide a fixed and dedicated set of processing resources, a software decoder runs on a variety of different processors, with speeds differing by an order of magnitude. Moreover, since processing resources are normally shared by a number of applications, the availability of these resources to the software decoder will vary as the system load changes. Such variations in the available processing resources can result in undesirable skips and/or distortion of the output signal, significantly degrading the quality of the presentation.

To address these problems, we have developed the Presentation Processing Engine (PPE) framework that (1) simplifies the development of multimedia software components by promoting the reuse of code, design patterns, and domain expertise, and (2) enables the components to dynamically adapt the quality of the presentation to the available resources in heterogeneous

*This research was supported in part by IBM, Intel IAL, the Intel Graduate Fellowship, the National Science Foundation (Research Initiation Award CCR-9409666), the National Science Foundation (Career Award CCR-9624757), NASA, Mitsubishi Electric Research Laboratories (MERL), Sun Microsystems Inc., and the University of Texas at Austin.

environments. The following sections describe the underlying design and provide examples of using the PPE framework to develop media processing components.

2 Framework Design

Presentation processing involves applying a sequence of transformations to a data stream between its source (e.g., microphone, camera, file) and sink (e.g., speaker, display device, storage server). The design of the PPE framework is inspired by the observation that many presentation processing components employ the same primitive compression and signal processing transformations (e.g., Huffman coding, dithering, etc.), operate on a common set of data types (e.g., frames and blocks), and share a reusable design architecture. To facilitate the development of such components, the PPE framework provides a library of reusable modules, which implement the primitive transformations, as well as the mechanisms that allow these modules to be composed into processing pipelines [4].

The PPE module library uses fine-grained modular decomposition to effectively decouple the environment-specific elements of the code from the reusable transformation implementations. This allows for significant code reuse when developing new components, and also makes it possible to incorporate reusable signal processing modules at intermediate stages of compression, where factors of ten speedup can be achieved [5]. Whereas such efficiencies are normally obtained by tight coupling of signal processing and compression operations, in the PPE framework these transformations are implemented as separate modules, and hence, can be reused to compose other software components.

The PPE framework facilitates the reuse of domain expertise by encapsulating primitive transformation implementations inside composable modules. The component developer need not be concerned with the details of transformation algorithms or the underlying digital representation of continuous media objects, but only needs to specify which transformations are to be applied, and their relative order. The framework will perform the task of constructing an efficient processing pipeline that performs these operations.

The PPE framework addresses the problem of resource variation during presentation by providing mechanisms that allow a component to dynamically reconfigure its implementation pipeline with different modules. Throughout playback, a component can switch between module implementations that are functionally equivalent but have different processing cost and quality characteristics. For example, different dithering implementations, ranging from low to high complexity, can be dynamically configured to realize different cost vs. quality tradeoff points. This allows the component to dynamically adapt its implementation to different end-station capabilities and changing resource loads, and can provide the user with flexible control over how the quality degrades when insufficient processing resources are available.

Whereas dynamic reconfigurability enables adaptation, it also carries an associated efficiency cost due to the procedure call overhead of dynamically dispatched method invocations. The more frequently a module interface is called, the more costly the performance penalty. Compile time (static) binding of modules can significantly reduce the overhead of module boundary crossing by allowing efficient choices to be made during code generation. The statically bound modules are implemented using parameterized types and inline methods to minimize the boundary crossing overhead, effectively collapsing a sequence of modules into highly efficient code. The drawback of static composition is that, once modules are bound in this fashion, they cannot be dynamically reconfigured. Hence, the development of modular, configurable, and efficient presentation processing components requires a carefully engineered balance between the use of static and dynamic composition of modules. To facilitate reuse of this design pattern each time a new component is developed, the PPE framework provides mechanisms that allow modules to be composed both statically and dynamically [3].

3 PPE Composition: Examples

The PPE framework has been used to develop a variety of decompression engines (e.g., JPEG and MPEG) and media player components (e.g., Tcl/Tk, Netscape Navigator Plug-in). This section illustrates how the framework is used to develop such components.

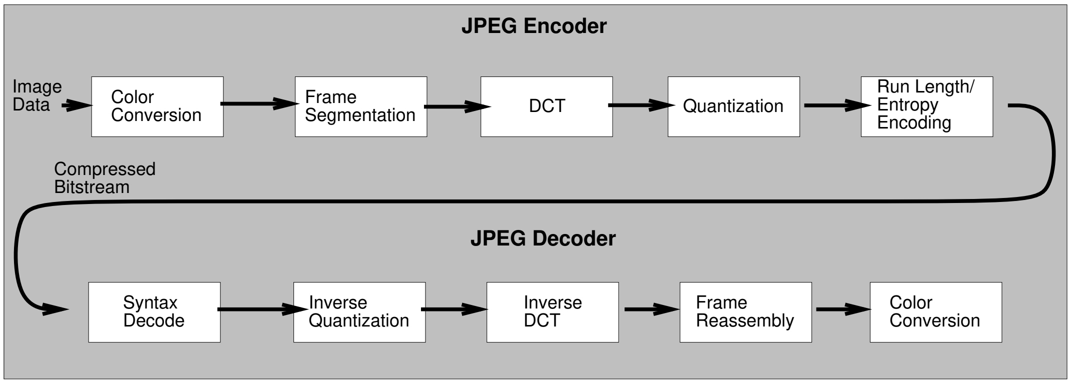

Figure 1: Sequence of operations in a JPEG Encoder and Decoder.

