

A Real-Time Multimedia Composition Layer

Stefan Müller Arisona

Computer Systems Institute
ETH Zurich
CH-8092 Zurich

s.mueller@computer.org

Simon Schubiger-Banz

Swisscom Innovations
CH-3006 Bern

simon.schubiger@swisscom.com

Matthias Specht

Anthropological Institute
University of Zurich
CH-8050 Zurich

specht@ifi.unizh.ch

ABSTRACT

We introduce a light-weight multimedia composition layer which is situated on top of existing music, sound, and graphics frameworks and libraries. The layer provides applications with a programming interface for multimedia composition, automating real-time media processing and synchronisation. Our work facilitates the mapping of media frameworks to a unified processing graph. We present a graph segmentation algorithm that solves the problem of communication between threads of multiple frameworks while providing global consistency and without affecting processing performance. In addition, support for logical and physical time is included and enables framework-independent realisation of complex multimedia designs. The paper discusses the layer's architecture in detail and shows how multimedia applications can efficiently exploit its capabilities.

Categories and Subject Descriptors: H.5.1 [Information Interfaces and Presentation]: Multimedia Information Systems, H.5.5 [Information Interfaces and Presentation]: Sound and Music Computing, I.3.2 [Computer Graphics]: Graphics Systems.

General Terms: Algorithms, Design.

Keywords: Multimedia Composition, Multimedia Frameworks and Libraries, Cross-media Processing, Real-time Multimedia.

1. INTRODUCTION

Today, there exists a vast number of multimedia processing frameworks and libraries, some of them integrated into operating systems, others as add-ons or application-specific libraries. Their scope ranges from basic I/O (e.g., for digital audio) and state machines (e.g., for graphics handling) to higher level signal flow processing graphs for audio and music processing or hierarchical structures, as they are found in scene graphs for computer graphics. As each of these components successfully operates in its target domain, there is no or only rudimentary support for building, i.e., *composing* a high-level structure for multimedia processing out of basic elements. The situation is particularly dramatic when it comes to the question of how different frameworks are interconnected – a mandatory task when an application needs to simultane-

ously process multiple media types. Consequently, complex composition mechanisms are often hidden in the application logic and have to be re-implemented from scratch for every new application.

This paper addresses the question of how multimedia composition, particularly in real-time and across media types, can be extracted from application logic and generalised in a flexible and efficient manner. Therefore, we introduce a light-weight multimedia composition layer called DECKLIGHT, which is situated on top of media processing frameworks, and which provides application designers with an unified interface for multimedia composition. All composition issues are consistently de-coupled from application logic, as well as from the underlying layers.

At its core, DECKLIGHT employs a global *processing graph*, which consists of interconnected *processing nodes*. The nodes may be seen as the “glue” between the application layer and an underlying media processing layer. Each processing node acts as a building block for multimedia processing that can be configured through a number of *input ports* and represents its state (or a part thereof) by a number of *output ports*. While this scheme *per se* is not new, here, it is generalised in terms of allowing multiple frameworks to be represented within a single graph.

Each of these frameworks is allowed (but not required) to use its own thread(s) of execution and may impose an individual model of computation (e.g. flow-based, as commonly applied for audio processing, or hierarchical, as known from graphics scene graphs). Thus, multiple subgraphs inside the graph may run asynchronously against each other. Several mechanisms in DECKLIGHT deal with hiding this asynchrony from the application, as well as transparently controlling connections between asynchronous subgraphs. In addition, all modifications to the graph, such as node creation, connection establishment, or port I/O, are collected in *transactions*, and executed atomically at well-defined synchronisation points for every subgraph.

DECKLIGHT's multi-threaded operation efficiently leverages recent developments in mainstream CPU architectures (multi-core, multi-processor) while at the same time hiding synchronisation and data integrity issues from application programmers, which drastically reduces development time. In addition, DECKLIGHT can be employed throughout the complete development cycle for multimedia applications, from initial experiments and early prototypes to production-grade final results.

Regarding terminology, we will use the term *framework* for underlying processing entities and thereby also include class libraries (and the like), which are not strictly frameworks in the common sense (e.g., as defined by [10]). We shall however note that the composition layer reveals itself as a framework to integrators of underlying entities and as a library to application developers.

The paper starts by giving an overview of existing multimedia

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AMCMM'06, October 27, 2006, Santa Barbara, California, USA.
Copyright 2006 ACM 1-59593-501-0/06/0010 ...\$5.00.

processing frameworks and applications. We will also show how DECKLIGHT is related to a general multimedia composition and performance platform. Section 3 will discuss the basic requirements that preceded the realisation of DECKLIGHT and how they defined the fundamental design factors. In Section 4, the principal architectural components of our approach will be discussed in detail, and we will outline how they were realised in Section 5. Realised multimedia applications and the lessons learned will be discussed in Section 6. The paper concludes with final remarks and an outlook in Section 7.

2. RELATED WORK

Graph-based processing has a long tradition in many domains of computing. From our viewpoint, two of the main advantages of graphs are a) that graphs are formally established in terms of various computational models, verification, and transformation. And b), graphs can intuitively be represented and manipulated in graphical user interfaces (GUIs).

One important class of graph-based computing models are process networks [11] and the closely related dataflow graphs [13]. These models can be applied to represent data flow in digital signal processing networks: Typically, graph nodes act as sources, sinks, or arbitrary combiners of input and output of media streams (e.g. audio or video), together with a number of control-rate parameters, and process scheduling is determined by the availability of data. This class of graphs appears (with variations) in virtually any framework or application that employs graphs for digital signal audio or video processing. A throughout review of existing dataflow environments is given in [13], and we shall list relevant examples in comparison to our work below. Notable is the current trend of integrating flow-based signal processing networks into operating system services, as it is done for example by Apple's Core Audio [4] and Core Image [5], where *Audio Units* and *Image Units* are interconnected to build audio and image processing flow graphs.

