Journal of Visual Languages and Computing (2001) **12,** 283–297
doi:10.1006/jvlc.2001.0209 available online at http://www.idealibrary.com on IDEAL®

A Visual Programming Language for Expressing Rhythmic Visuals

FRED COLLOPY* AND ROBERT M. FUHRER†

**Case Western Reserve University, 693 Enterprise Hall, 10900 Euclid Avenue, Cleveland, OH 44106-7235, U.S.A., E-mail: collopy@po.cwru.edu, http://RhythmicLight.com,*
†Computer Music Center, IBM T. J. Watson Research Center, Route 134, Yorktown Heights, NY 10598, U.S.A., E-mail: rfuhrer@watson.ibm.com

Received 24 February 2000; revised 14 July 2000; accepted 2 February 2001

Sonnet + Imager is an object-based toolkit for creating instruments that produce abstract graphics in real-time. It is implemented as a visual programming language of the component-circuit variety. It was designed by identifying and addressing some of the principle limitations in the Max-based graphics engine, Imager. Beyond that, we wanted to address rhythm directly. This required us to make time a first-class element of the language. The model of time relies for its power on the notion of functors, an encapsulation of mathematical functions that can be related to the dimensions of rhythm. All the elements of time manifest themselves directly in the visual language, as components and data packets, thereby creating natural flows that describe rhythmic structures. The resulting design is modular, intuitive, interactive and extensible.
© 2001 Academic Press

1. Introduction

FOR CENTURIES INVENTORS, painters and film-makers have attempted to create dynamic color graphics that are in some sense the visual equivalent of music. Early experimenters had to cope with difficulties related to illumination, color purity and control. The abstract film-makers of the early 20th century often engaged in tedious and time-consuming processes to produce even short films. Survage created over 200 sketches, but never completed a film. Fischinger spent weeks cutting out figures for one of his films. The Whitneys invested years building special equipment to realize their ideas. In short, the lack of appropriate tools severely limited their productivity, and kept a universe of visual possibilities out of reach.

Today, powerful affordable computers permit the creation of a new breed of graphic instruments. Moreover, MIDI controllers make it possible to develop instruments that permit a player to control complex graphic pieces in real-time, an objective of the inventors of early light organs and some of the modern artists.

This paper describes a visual language and an interactive environment for the creation of dynamic visual performances. Specifically, we describe how two software development efforts are being fused to create an integrated environment for sonic and visual music. The first project, Sonnet, is a visual language for implementing real-time processes. It was designed as a visual programming language of the component-circuit variety, and has been used in computer music applications. The other project, Imager, is an engine for generating abstract graphics in real-time. We refer to the integration of the two as Sonnet + Imager.

We use the term lumia to denote a visual art that is about color, abstract form and movement. This term was introduced in 1947 by Thomas Wilfred, who invented one of the most important visual instruments of his time. Like 'music', the name is simple, short and of meaningful root. The instruments to play lumia can assume a wide variety of forms and functions. Indeed, there is no reason to expect less variety than there is among musical instruments. Some of the instruments will be designed to be played improvisationally. We envision a future in which lumianists join with musicians, assuming a position as a full-fledged member of an ensemble, or where groups of them 'jam' together.

The paper starts with a brief history of the development of lumia and instruments to create them over the last two centuries. We then identify the design goals for the kind of visual instruments of which Sonnet + Imager is intended to permit construction. We elaborate with a motivational example, which illustrates the kind of visual language we are aiming for. Next, we describe the earlier implementation of Imager, along with the limitations that drove us to re-design and re-implement it. Sonnet is described in the following section, in which we explain what properties make it a suitable context for exposing the functionality of the new Imager design. Finally, we describe Sonnet + Imager, the encapsulation of Imager in the Sonnet visual programming language, with particular emphasis on what the combination offers that the original Imager did not.

2. A Brief History of Lumia

Interest in the design of instruments to play graphics in the way that musicians play with sounds is not new. The idea for such instruments can be traced at least to the early 18th century when Louis-Bertrand Castel, a Jesuit priest, designed his *clavecin oculaire*. Through the remainder of that century and the next a variety of light organs were designed. These instruments were often premised on the idea that particular notes had some direct relationship to particular hues. While this notion has not withstood the test of time very well, the more general idea of controlling colored light and relating it to music has endured.