3.1 JPEG Decoder

JPEG is a widely used standard for image compression and video conferencing [6]. A JPEG encoder performs a series of transformations on raw image data to produce a highly compressed output stream, which can then be transmitted to a JPEG decoder that applies the inverse of these transformations to decompress the images. Figure 1 shows the series of transformations for JPEG encoding and decoding.

To develop a JPEG decoder, a set of reusable PPE library modules that perform these transformations are composed as illustrated in Figure 2. In this decoder, input from the media object is parsed by the syntax decoder and separated into control data and coded image data (JPEGSyntaxDec). Whereas control data is used to set decoder state variables and to determine module configurations, the coded image data is passed down the pipeline of modules that reconstruct the image for display. The block decoders (BlockDecs) read compressed data from the input bitstream and perform the Huffman and run-length decoding operations necessary to reconstruct small blocks of the image in the frequency domain. These blocks are then inverse quantized, transformed back to the spatial domain (IDCTDec), and then used to reconstruct luminance and chrominance frames (FrameDec). [1] Finally the frames are transformed back into a displayable color space and dithered, if necessary, to remove the banding effects of color quantization on color-mapped displays (ColorConversion and Dither).

The modules in this pipeline are shown having different shapes that fit together to illustrate the fact that module interfaces are strongly typed to ensure that only feasible configurations are permitted. Modules shown with solid borders have dynamic linkage, and may be reconfigured throughout playback. Modules shown with a dashed border are statically bound, using type parameterization, for the sake of efficient implementation.

The block decoder, shown in Figure 2(b), exemplifies how static module parameterization is used to achieve high performance while maintaining modularity. BlockDec is a module that must employ functionally different Huffman decoders, depending on the control data associated with a block. While it is possible to dynamically reconfigure a block decoder's internal modules to handle these cases, this approach is inefficient because the overhead of an abstraction boundary crossing, i.e. a dynamically dispatched method call, will be incurred thousands of times per image. To avoid sacrificing modularity, inlined interface methods and module parameterization are employed to compose these components into a high performance, statically bound implementation. A number of block decoder types are created by parameterizing the block decoder with a pair of Huffman decoders that provide inline methods for their decode functions. When combined with the selective inlining of performance critical functions, this parameterized approach maintains a highly modular architecture whose abstraction boundaries are enforced at compile time, but are compiled away to produce efficient run-time code.

[1] Many lossy compression algorithms convert images into a luminance/chrominance color space (e.g., Y,Cb,Cr) to achieve higher compression gain with less perceptible loss in the image.

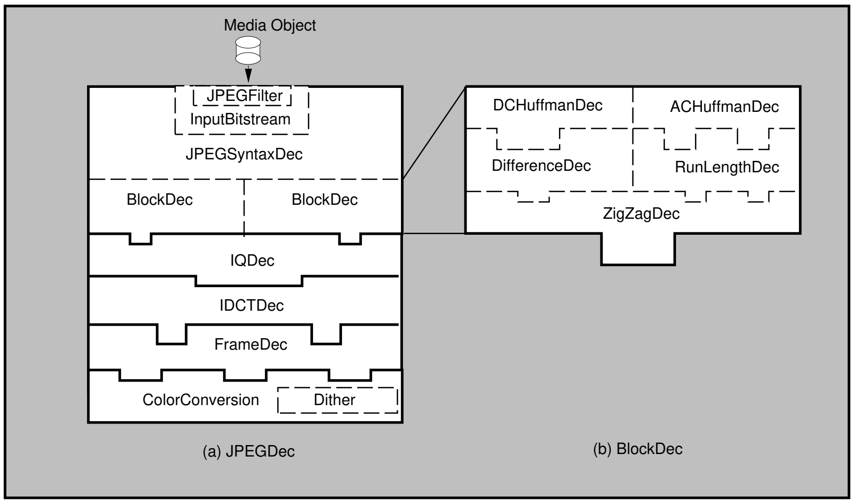

Figure 2: Module Composition for a JPEG Decoder. (a) JPEG Decoder (b) block decoder module. Different shapes are used to indicate different module types. Dynamically bound modules are shown with a bold outline; statically bound modules are shown by a dashed outline.

3.2 Tcl/Tk Media Player

Traditionally, domain expertise in both compression and window system programming has been required to develop just a single-format media player, targeted at a specific window system environment. However, by leveraging the power of the PPE and Tcl/Tk[2] toolkits, an adaptive, multi-format media player can be rapidly developed without domain expertise. Developing such a media player involves making selected PPE objects controllable by Tcl commands, and then writing a Tcl script that implements the media player.