A graph class apart from dataflow graphs appears in computer graphics: Typically *scene graphs* [19, 21, 25] are applied to represent three-dimensional object-hierarchies. The scene graph can recursively be processed from a root, e.g., a viewport, which is connected to child nodes that build a directed acyclic graph. That way, transformations and other object properties can easily be represented by specific nodes that ultimately characterise the graphs sources (or sinks, depending on how the the graph is directed), i.e., geometric objects. Since most scene graph implementations are realised on top of graphics state machines such as OpenGL, they internally reorganise the graph's structure for performance improvements (such as sorting nodes by state).

What distinguishes our work from preceding work is that DECKLIGHT does not enforce a particular computational model, but rather attempts to provide the conceptual framework that allows for seamless adaptation of underlying models. For example, this characteristic leverages the integration of scene graphs and dataflow graphs into a single global processing graph. Nevertheless, there are a few notable examples of existing work relevant to our work.

HP's NIZZA framework [22] is a real-time framework for general media stream processing (i.e., not restricted to audio). As in our work, NIZZA emphasises on parallel execution of the underlying computational model (where possible). Unfortunately relatively little is published on the actual realisation. In addition, the creation and modification of the flow graph seems rather static, i.e., fixed during operation, whereas in DECKLIGHT the processing graph can be modified at any time.

AURA [6] is a framework for audio and music processing built on top of the W system [7]. W is a framework with the primary

function of interconnecting building blocks within programs – very much like DECKLIGHT's principal goal. For our work, the most important point of W is the way it structures real-time processing: Different processing tasks can be assigned to designated *zones*. Inside a zone processing occurs synchronously, where as connections between zones are asynchronous, i.e., each zone has its own thread of execution. Similar concepts exist in hardware design and are known as *globally-asynchronous, locally-synchronous (GALS)* systems [17]. In DECKLIGHT, zones are equal to subgraphs of the global processing graph. The subgraphs are controlled by *schedulers*, and our work describes in detail how the subgraphs can be automatically constructed within given constraints and without having to interrupt processing.

CLAM [3] is a C++-based software framework for research and application development in the audio and music domain, and in particular offers tools for the analysis, synthesis and transformation of audio signals. What distinguishes CLAM from other frameworks is its rigorous application of the Object Oriented paradigm and it proposes to model a system in terms of objects and relations between objects: The *Digital Signal Processing Object Oriented Metamodel (DSPOOM)* classifies signal processing objects and the OO metamodel is closely related to dataflow process networks. The DSPOOM defines *Processing Objects* that contain Ports for data flow interconnections (synchronous) and Controls for control flow interconnections (asynchronous). Compared to CLAM, DECKLIGHT shares many concepts such as strict object-orientation, the repository, dynamic composition, dynamic types, and introspection. In contrast, DECKLIGHT also provides input and output ports, but does not distinguish between synchronous data and asynchronous control, as asynchronous connections are realised through asynchronously running subgraphs. In addition, from an implementation view, DECKLIGHT's processing nodes are more light-weight as compared to CLAM's Processing Objects, since the processing nodes' main purpose is to act mainly as a glue between the application and the underlying processing model.

MET++ [1] is an object-oriented multimedia framework for arbitrary media types and is built on top of ET++ [23]. Thus, and compared to other frameworks (including DECKLIGHT) it not only provides mechanisms for multimedia processing, but contains a fully-fledged application construction framework with GUI element and document classes. MET++ includes a notable speciality that is lacking or only rudimentary supported in many existing music or multimedia processing environments – the way it deals with time: Temporal relationships are expressed by a composition hierarchy, which allows for supporting symbolic and physical time bases [14] and for automatic time calculation. In addition, time-dynamic media objects provide time-dependent information in terms of sequences or time lines. In DECKLIGHT, the concept of a time hierarchy is generalised and embedded as a time subgraph as part of the global processing graph and we shall see that time-generating or -modifying objects are just another class of processing nodes.

Besides of the mentioned frameworks, a notable example from the application domain is the Max/MSP [18] (and its variations). Max is a visual programming environment specialised for media processing and is very popular among media artists and computer musicians. In Max, so called *patches* define visual programs, where the program logic is represented by a graph. For example, there are means for control structures and calculation, such as loops, conditionals, or arithmetic. In contrast to Max, DECKLIGHT does not attempt to provide graph composition at such a microscopic level, but rather emphasises macro-composition of components that reflect a potentially complex operation on the underlying framework.

Another difference is DECKLIGHT’s capability to be integrated into arbitrary applications, providing the developer with numerous degrees of freedom to adapt its behaviour to the application’s needs.

At a larger scope, DECKLIGHT has emerged as a host-based multimedia composition layer as part of the networked multimedia authoring and performance platform *Soundium* [20, 16]. The platform implements a thin network layer exporting DECKLIGHT’s API for inter-host communication as well as a GUI for interactive multimedia design and management. A screenshot of Soundium’s GUI containing views of the *design tree*, an interactive design editing component, of the processing graph, and of the processing node inspector is shown in Figure 1.

DECKLIGHT itself does not provide any means for processing, but rather leverages the integration of multiple underlying processing frameworks, and provides a unified and consistent view to applications. The principal questions that arise at this point are whether virtually every underlying framework can be represented by a graph structure (which is a requirement in order to fit into DECKLIGHT’s global processing graph), and how a multitude of frameworks can be interconnected in a non-intrusive manner. We shall see in the upcoming sections how these issues are accomplished.

3. REQUIREMENTS

DECKLIGHT arose from our needs to tightly interconnect real-time processing of multiple media – particularly music, sound, and graphics – and also integrate I/O devices and network capabilities in a uniform manner. Based on these needs, we shall first identify necessary design requirements. The first set of requirements are those that arise from the interface between the application layer and DECKLIGHT, i.e., the high-level interface that is applied when developing applications using DECKLIGHT:

R.1 – Uniform Handling of Different Media Types. The composition layer shall employ a data structure which uniformly represents processing of different media frameworks and hence multiple media. Relationships across media types should be dealt with equally as intra-media relationships.

R.2 – Consistent State Modification. Modifications to the processing state should be allowed at any time without interrupting real-time operation. These changes shall be applied consistently in respect to underlying processing mechanisms. For instance, changes to a graphics scene graph shall occur at frame boundaries. Multiple change operations shall be collectable in transactions and executed atomically (with respect to well-defined synchronisation points).