Early in the 20th century, abstract artists came to be interested in both color and form for their own sake. Many of these abstract artists were also intrigued by the relationship of their painting to music. Frantisek Kupka, an artist of the Section d'Or in Paris, declared 'I believe that I can find something between sight and hearing and I can produce a fugue in colors as Bach has done in music (quoted in [1] p. 108).' At the Bauhaus, Wassily Kandinsky, Paul Klee, Laszlo Moholy-Nagy and others wrote about how painting colors and forms was like making music. They even created plays and kinetic light machines to explore their ideas.

Music is an art embedded in time. Leopold Survage, a Cubist painter who moved in circles that included Matisse, Picasso, Braque, Modigliani and Leger, was among the first, both to articulate the limits of abstract art and to see the potential of cinematography for creating a new art. 'An immobile abstract form does not say much...it is only when it sets in motion, when it is transformed and meets other forms, that it becomes capable of evoking a feeling (quoted in [2], p. 412).' By adding the dimension of time to abstract painting, Survage put a third element—rhythm—in place. 'I am creating a new visual art in time, that of colored rhythm and of rhythmic color (quoted in [3], p. 36).' Through the remainder of the 20th century, film makers such as Oskar Fischinger, John and James Whitney, Mary Ellen Bute and Harry Smith explored this idea, developing vocabularies and principles for a new art, an art of abstract motion in time, an art that integrates sound and graphics.

3. Design Objectives

In many respects this project can be viewed as improving and extending ideas that were implemented in Imager, which will be described in more detail in the next section. Imager is a graphic engine designed to generate abstract images in real-time. It is based on the constructivist aesthetic model, in which potentially complex images are built up from simple graphic elements.

We had four principal design objectives for the new Imager architecture. First, we wanted to devise a more coherent conceptual framework for expressing rhythmic visuals than was present in the earlier Imager design. Second, through the design of more intuitive and powerful controls we wanted to better expose key aesthetic properties and thereby create a more useful environment for the artist. Third, we wanted the time dimension to be more tightly integrated with both the aesthetic and the rendering models. And fourth, we wanted an extensible architecture; one that serves as a toolbox of objects and building rules from which designers and artists can construct new instruments.

The extensible character of the architecture would manifest in two ways. It should permit the construction of complex forms and behaviors from simpler ones and it should permit the addition of fundamentally different types of forms and different parametric modulations. As we have said, Imager is based on a constructivist aesthetic model. Our design objective included allowing the substitution of alternative aesthetic models such as fractals, movie clips, captured still images and cellular automata.

4. Imager

Imager was designed to function as an instrument for 'playing' graphics. It is constructivist in the sense that graphic scenes are created by combining basic graphic elements, such as points, lines and polygons. This simple set of objects can generate surprisingly complex and subtle textures, as shown in Figure 1.

For each visual object, attributes describing color, form and movement are specified. Object attributes can be combined into groups and stored as 'presets'. A player can then use MIDI controllers to trigger and manipulate the presets. MIDI note events can be configured to trigger the appearance of a given set of visual objects, or changes in some of their visual parameters such as line thickness or hue.

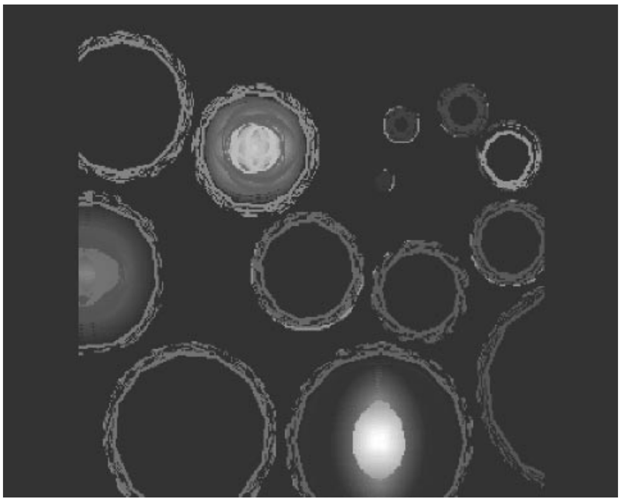
Figure 1. A single frame from a visual performance

Although many aesthetics could serve as a basis for visual experience, the constructivist aesthetic (employed by such modern artists as Kandinsky, Klee, Max Bill and Karl Gerstner) has a close historical link with efforts to integrate visual and sonic experiences. For them and other modern artists, *form* and *color* assumed dominant roles in expressiveness. Form is built up from simple elements, such as points, lines, planes, angles and conic sections. Color is interesting in its own right, not merely as a way of rendering objects in the world.