Three objects will be needed to implement the media player: (1) A *source*, which gets the compressed data and passes it to the PPE, (2) a *PPE*, which is composed of modules that perform the necessary transformations to decode a particular stream, and (3) a *sink*, which presents the uncompressed stream on some output device, such as a display window. To make each of these objects controllable by Tcl commands, we define an interface class, called TclObject, that has one method: DoTclCommand. For each of the three objects, we create a class that inherits the TclObject interface, and contains an instance of the corresponding PPE object. Each of these classes will implement DoTclCommand by calling the appropriate method(s) of its internal PPE object. Figure 3 shows these class relationships.

Creating a Tcl media player can now be done with just a few lines of code that create source, PPE, and sink instances, initialize the source with the media object to be played, and configure the PPE. More Tcl code can be written to implement policies for adapting to the runtime environment by reconfiguring the PPE. Since implementing these features requires only knowledge of the possible module configurations, not their implementations, a powerful adaptive media player can be developed by a programmer with very little domain expertise. [3]

[2] Tcl [2] is an interpreted scripting language that can be used to control and configure software components. When a Tcl interpreter is embedded inside a component, users and applications can then configure and control the component dynamically using the Tcl scripting language. Tk is an extension to Tcl that allows user interface objects (e.g. windows) to be quickly and easily developed.

[3] This was, in fact, the case when novice multimedia programmers used our Tcl/Tk wrappers to create a Netscape Navigator plug-in with less than 50 lines of original code [1].

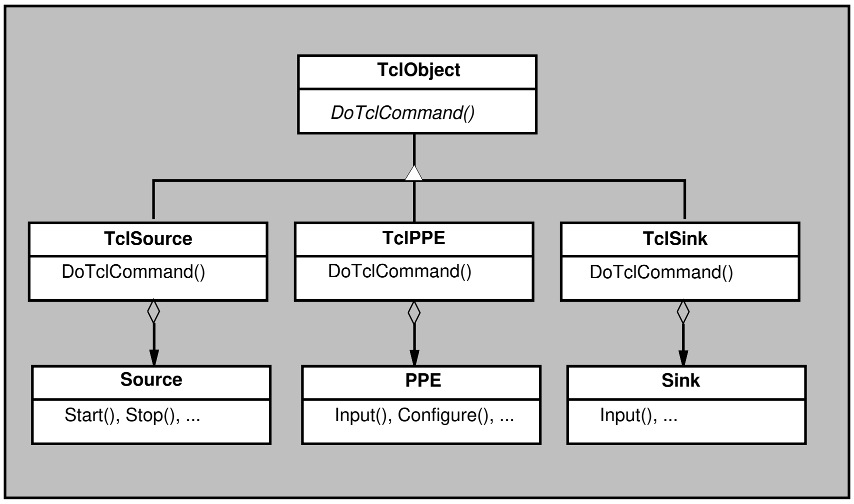

Figure 3: Class diagram for PPE objects that are controllable by Tcl commands

4 Conclusion

We have described the high-level design of a framework for developing software components that decode and process digital audio and video data. [4] This framework provides mechanisms that enable components to be developed using a modular compositional approach that has advantages in both software development and runtime performance. The framework supports the development of new media processing components by facilitating the reuse of code, design patterns, and domain expertise. The fine-grained modular decomposition and mechanisms for dynamic configurability enable components to adapt their implementations to heterogeneous environments, fluctuations in resource availability, and changing client preferences. Our implementation has shown how the well engineered balance between static and dynamic composition can be used to maintain a highly modular, but efficient object-oriented implementation. Moreover, performance gains are achieved by enabling signal processing modules to be inserted at intermediate stages of compression. Given that compression technology is still evolving and presentation processing is a common bottleneck in communications performance, improvements in this area will have a positive impact on the performance and structure of future distributed multimedia applications.

References

[1] B.S. Cherukupally and P. Uppuluri. Netscape Plug-In. Multimedia Systems class project, December 1996.

[2] J.K. Ousterhout. *Tcl and the Tk Toolkit*. Professional Computing Series. Addison-Wesley, 1994.

[3] E. J. Posnak, R. G. Lavender, and H. M. Vin. Adaptive Pipeline: an Object Structural Pattern for Adaptive Applications. In *The Third Pattern Languages of Programming conference*, Monticello, Illinois, September 1996.

[4] E. J. Posnak, H. M. Vin, and R. G. Lavender. Presentation Processing Mechanisms for Adaptive Applications. In *Proceedings of Multimedia Computing and Networking*, San Jose, California, February 1996.

[5] B. Smith. Fast Software Processing of Motion JPEG Video. In *Proceedings of the ACM Multimedia'94*, pages 77–88, October 1994.

[6] G. K. Wallace. The JPEG Still Picture Compression Standard. *Communications of the ACM*, 34(4):31–44, April 1991.

[4] A more detailed description of the design and implementation of this framework may be found at http://www.cs.utexas.edu/users/ejp/CACM.ps.