R.3 – Automatic Timing Control. Timing and time control are crucial issues in real-time multimedia systems. The multimedia layer shall employ mechanisms for automatic synchronisation across different subsystems. In addition, there shall be support for high-level sequencing of processing tasks.

R.4 – Hiding of Low-Level Processing Issues. Low-level processing issues such as parallel execution (e.g., through multi-threading) or resource management shall not be visible to the application layer.

R.5 – Error Tolerance. The system shall handle errors gracefully. In particular, errors that occur due to state modification during real-time operation shall not prevent unaffected parts of the system from ongoing operation.

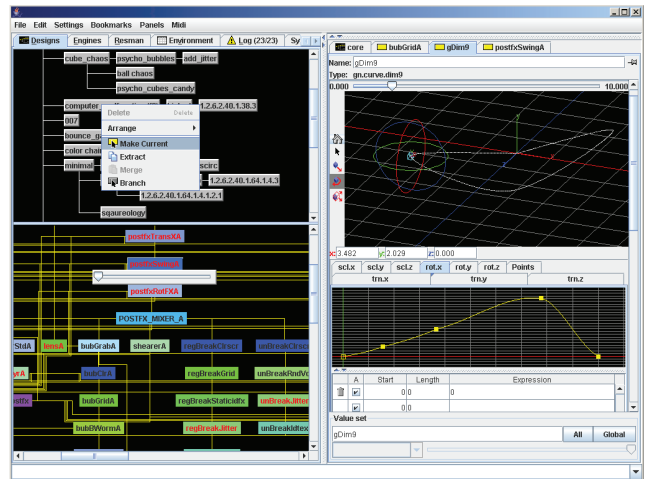


Figure 1: Screenshot of Soundium’s GUI. Top left: The design tree, an interactive design editing component. Bottom left: The global processing graph representing the current system state. Right: A node inspector for modification of a selected processing node.

The second group of requirements emerges from the task of integrating underlying frameworks:

R.6 – Non-Intrusive Integration. It should be possible to integrate underlying frameworks without modifying them. In many cases such frameworks are available as binary libraries only and cannot be modified.

R.7 – Flexibility and Extensibility. The multimedia composition layer should be flexible enough to be capable of adapting characteristics of lower layers, e.g., particular processing mechanisms or resource requirements. This also includes means of extending internals in order to cope with previously unanticipated features.

R.8 – Performance. Processing performance of an underlying framework shall not be compromised beyond the effort required to modify its state.

In the course of this paper we will show how these requirements are actually satisfied in the design of DECKLIGHT. As we have shown in Section 2, numerous multimedia processing frameworks rely on graph-based processing mechanisms, and the graph-based approach is often the single commonality among several frameworks. Hence, employing a processing graph as a global structure was the most central and important design decision in DECKLIGHT, and the major step to fulfil requirement R.1.

We have already raised the question whether it is possible to integrate every framework in the global processing graph. While this is obviously possible for graph-based frameworks, we can, in the extreme case, integrate a whole framework in a single processing node – at the cost of losing the generality of the implementation. For example, this method has been applied when integrating a black-box framework such as a speech recognition engine, which takes audio data and configuration parameters as input and delivers analysis results as output.

Mapping the requirements to the concept of the global processing graph will leave us with the questions of how the graph needs to be organised in order to guarantee efficient and flexible operation,

what kind of operations are required for real-time graph manipulation, and how the graph is divided into subgraphs for representing underlying frameworks. The next section will discuss these issues in detail in terms of an architectural description of DECKLIGHT.

4. ARCHITECTURE

Figure 2 gives an overview of DECKLIGHT’s architecture and its position between the application layer and the underlying frameworks. The vertical split into two sublayers directly reflects the two categories of given requirements: The upper sublayer deals with providing uniform composition means to the application layer, whereas the lower sublayer deals with integration of underlying framework functionality and internal processing issues. In this section, tasks depicted by the arrows in Figure 2 will be thoroughly discussed. We start with the organisation of the processing graph, and how the application layer can issue operations on the graph. Then, and most importantly, we will show how the graph is segmented and mapped to requirements of each underlying framework. In addition, timing and error handling will be discussed.

4.1 Processing Nodes, Ports and Connections

As already mentioned, the processing graph is built of *processing nodes*, which can be connected. The connection points are called *ports*, and we distinguish between input ports for reading data, and output ports for writing data (from the perspective of a processing node). Ports are typed, and a connection can only be established between ports of the same type. DECKLIGHT currently distinguishes three basic port models: a) *Bus ports* are the most common type and are used for bus-style data transport. Input ports can be connected to at most one output port. b) *Queue ports* implement message semantics, where messages are queued at input ports. Finally, c) *Link ports* do not transport data at all but rather reflect a structural relationship of the underlying framework. An example are child–parent connections in 3D graphics scene graphs, where rendering state is implicitly transferred.

Basic port types are defined for bus and queue ports: Integer and floating point types, scalar as well as vectors of fixed and variable dimension, string ports, and time ports (see below). For instance, 32-bit floating point vector busses are used to transport frames in audio flow networks. In order to better reflect special requirements of underlying frameworks, arbitrary port types such as handles to resources or opaque framework data can be defined.

Before the application layer can start composing a processing graph, it needs to know about available processing nodes and their port configuration. Therefore DECKLIGHT maintains a processing node class repository, which allows for inquiry of registered processing node classes. As a result, class IDs for node creation, human readable textual information (name, description, etc.), and port lists can be obtained. This introspection mechanism leverages GUI-based applications as well as networked operation, where the application layer is replaced by a network handler, and be controlled through remote procedure calls [20].

4.2 Graph Composition

Once processing node and port information is obtained, the application layer may apply graph composition using a small set of operations:

1. *Create*. Creates a node of a given class ID and return a handle to the newly created instance.
2. *Destroy*. Destroys a processing node instance. The node must be previously disconnected.