For the purposes of lumia, however, color and form are incomplete; lumia must deal with color and form *in time*. Hence, in Imager's model, a third element, rhythm, describes changes in these two. As mentioned earlier, the painter Leopold Survage believed that colored visual forms could play a role analogous to that of sound in music, and that these forms could be described by three factors: color, form proper and rhythm. In fact, these three elements (color, form and rhythm) completely describe the space of dynamic visuals. Additionally, rhythm provides the means by which graphics and music can be linked, both compositionally and improvisationally.

For each of these three domains, color, form and rhythm, it is necessary to decide how composers and players will control them. In effect, one needs to determine what 'knobs' will be available. In making these decisions, Imager's design was guided by aesthetic considerations wherever possible. The choice of the hue, saturation and value (HSV) color model serves to illustrate.

Hue is the principal way in which one color is distinguished from another. Describing and managing hues is generally taken to be the central problem of color theory. Indeed, the very language we use to denote colors is associated primarily with their hues. A hue can be referenced by its angle around a color wheel, for example, red at 0, yellow at 60, green at 120, blue at 240 and purple at 300. In a well-balanced color wheel, complementary colors appear at 180-opposite positions.

Saturation describes how pure a particular hue is. It is also referred to as the intensity, strength or chroma of a color. A particular hue becomes less saturated by mixing gray with it. Reducing saturation at a constant value has the effect of adding white pigment, producing what artists call *tints*.

Value is the quality that differentiates a light color from a dark one. It is also referred to as lightness. A particular color moves toward black by a reduction in its value. Low-valued colors are less visible than higher valued ones. Decreasing value while leaving saturation alone has the effect of adding black pigment, producing what are referred to as different *shades*. Finally, what artists refer to as *tones* can be created by decreasing both saturation and value. There is a substantial literature that uses these concepts to describe art history and technique.

Decisions about Imager's color model were, in short, driven by artistic considerations. The hue, saturation, value (HSV) color model was chosen for Imager because these concepts and the related concepts of tint, tone and shade have artistic meaning. A similar approach was taken to design Imager's other components.

The version of Imager that served as the starting point for the collaboration here was implemented to run under Max on the Macintosh. While it is capable of producing a significant variety of visuals, as illustrated by the color images located at *http://RhythmicLight.com*, the structure of that version makes it difficult to extend and to create new varieties of visual instruments. In particular, despite the fact that Imager was implemented on top of an interactive programming environment (Max), that environment's malleability does not directly empower the artist. Rather, Imager's functionality is contained primarily in one monolithic object and one monolithic Max circuit. To extend the functionality requires understanding and extending that object, and making corresponding changes to the Max circuit in which it is embedded. Beyond that, several of its features are encumbered by hard-coded constraints on the number and, more importantly, the behavior of various kinds of objects. Finally, it is tightly tied to both the platform and the environment in which it runs. These limitations were the motivation behind the effort that produced Sonnet + Imager.

5. Sonnet

Sonnet was originally designed as a visual environment for associating runtime actions with running programs. It has since evolved into a visual programming language for the rapid development of real-time applications [4]. Sonnet uses a circuit metaphor, as shown in Figure 2, and embodies event-flow semantics. Data exist in Sonnet as quanta known as 'packets'. Sonnet behavior is expressed in two forms: as primitive 'components' and as 'circuits', which are interconnections of components. The programming activity in Sonnet consists of constructing different arrangements of components into circuits that perform some computation.

Components are entities that have (1) a set of strongly typed input and output 'ports' through which data packets flow, and (2) an Execute() method. Any input port may be designated as a 'trigger'. A trigger input causes its owning component's Execute() method to be invoked whenever that input receives a data packet. A component may maintain additional state data; however, this is purely a matter for the component's internal implementation.

As shown in Figure 2, components are interconnected using 'wires' which attach one output port to one or more input ports. It is permissible for a component to have no

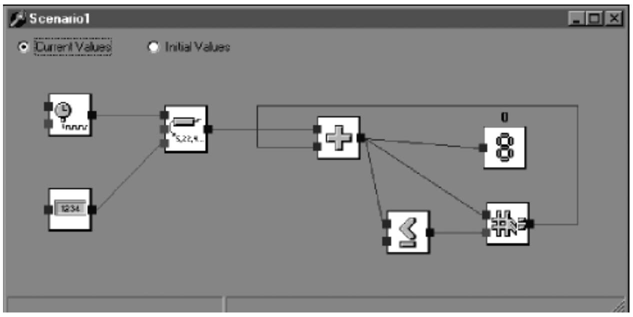

Figure 2. A program expressed as a Sonnet circuit

inputs or to have no outputs (as is often the case with interface components which act as one-way gateways to external devices).

A circuit can collapse into a single component, known as a 'chip', and be used in the same manner as a primitive component. Chips allow the programmer to structure Sonnet programs hierarchically, and to create libraries of often-used circuits.

Sonnet embodies event-flow, rather than data-flow, semantics. Whereas a data-flow model would require that *all* of a component's inputs receive data packets before it is allowed to execute, Sonnet's event-flow model permits component execution based on the arrival of data at *any* input. Likewise, a data-flow model normally implies that new data is generated at each output of a given component after its evaluation. In Sonnet, a component is free to decide at each execution whether to generate output packets for any, all or none of its outputs. This model is a more natural match to Sonnet's application to real-time environments, in which reacting to individual stimuli is a common theme. (An extension of this model to a more general one that allows arbitrary triggering conditions is readily achieved by replacing the so-called 'semantics module'.)

Sonnet models external and real-time stimuli as the asynchronous ('spontaneous') generation of data packets on a component's output. For example, a standard clock component keeps a timer that periodically causes the component to execute. More precisely, on each clock tick, the clock handler requests that Sonnet deliver a packet to a trigger input port on the associated component. This in turn causes the component to execute and generate a packet on its output port. User-interface components such as sliders and push-buttons use similar logic.

In order to handle external stimuli and components that generate packets on different output ports at each execution, Sonnet 'schedules' component executions. Each external stimulus triggers a cascade of component executions that flow through a Sonnet circuit in a *thread*. A thread is represented by a stack of pending data packet deliveries to trigger input ports. Sonnet is able to handle multiple threads traversing the same circuit in parallel, which arises, e.g. in the case of closely spaced MIDI notes propagating through an 'echo' circuit.

Sonnet is a strongly typed language. That is, all ports accept or produce a well-defined type of data packet. Sonnet normally disallows the connection of ports of incompatible types, to ensure type safety. At the same time, Sonnet supports parametrically polymorphic components. This addresses the problem of losing type information through, e.g. a generic delay component such as that shown in Figure 3. That is, once the delay

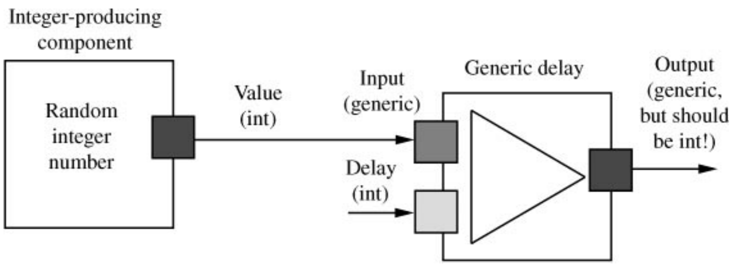

Figure 3. Lack of polymorphism causes type-unsafe circuits

```
HRESULT Adder::Execute(IsonnetComponentActivation pAct, ISonnetEvent pEvt)
{
      ISonnetIntPacket  pPkt1, pPkt2;
      pAct->GetInputValue(0, &pPkt1);
      pAct->GetInputValue(1, &pPkt2);
      pAct->SendOutput(0, pPkt1->Value() + pPkt2->Value());
      return S_OK;
};
```

Figure 4. An implementation of a simple component's Execute() method

element's input is wired to an output producing, say, integers, it should be illegal to connect the delay's output to an input expecting a string. Sonnet inductively propagates such type information through the circuit connectivity graph as connections are made, in order to maintain the circuit's type safety.