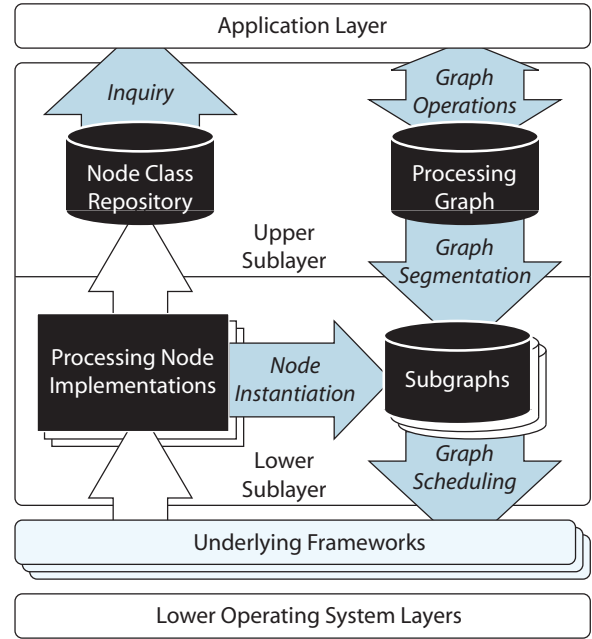


Figure 2: Architecture Overview. DECKLIGHT itself is split into two vertical sublayers.

3. *Connect*. Connects an output and an input port of given instances and returns a connection handle.
4. *Disconnect*. Removes a connection.
5. *SetInput*. Sets an input port to a provided value.
6. *GetInput, GetOutput*. Returns a value of a given input or output port, respectively.

Multiple operations can be collected in a list and be issued as a single, atomic *transaction* (similarly to the message bundling mechanism in Open Sound Control [24]):

```
transaction[] = ( disconnect[], destroy[], create[], connect[],
                 setinput[], getinput[], getoutput[] )
```

where [] depicts a list of variable length. The given order prevents consistency problems, for example it guarantees that instances cannot be created and instantly destroyed in the same transaction.

Clearly, in a real-time system, atomic execution of multiple state changes is crucial: For instance, if an application needs to modify multiple parameters of an audio filter processing node, all changes are required to be committed at the same time, otherwise undesired (and audible) effects will result. Transactions fulfil of requirement R.2 (Consistency), and we will show how they can be implemented without interrupting real-time operation in the next two sections.

4.3 Graph Segmentation

Up to now, we have dealt with the upper sublayer in DECKLIGHT. As the processing graph is now available for processing, the main issue is how to map it to the multitude of underlying frameworks. The short answer is: Segment the processing graph into subgraphs, one for each framework, as indicated in Figure 2. Unfortunately, the realisation of the short answer is much more complex, and our proposed solution is the central contribution of this paper, as it enables an unified view across multiple underlying

frameworks, while respecting their internal characteristics (R.6. – Non-Intrusive Integration).

Instead of running the whole processing graph synchronously in a single thread of execution, DECKLIGHT allows processing nodes either to be *bound* to a thread assigned to the underlying framework, or to be *unbound*, i.e., they are capable to be processed in the scope of several, or even arbitrary frameworks. For example, nodes of a scene graph typically remain bound to the thread¹ provided by the scene graph framework, since they must use rendering state information, which cannot be shared across multiple threads. In contrast, processing nodes that perform generic tasks, such as mapping values between ports, are not required to run in the domain of a specific framework, and are therefore unbound.

Similarly to W [7], this schema adapts the hardware concept of a *globally-asynchronous, locally-synchronous (GALS)* system [17] in the software domain: The processing graph as a whole consists of multiple synchronous islands, which run asynchronously against each other. Inside a synchronous island, the subgraph of bound processing nodes is transformed into a sequential list, which is processed by the corresponding framework-assigned thread. In DECKLIGHT, such threads are called *schedulers*, referring to the well-known operating system concept.

As we deal with a directed graph (with cycles allowed), the algorithm for subgraph construction requires the presence of *root processing nodes* as a starting point. Root processing nodes are typically those nodes in an underlying framework where processing is targeted at, e.g., a sink in an audio flow graph, or a viewport in a scene graph. An additional requirement is a precedence value assigned to schedulers. It defines which scheduler takes priority when multiple schedulers attempt to bind an unbound processing node. Algorithm 1 introduces a graph segmentation method that not only identifies the subgraphs for each scheduler, but also binds unbound processing nodes, resolves graph cycles, and transforms the subgraphs into sequential lists ready for processing (Section 4.4).

We will illustrate how Algorithm 1 works by the use of Figure 3, where the white nodes could depict a hypothetical audio flow graph, the black nodes a scene graph, with attached sequencing nodes in grey. In the Figure, the nodes $P_0 \dots P_3$ can only be scheduled by the “white” scheduler (which has P_0 as its root node), and the nodes $Q_0 \dots Q_3$ are restricted to the “black” scheduler. The nodes R_0 and R_1 are unbound, and may be taken by white or black. For now, let us assume that the black scheduler has a higher precedence than the white one, thus R_0 and R_1 will end up in the domain of black scheduler. If Algorithm 1 is applied to P_0 and Q_0 (in either order), the resulting processing lists are:

$$\begin{aligned} list[white] &= (P_2, P_1, P_3, P_0) \\ list[black] &= (R_1, Q_3, R_0, Q_2, Q_1, Q_0) \end{aligned}$$

The sequence in each list may change depending on how the `incomingConnections` list is ordered. However, the returned processing sequence is correct invariantly of these differences. For instance, $list[white] = (P_3, P_2, P_1, P_0)$ would result if the incoming connection $P_3 - P_0$ was processed before the connection $P_1 - P_0$.

The method `isSchedulableBy()` allows processing nodes independently to define whether they allow themselves to be bound by a particular scheduler.

The given algorithm constructs subgraphs from scratch for an existing processing graph. There are two problems with this scheme: First, incoming transactions are either processed in the application

¹A scene graph framework may use multiple threads internally for processing, but normally provides one thread for control.

```

List[] Segmentation::getSequences(List schedulers) {
    List[] lists;
    foreach (s in schedulers)
        lists[s] = s.getSequence();
    return lists;
}

List Scheduler::getSequence() {
    List l = empty;
    this.cycle++;
    foreach (rootnode in this)
        this.getSequence(l, rootnode);
    return l;
}

void Scheduler::getSequence(List& l, Node n) {
    // can we schedule the node?
    if (!n.isSchedulableBy(this)) return;
    // can we take it?
    if (n.scheduler != null &&
        n.scheduler.precedence() >
        this.precedence()) return;
    // have we been here?
    if (n.cycle == this.cycle) return;

    // ok, the node is ours. now recursively walk graph
    n.cycle = this.cycle;
    n.scheduler = this;
    foreach (connection in n.incomingConnections)
        getSequence(l, connection.sourceNode)

    // finally append original node to list
    l.append(n);
}

```

Algorithm 1: Graph segmentation (C++-style pseudocode). The algorithm returns processing sequences for each scheduler.

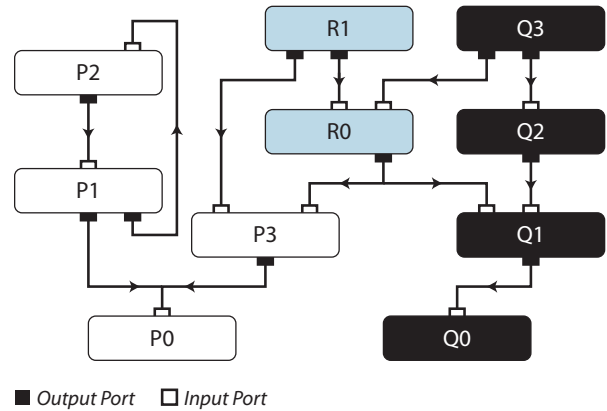


Figure 3: Subgraph example. The nodes P_n and Q_n are bound to a separate scheduler, whereas R_n are unbound (if not connected).

thread, or in a dedicated thread, whereas schedulers process subgraphs using their own threads. If we updated the graph atomically, all schedulers needed to be stopped at a synchronisation point (more on this in Section 4.4), where the new subgraphs are passed to the schedulers. However, since schedulers may run at completely different rates (with respect to available synchronisation points), stopping all schedulers at once would dramatically interfere with

real-time processing, in particular with real-time audio processing. Therefore, the subgraphs must be double-buffered, so they can be passed to each scheduler independently. Obviously, synchronisation points are still required, but not coevally. This scheme allows for processing of large transactions without interrupting ongoing processing.

Second, incoming transactions reflect differential changes to an existing subgraph, and instead of completely rebuilding each graph, we may adapt Algorithm 1 to a version which can deal with differential changes. Therefore, we first need to examine how each transaction operation effects on the existing graph. Clearly, the *SetInput*, *GetInput*, and *GetOutput* operations have no influence on the processing graph’s structure. The same applies to *Create*, and *Destroy*: After its creation, a processing node is not connected yet, and upon destruction, a processing node is required to be disconnected.

Thus, our candidates are *Connect* and *Disconnect*. One of the main issues with differential graph connects and disconnects is that they may affect more than just the two nodes of interest. For instance, assume that the connection $R0 - Q1$ in Figure 3 is removed. Then, the corresponding processing lists change from

$$\begin{aligned} list[white] &= (P2, P1, P3, P0) \\ list[black] &= (R1, Q3, R0, Q2, Q1, Q0) \end{aligned}$$

to

$$\begin{aligned} list[white] &= (P2, P1, R1, R0, P3, P0) \\ list[black] &= (Q3, Q2, Q1, Q0) \end{aligned}$$

after rerunning Algorithm 1, i.e., processing nodes may be relocated from one scheduler to another by connecting or disconnecting graph edges.

Finally, Figure 3 also indicates that there remain connections across different schedulers, e.g., the connections $R1 - P3$ and $R0 - P3$. Again, since we deal with asynchronous schedulers, we need to take special care for such connections, as we will see in the next section.

4.4 Scheduling and Processing

In the previous section, we have shown how a globally-asynchronous processing graph can be split into locally-synchronous subgraphs. Here, we show how the subgraphs are injected into the processing stream of each underlying framework and how asynchronous connections between schedulers are dealt with.

Typically, processing in any framework can be classified into either a event-based loop, where the framework’s main thread sleeps until it is invoked by an external or internal event, or a constant-rate processing loop, which is the normal case for audio flow processing or fixed-framerate video processing. In either case, if we need to modify the framework’s state (a part of which is represented by a subgraph), a *guarded section* where the framework allows for consistent state changes is required. For event-based processing, this section is defined by the sleeping state, and for loop-based processing we may update the state at the point, where a processing-loop is complete (e.g., between audio or video frames). The requirement of providing a guarded section is not imposed by DECKLIGHT, it is rather a requirement which holds for any piece of code that needs to consistently update state of an asynchronously running process – if an underlying framework needs to be stopped before updating it, it will not be useful for any kind of real-time processing.

It is in the guarded section, where DECKLIGHT directly interacts with an underlying framework. As discussed in the previous section and illustrated in Figure 4, we strictly adhere to a multithreaded

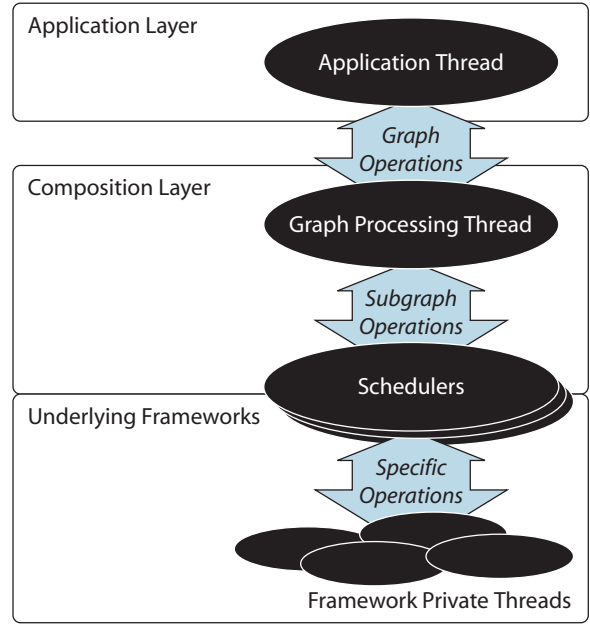


Figure 4: Multithreaded setup in DECKLIGHT. The scheduler threads are created by the composition layer but are running in the context of the underlying framework.