This feature also allows a greater degree of component re-use. For example, a single suite of vector-manipulation components is provided with Sonnet that can be used to operate on homogenous vectors based on any element type.

The Sonnet environment supplies a basic set of components for control, logic and mathematics, and for string, vector, list and matrix manipulation. A set of MIDI I/O components is also included, so that Sonnet programs may be written that produce and respond in real-time to MIDI events. Additional component sets include digital audio and GUI controls (pushbuttons, checkboxes, sliders and so on).

Sonnet's architecture and implementation is modular, a fact which manifests itself in several ways. Most notably, everything in Sonnet is represented by a software object: both data types and components are objects that support a specific set of pre-defined interfaces. No data or component types are treated specially by the Sonnet infrastructure (with the exception of chip I/O ports); all are provided by DLL's loaded at run-time. As a result, it is easy to add new components and packet types to the set available to Sonnet programmers. Most interfaces are supported by base classes, so that essentially all that a component writer needs to supply is an appropriate Execute() method. An example of this appears in Figure 4.

Sonnet allows components to be written in a variety of languages, including C, C++, Java and Visual Basic. Most of the standard computational components are implemented in C.

6. Sonnet + Imager

Creating and playing visual compositions requires control of four elements: visual objects, rhythm, interaction and overall compositional structure or sequencing. In

Sonnet + Imager, each of the elements in this sequence builds on the previous ones to provide larger-scale compositional control. We first give an overview of visual objects and of rhythm. Then, we give a somewhat more detailed synopsis of the Imager class library and associated Sonnet components for these two elements. Finally, we briefly describe the mechanisms used to support real-time interaction and compositional sequencing, which are built upon the previous two elements using ordinary Sonnet constructs.

7. Visual Objects: Color and Form

Describing visual objects requires languages for describing color and form. We have already discussed the reasoning that led to our use of the HSV color model. In addition to selecting a basic color model, it is necessary to provide constructs that make it easy to use color to achieve tensions, resolutions, harmonies and similar affects. Only then, can color be effectively used to connect musical and visual ideas.

One approach to controlling color is suggested in the work of color theorists. Ogden Rood, for example, argued that harmonious color combinations are found in pairs separated by 90° on the color wheel, as well as by triads of colors that are 120° apart [15]. Joseph Albers noted that strong complementary colors (those separated by 180°) produce after-images and vibrations [6]. Faber Birren observed that people find pleasure in harmonies of color based on analogy (adjacent hues) and on extreme contrast (complementary hues) [7]. When adjacent colors are used, the effect is to produce color schemes that are predominantly warm or cool in feeling. When complementary colors are used, the result is more startling and compelling.

For color to be played in real-time, it is useful to be able to move rapidly and easily from dissonant to harmonious, or from warm to cool, motifs. Sonnet + Imager facilitates manipulating the color of objects in just such a manner. For example, a line's and a ring's colors can be connected in a circuit so that a complementary relationship is maintained. With Sonnet + Imager it is possible to create a simple circuit so that as the ring's hue changes from red to yellow, the line's hue goes from blue to purple.

In addition to color, form must be represented in a graphic instrument. Some designers have based instruments on a single family of forms, such as kaleidoscopic imagery [8]. Others have provided a direct mapping of certain musical parameters onto visual ones. Sonovision, for example, related the frequency and size of an ellipse to the music's frequency and loudness, respectively [9]. Some systems, e.g. GEM [10], simply expose the forms provided by the underlying computer graphics toolkit, such as QuickDraw or OpenGL, leaving it to the composer to build higher-level forms. Our approach calls for building a vocabulary from the simple geometric elements of abstract art, an approach known as constructivist.

In the constructivist model, the abstract elements of mathematics are also the formal elements of abstract art. In his book *Point and Line to Plane*, for example, Kandinsky established such elements as points, lines and planes as more than tools to represent other objects. Instead, forms became the content of his art, and that of the modern artists who followed [11]. Kandinsky, Klee [12], Gerstner [13] and others have written extensively about how these simple elements can be combined and juxtaposed to represent such abstract ideas as causality, tension and growth. The modern artists' basic

```
interface OneArgFunctor {
  double Evaluate(double t);
};
```

Figure 5. A simple interface defining a class of functors

strategy of constructing works from simple elements is well suited to computer-based art.

In Sonnet + Imager interfaces are provided to critical parameters, such as the number of sides of polygons, or the eccentricity of conic sections. Functions and envelopes can be applied to these parameters. Mappings between these parameters and musical events are then defined at the time of composition or performance.

While the constructivist model is extremely powerful, Sonnet + Imager permits the addition of other aesthetic models. Other classes, such as string art, fractals and ambient forms can (and will) be added in modular fashion. This is thought to be one of the ways in which visual instruments can be personalized and customized. Artists can make the foundation elements of whatever aesthetic they wish to work with first-class elements in the environment. Their newly created types can inherit the color, movement and other methods that have been defined to apply to our basic classes of forms.

Sonnet + Imager's interactive, interpreted circuit structure allows for maintaining many kinds of relationships among visual forms. For example, relationships can be established over shape, thickness or texture. These interactions form one basis for compositional structure. As a result, our architecture provides support both for establishing and for controlling the structure and mood of visual compositions.

8. Functors: Navigating Parameter Spaces

Our objectives include creating a platform for devising a *variety* of interactive visual instruments. It is important, therefore, to have 'knobs' that are at once easy to understand and use and powerful. To aid in achieving this, we express all of Sonnet + Imager's rhythmic controls using a single construct: functors.

Functors are polymorphic software objects that encapsulate arbitrary parametric functions. Many functor clients need know only (1) its domain (the universe of valid inputs) and range (the space of possible outputs), and (2) how to invoke a method to request that it computes its output value given a domain value. In this manner, the specific function embedded in the functor (e.g. a linear curve, piecewise-linear envelope or sine wave) is decoupled from the clients that use it. (Naturally, clients who wish to manipulate the function a given functor computes use a distinct interface that exposes that kind of functor's particular attributes, as described below.)

More specifically, a class of functors is represented at the lowest level by a trivial interface with a single method to evaluate the underlying function. An example appears in Figure 5, roughly in C++ syntax. The functor shown embodies a 1-dimensional path, since it maps the real numbers onto the real numbers. The simplicity of the interface makes providing connectivity between, and substitution among, such functors straightforward.

A functor implementation simply needs to support the appropriate evaluation interface (such as OneArgFunctor above), along with the interfaces needed to permit

```
interface LinearFunctor: public OneArgFunctor {
public:
  void  SetOffset(double k0);
  void  SetSlope(double k1);
};
```

Figure 6. An interface that provides access to both a linear functor's parameters and its evaluation

manipulating the function's definition. An example of the latter sort of interface for a linear functor appears in Figure 6.

The decoupling of client from function has two consequences. First, functors over the same domain can be interchanged without affecting a compatible client. For example, given a linear functor mapping real numbers to real numbers (e.g. $f(t) = k_0 + k_1 t$), we can change the particular linear function in use by manipulating the parameters (k_0 and k_1) that define the function. Alternatively, a quadratic functor (such as $f(t) = k_0 + k_1 t + k_2 t^2$) can be substituted for the linear functor. In both cases, the client continues to work without modification.

Second, a given functor can be used with any compatible client. For example, a functor embodying a Fermat spiral over 2-D space can be used by any client that conforms to the 2D evaluation interface. Further, the client is free to interpret the function as navigating *any* pair of parameters, such as x and y-coordinates, hue and saturation or saturation and x-coordinate.

Functors can be constructed to encapsulate functions that compute non-numeric values, such as strings, discrete symbols, visual objects and so on. It is also possible to construct functors that embody functions of multiple arguments, such as $f(t_1, t_2) = k_0 t_1 + k_1 t_2^2$.

Sonnet + Imager provides a toolbox of simple one-parameter functors, including polynomial (linear, quadratic, etc.), transcendental (sine, cosine), as well as various two-parameter algebraic functors used for combining functors in intuitive ways.