environment: Each underlying framework runs in its own scheduler thread, and, if needed, may run a number of additional threads. The graph processing thread passes graph updates – all operations, except processing node creation and destruction as they are performed directly by graph processing – to individual schedulers, which process them as given in Algorithm 2 in their guarded section: First, input ports are updated with incoming packets², and input data is fetched from asynchronous incoming connections. Then, the scheduler iterates through the ordered processing list, which is previously generated by graph segmentation (Algorithm 1), fetches data from synchronous incoming connections (i.e., those that connect processing nodes *inside* a given scheduler), and calls `cycle()` for each processing node. The `cycle()` method is very central to each processing node, as it updates the system state and builds the glue between DECKLIGHT and an underlying framework. After the loop, data is pushed into asynchronous outgoing connections, and packets for *GetInput* and *GetOutput* operations are created.

So far, we have not discussed how data is transferred through connections. Clearly, since connections are typically allowed to be established across scheduler borders, they need internal means for integrity of the data flow. At the same time, optimisation for in-scheduler connections, e.g., zero-copy, is desirable in order to avoid performance penalties. Thus, DECKLIGHT’s transfer mechanisms are specific to the given port models, and for the same model there is one method for asynchronous and one for synchronous transfers. For *bus ports*, triple buffering is applied for asynchronous connections, with guarding the middle element against the two schedulers. If the connection is synchronous, zero-copy single buffering can be applied, and there is no need to add the connections to the incoming synchronous connections list in Algorithm 2. For *queue ports*, the same applies to asynchronous connections, but in addition, a syn-

²Port values are encapsulated in packets to simplify queuing and serialisation.

chronous transfer is required in order to pass the queue from the output to the corresponding input port.

Obviously, asynchronously transferring data on busses imposes a problem related to the fact that the schedulers may be running at different rates: If the output port's scheduler runs at a similar rate than the input scheduler, aliasing effects may occur, and individual port writes are lost if the output rate is higher than the input rate. DECKLIGHT approaches this problem by providing special asynchronous connections for busses, such as averaging, minimising or maximising values if multiple writes occur between two read operations. For instance, these types of connections are applied when transferring analysis data such as audio levels or spectral characteristics from high-rate audio processing to medium-rate graphics processing.

Finally, processing nodes may have their private background threads, for example for loading of large resources via the network, where the synchronous, typically constant-rate operation of the frameworks is not allowed to be interrupted. Here, DECKLIGHT does not enforce particular mechanisms and the processing node is responsible for handling asynchronous data between its scheduler's thread and the background thread.

4.5 Dealing with Time

Timing and sequencing issues are ubiquitous in interactive multimedia systems. They are found at various levels ranging from coarse-grained time control structures to microscopic synchronisation mechanisms. In computer graphics, time is most often associated with animation, and animation curves are applied for time-dependent control of objects and their parameters. In music, time is typically bound to a linear score, but with a closer look, musical time can become very complex and be organised in hierarchical structures, as it is the case in a *Chopin Rubato* [14]. Finally, in audio processing, timing mechanisms at the precision of individual samples are normally required. This variety of timing issues boils down to specific mechanisms in each underlying framework, and when attempting to provide comprehensive support for timing in a generic composition layer, mapping and transformation of different time concepts has to be accomplished.

In DECKLIGHT, time is de-coupled from framework-specific timing mechanisms by employing *time ports*. Each processing node that has an explicit notion of time (i.e., its state is time-dependent with respect to multiple calls to the `cycle()` method), requires at least one input time port. This method has several advantages: First, an application can easily distinguish between processing nodes that are time-dependent and those that are independent of time. This is typically useful when providing a user interface that allows the user to quickly identify time-related structures. Second, time ports allow for the construction of complex time processing graphs (similar to MET++'s temporal hierarchies [1]), where processing nodes create or process time that is finally delivered to processing nodes that require time for their operation. Employing a graph for timing takes us away from the linearity of a one-dimensional sequencer and allows for timing control at all levels. For instance, it allows a combination of a multitude of time sources, such as fixed clocks, rhythm analysis results, or discrete asynchronous events. Third, the input type of time ports does not have to be a physical time, but can be a logical (or symbolic) time with a given unit, as it is applied in musical time in terms of beats at a given tempo. The logical time can then easily be mapped to the internal physical time required by each underlying framework. Finally, time ports can effectively be applied together with ports that take curves as input in order to deal with animation curves and envelopes for sound processing.

```
void Scheduler::update() {
    // first, process set inputs
    foreach (packet in this.portListIn)
        packet.port().set(packet.data());
    // update asynchronous incoming connections
    foreach (connection in this.asyncConnectionsIn)
        connection.asyncLoad();

    // cycle through each node in ordered processing list
    foreach (node in this.processingList) {
        // update synchronous input connections for node
        foreach (connection in node.syncConnectionsIn)
            connection.syncLoad();
        // allow node to update state
        node.cycle();
    }

    // update asynchronous outgoing connections
    foreach (connection in this.asyncConnectionsOut)
        connection.asyncStore();
    // finally, process get inputs / outputs
    foreach (packet in this.portListOut)
        packet.data = packet.port().get();
}
```

Algorithm 2: Scheduler guarded section (C++-style pseudocode). In this section, DECKLIGHT interacts with the underlying framework, and graph and connection updates for every scheduler take place.

4.6 Error Handling

In a real-time system, non-critical errors need to be handled gracefully without interrupting the processing flow. Errors should be collected and reported to the application layer, which then may take corrective actions. With the given architecture, we can distinguish between *logical errors* that occur at transaction level, and *runtime errors* that occur during processing in the underlying framework.

DECKLIGHT's graph processing thread (as shown in Figure 4) catches logical errors by analysing incoming transactions. Typical types of errors are illegal processing node handles, illegal port IDs, connection type mismatches, or destruction of processing nodes that are still connected. If an individual operation fails, it is marked with an error signature and the transaction is sent back to the application. In this case, DECKLIGHT's system state is not modified.

Runtime errors (e.g., resource allocation errors, invalid port values, or processing overload) are reported via error ports. Values from error ports can either be collected by the application directly, or can be wired into a separate error subgraph, where specific processing nodes handle incoming errors. Similarly to the time graph introduced in the previous section, an error graph can flexibly reflect an application's need for dealing with runtime errors.

5. REALISATION

DECKLIGHT, as described in the previous section, is implemented in Standard C++ and makes use of many modern C++ concepts such as type traits for port implementations or automatic memory handling for management of asynchronous objects [2]. The realisation of the composition layer pays particular attention to requirement R.7 (Flexibility and Extensibility) and provides specific programming interfaces at different integration levels.

At the lowest level, DECKLIGHT provides an interface to integra-

tors of underlying frameworks. As we have seen earlier, they typically require a separate scheduler running in its own thread. Thus, an actual integration needs to cope with scheduling peculiarities and with particular connection types, as they appear for example as scene graph child-parent relationships in a graphics framework. At this level, considerable expertise is required with respect both to internals of the underlying framework, as well as to the processing graph segmentation and scheduling mechanisms described in Section 4.

The intermediate level deals with implementing processing nodes that map processing functionality to the underlying framework, and provides a programming interface for easy processing node creation and port access. Basically, for each processing node, a separate class needs to be inherited from the processing node base class (or a derivative of it). The class defines the input and output ports, and the `cycle()` method for state updates. Through C++ operator overloading, ports can basically be accessed in the same way as normal member variables, and scheduling and data integrity issues are completely hidden.

At the highest level, the transaction processing interface is provided to applications that make use of the multimedia composition layer. As indicated in Figure 2, the API basically consists of mechanisms to initialise the graph processor and the underlying frameworks, means for inquiring the node repository, operations for transaction assembling and execution, and for error handling from an application perspective.

Obviously, integrating frameworks does not come for free, since the mapping of underlying functionality to provided functionality must be explicitly defined through processing nodes and their port definitions. While the mapping provides a powerful mechanism of consolidation of underlying processing facilities, it requires at the same time considerable design decisions in terms of granularity: For example, when implementing processing nodes for OpenGL, one could decide to directly map OpenGL function calls to processing nodes, and their parameters to corresponding input ports. While this very fine granularity might make sense for educational purposes or OpenGL experiments, it would be rather limited in terms of performance (as every OpenGL call is encapsulated in a processing node). Thus, a more coarse-grained implementation, combining multiple OpenGL calls into dedicated processing nodes will deliver better results, both in terms of performance and usability, since the mapping provides a more high-level view on well-known computer graphics concepts. An example would be to provide a node that draws a shape, and also includes colour, texturing and pre-transformation parameters.

Finally, the implemented processing nodes can be linked together with the underlying framework's libraries and the graph processing and scheduling libraries and are ready for use by application code.

5.1 Integrated Frameworks

Currently, DECKLIGHT integrates a number of frameworks of the audio and graphics processing domain: For audio processing, audio I/O for the most common platforms is available, as well as standard filters and digital audio effects. In terms of graphics processing, there is an integration of a custom graphics and video processing framework based on OpenGL, as well as a proof-of-concept integration of *OpenGL Performer* [19]. The latter was realised in order to examine how well our model fits for external frameworks that enforce individual characteristics, e.g., for scheduling. In addition, a proof-of-concept integration of *sh* [15], a high-level metaprogramming language for GPUs (Graphics Processing Units) has been successfully accomplished.

In addition to the frameworks for audio and graphics processing there is support for MIDI processing, such as MIDI I/O and filtering of MIDI messages. Further, there are nodes for HID devices such as keyboard and mouse input, and nodes for Bluetooth networking.

Besides these specialised frameworks, there is a large set of generic processing nodes, which are not directly related to a particular underlying framework. Among others, the set includes nodes for time and error processing, for port type conversions, and for port range mappings (e.g., when transferring an audio level to an image scale factor).

DECKLIGHT runs on Linux and Mac OS X, and work on a Windows implementation is currently being carried out.

6. DISCUSSION

The system, as it is currently implemented, has been employed for a wide range of real-time media processing applications. One example is the *Digital Marionette* art installation, which is the realisation of a puppet in the digital domain. The puppeteer can control the marionette using traditional puppet handles and speech input: The handles' positions are digitised and processed in order to control facial expression of an over-dimensioned puppet head [12]. Speech recognition is applied in order to extract phonemes which are then mapped to *visemes*. All input data is retrieved and processed by DECKLIGHT and then applied to render the 3D head and to play back lip-synchronised voice.

Another example is real-time processing of data from mobile phones such as SMS (Short Message Service), MMS (Multimedia Message Service), and Bluetooth data in public spaces: DECKLIGHT uses short range Bluetooth dongles in order to detect devices in range, and collects SMS and MMS messages and content sent via Bluetooth in order to project them on large scale screens.

A third example is in the domain of music-synchronised live visuals applications, where real-time audio analysis is applied in order to extract audio features that can be interactively mapped to visual content [8, 9].

While each of above examples could have been realised without DECKLIGHT by hard-wiring existing frameworks, we have observed a drastic speed-up in the development cycle of each of the mentioned projects. In particular, DECKLIGHT not only enabled *rapid prototyping* for initial steps, but also leveraged *rapid product development*, as the final product was directly reused from the prototype with minor changes only. In addition, since DECKLIGHT acts as a layer that can be plugged into any application, resulting products are completely de-coupled from any interactive authoring environments (as it would be the case for Max/MSP).

With an eye on the presented design and realisation, the concept of asynchronous subgraphs for each underlying framework has proven to be both convenient and efficient, in particular with respect to current developments in CPU architectures, where parallelism is constantly increased. While multi-threading at the beginning requires a careful design especially for data integrity and memory management, it considerably simplifies the integration of additional frameworks. Here, a substantial contribution are DECKLIGHT's port and connection mechanisms, which mostly hide multi-threading issues, allowing non-expert C++ programmers to implement processing nodes. Experience has shown that bus ports are sufficient in the majority of cases: First, bus ports are well suited for synchronous flow graphs, and second, in case of crossing asynchronous frameworks borders, DECKLIGHT offers special means of dealing with data integrity (as shown in Section 4.3). In contrast, queue ports are more critical in terms of overflows and memory management. However, they still can be applied in a convenient manner, where absolutely needed.