Functors have many uses. They can be used to create classes of forms that go beyond the basic constructive elements described earlier. Interpolated forms are composites, created by interpolating two functors and then creating a trajectory form from the result. That is, given two 2-D functors f_1 and f_2, and a 1-D functor g, we construct a 2-D interpolation functor of the form $h(t) = f_1(t) + g(t)^*(f_2(t) - f_1(t))$. All functors are ordinarily (but not necessarily) normalized with respect to the common parameter t so that their spatial paths begin at $t = 0$, and end at $t = 1$. (E.g., the circular functor crosses the positive x-axis at $t = 0$ and 1.) In the simplest case, $g(t)$ is a constant, say 0.5, in which case the interpolation traces out a path halfway between the two paths represented by f_1 and f_2. Note that the interpolating functor $g(t)$ need not be constant, but rather, can vary with t, which greatly widens the universe of possible shapes.

The envelope functor is another flexible functor. It encapsulates a set of samples of a function. Its appeal derives in part from the fact that the function can be drawn graphically, without needing to know its precise mathematical form. It can be particularly effective in describing motion trajectories in 2-space: the artist simply draws the desired path.

Functors appear in three forms in Sonnet + Imager: as individual primitive components, as circuits or chips and as packets. The circuit metaphor employed by

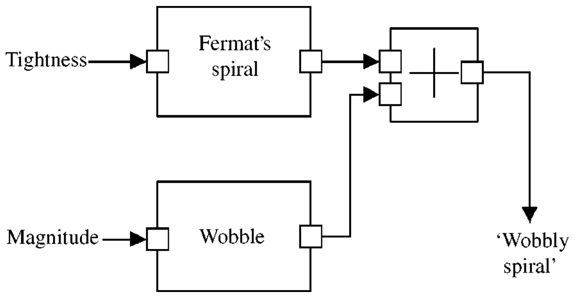

Figure 7. Building trajectories from simple components

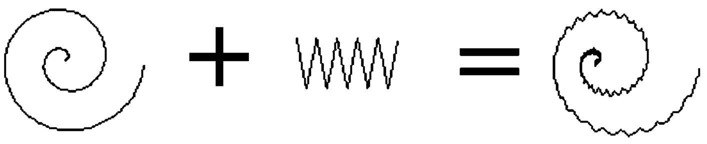

Figure 8. The result of adding two trajectories to produce a compound trajectory

Sonnet + Imager allows functors to be 'hooked up' to suitable clients using simple 'wires'. No code needs to be written, and type safety is guaranteed.

More elaborate functions can be composed from simpler ones by wiring functors together into circuits. For example, a 'wobbling spiral' path can be constructed by combining two primitive functors (a wobble functor and a spiral functor) with an adder composition functor, as illustrated in Figure 7.

The result of performing such a composition appears in Figure 8, in which a spiral functor was added to a cyclic 2-D linear functor (the 'wobble'). The packet produced at the adder's output is itself a functor, which is then fed to whatever down-stream component desires its result. This circuit could be collapsed into a chip and reused as a single component. Alternatively, the adder can be replaced by a multiplier, effecting a 'variable dilation' of the spiral.

Functors also become the focal point for visually manipulating rhythms (to be discussed in the next section) in both the spatial and temporal dimensions. In particular, we envision visual representations of functors that offer graphic controls (somewhat like spline control-points) that correspond to 'rooting' rhythmic events or synchronization points. Manipulating these control points in the spatial dimensions changes the parametric path, thereby affecting the nature of the visual rhythm. Likewise, moving the control points along the temporal dimension affects the timing of the visual rhythm. The net effect is that the artist can explicitly manifest decisions such as 'the hue should be blue at this rhythmic event'. Moreover, the decoupling of time from space allows the artist to easily alter each aspect independently; e.g. to move the said hue without losing the correlation to a significant musical event.

9. Rhythm: Linking Sound and Image Temporally

Given our previously identified goals, making time (and thereby rhythm) a first-class element in the language was a paramount consideration in the design of the architecture.

Because Sonnet + Imager is intended to facilitate the creation of moving graphic images that interact with musical performances, the temporal dimension plays an important role. It is our belief that rhythm is where music and graphics can most dramatically intersect and interact.

Rhythms can be produced through changes in any of the visual dimensions (e.g. color, shape, location and orientation). Considerations of rhythm pervade Sonnet + Imager's design. Manipulation of size, location and orientation are all ways in which objects become animated. Similarly, changes in the colors, textures and pen shapes can define rhythms. In short, dynamic changes in any attribute of a visual object contribute to the rhythms of the visual performance.