7. CONCLUSION AND FUTURE WORK

We presented DECKLIGHT, a real-time multimedia composition layer, which unifies graphics, music, sound processing, and basically any type of input or output media, by employing a global processing graph. The graph's processing nodes are responsible for the mapping of application data and state management of a dedicated part of the underlying framework. The layer operates by automatically subdividing the graph in to synchronous subgraphs which run asynchronously (i.e. multithreaded) with respect to each other.

Besides of the technical aspect of uniformly handling multiple media types, the concept of a global processing graph allows non-experts to exploit expert knowledge (which is embedded in the processing nodes) in order to design, implement and deploy real-time multimedia processing applications at product level.

Currently, DECKLIGHT's port subsystem is designed for running on a single host and transporting network data is accomplished by specialised ports. The design and realisation of a multi-host port and connection system is certainly an attractive domain for future work, as it will lift the the idea of a generic multimedia composition layer to the network level. In addition, as the amount of processing nodes is constantly growing, and the applications in terms of realised processing graphs are getting bigger and more complex, we currently focus on high-level composition mechanisms, which operate on graph semantics. The ultimate goal here will be to complement the simple graph modification operations with intuitive and powerful computer-aided design transformations.

8. ACKNOWLEDGEMENTS

We thank Jürg Gutknecht of ETH Zurich and Christoph P. E. Zollikofer of University of Zurich for supporting our work. This work was supported in part by Swisscom and by Corebounce association. Additional thanks are due to the reviewers for their valuable comments.

9. REFERENCES

- [1] P. Ackermann. Direct manipulation of temporal structures in a multimedia application framework. In *MULTIMEDIA '94: Proceedings of the Second ACM International Conference on Multimedia*, pages 51–58. ACM Press, 1994.
- [2] A. Alexandrescu. *Modern C++ Design*. C++ In-Depth Series. Addison-Wesley, 2001.
- [3] X. Amatriain. *An Object-Oriented Metamodel for Digital Signal Processing with a Focus on Audio*. PhD thesis, Universitat Pompeu Fabra, Barcelona, Spain, 2004.
- [4] Apple Computer – Core Audio. <http://www.apple.com/macosx/features/coreaudio>, 2006.
- [5] Apple Computer – Core Image. <http://www.apple.com/macosx/features/coreimage>, 2006.
- [6] R. B. Dannenberg and E. Brandt. A flexible real-time software synthesis system. In *Proceedings of the 1996 International Computer Music Conference*, pages 270–274. International Computer Music Association, 1996.
- [7] R. B. Dannenberg and D. Rubine. Toward modular, portable, real-time software. In *Proceedings of the 1995 International Computer Music Conference*, pages 65–72. International Computer Music Association, 1995.
- [8] J. Foote, M. Cooper, and A. Girgensohn. Creating music videos using automatic media analysis. *Proc. ACM Intl. Conf. on Multimedia*, pages 553–560, 2002.
- [9] M. Goto. An audio-based real-time beat tracking system for music with or without drum sound. *Journal of New Music Research*, 30(2):158–171, 2001.
- [10] R. E. Johnson. Frameworks = (Components + Patterns). *Communications of the ACM*, 40(10):39–42, 1997.
- [11] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing*, pages 471–475, 1974.
- [12] G. A. Kalberer, P. Müller, and L. Van Gool. Visual speech, a trajectory in viseme space. *International Journal of Imaging Systems and Technology*, 13(1):74–84, 2003.
- [13] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–799, 1995.
- [14] G. Mazzola. *The Topos of Music*. Birkhäuser, Basel, 2002.
- [15] M. McCool, S. D. Toit, T. S. Popa, B. Chan, and K. Moule. Shader algebra. In *Proceedings of ACM SIGGRAPH 2004 / ACM Transactions on Graphics*, volume 23(3), pages 787–795. ACM, 2004.
- [16] P. Müller, S. Müller Arisona, S. Schubiger-Banz, and M. Specht. Interactive media and design editing for live visuals applications. In *Proceedings of the International Conference on Computer Graphics Theory and Applications (GRAPP)*, pages 232–242, 2006.
- [17] J. Muttersbach, T. Villiger, and W. Fichtner. Practical design of globally-asynchronous locally-synchronous systems. In *Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 52–59, 2000.
- [18] M. Pukette. Max at seventeen. *Computer Music Journal*, 26(4):31–43, 2002.
- [19] J. Rohlf and J. Helman. Iris performer: A high performance multiprocessing toolkit for real-time 3d graphics. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 381–395. ACM, 1994.
- [20] S. Schubiger and S. Müller. Soundium2: An interactive multimedia playground. In *Proceedings of the 2003 International Computer Music Conference*. International Computer Music Association, 2003.
- [21] P. S. Strauss and R. Carey. An object-oriented 3D graphics toolkit. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, volume 26(2), pages 341–349. ACM, July 1992.
- [22] D. Tanguay, D. Gelb, and H. H. Baker. Nizza: a framework for developing real-time streaming multimedia applications. In *HPL-2004-132*, 2005.
- [23] A. Weinand, E. Gamma, and R. Marty. Et++ – an object oriented application framework in c++. In *OOPSLA '88: Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 46–57. ACM Press, 1988.
- [24] M. Wright and A. Freed. OpenSoundControl: A protocol for communication with sound synthesizers. In *Proceedings of the 1997 International Computer Music Conference*. International Computer Music Association, 1997.
- [25] R. C. Zeleznik, D. B. Conner, M. M. Wloka, D. G. Aliaga, N. T. Huang, P. M. Hubbard, B. Knep, H. Kaufman, J. F. Hughes, and A. van Dam. An object-oriented framework for the integration of interactive animation techniques. In *Computer Graphics (Proceedings of SIGGRAPH 91)*, volume 25(3), pages 105–112. ACM, 1991.