In designing a model of visual rhythm, we determined that we needed to address three aspects: *where, when* and *what.* By decomposing visual rhythm into these three aspects, we gained significant flexibility and created opportunities for reuse.

The first aspect of visual rhythm, *where*, refers to a path through N-dimensional space for any collection of N visual parameters. Any such path can form the basis of a visual rhythm. Paths need not be smooth or continuous. Rather, a path is represented by an arbitrary mapping of real numbers onto points in the N-space. This constitutes a very general and powerful view of rhythm. For example, the visual parameters hue, thickness and x-coordinate form one such three-dimensional space.

In practice, many interesting traversals of N-space can be implemented using a set of N distinct functions. For example, $\langle z_1, z_2, z_3\rangle >= f(t) = \langle kt, \ \sin(t), \ \cos(t)\rangle$ describes a path through 3-space in which the x-coordinate follows a linear path, while the y and z coordinates follow phase-inverted sinusoidal paths.

The second aspect of rhythm, 'when', specifies the dynamics of movement. Through an arbitrary function, encapsulated as a functor, variations in the rate at which an object moves along a path are specified. A temporal functor is used to map a time interval onto the parameter domain that will be used to drive a path function (to be described shortly). A trivial example linearly maps a time interval such as [5, 10] onto the parameter domain [0, 1]. A more interesting example maps the time interval [5, 10] onto [0,1] using the quadratic function $t = f(x) = -2.67 + 0.7x - 0.033x^2$. This function has the effect of dilating time so that it flows more quickly in the early portion of the time interval than in the latter portion.

Although the most obvious temporal mappings are monotonic and smooth, mappings need not be. For example, a sinusoidal temporal mapping defines cyclic motion along an arbitrary path. That is, the traversal moves back and forth along the entire path. The output of the temporal functor drives the path functor.

Finally, the 'what' aspect associates the abstract path with the concrete parameters to be affected. That is, the above 3-D path can be associated with hue, saturation and value, or alternatively with hue, thickness and x-coordinate. By decoupling the path's shape from the parameters that it affects, interesting paths can be applied to different sets of parameters without modification, other than scaling which can also be done in a modular fashion.

This micro-architecture provides considerable potential for reuse, an essential aspect of any toolbox. Given that the number and types of their dimensions match, any path through N-space can be substituted for another. A sinusoidal path in 2-space can be used to affect hue and saturation, or x- and y-coordinates and so on. A spiral can be substituted for the sinusoid. Likewise, one can vary the rate of traversal of

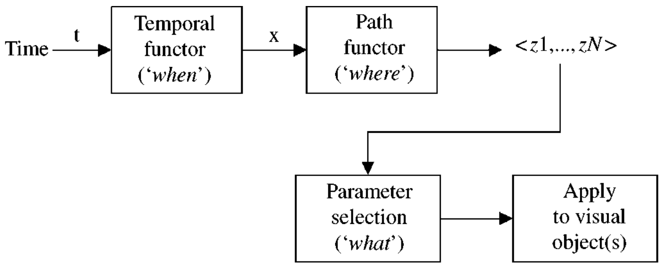
Time
t
Temporal functor ('when')
x
Path functor ('where')
<z1,...,zN>
Parameter selection ('what')
Apply to visual object(s)

Figure 9. A modular architecture for realizing rhythm

a path by substituting a different temporal functor. The architecture is illustrated in Figure 9.

10. The Imager Class Library

Facilities for creating and controlling visual objects and rhythms are embodied both as a set of C++ classes and as a set of encapsulating Sonnet components. The Imager class library contains classes for creating and manipulating 1-D functors, 2-D functors and color functors. It contains classes for defining visual forms, trajectories and for binding parameters to visual objects. The Imager class library is given at *http://RhythmicLight.com/imager.html.* Much of the library has been implemented and we grow increasingly confident that it will serve all four of our design objectives. Nonetheless, what is shown represents a snapshot in an evolving design.

11. Interaction: Input and Synchronization

Sonnet + Imager's improvisational performance capability relies on the ability to accomplish two things based on a performer's input: modulate visual parameters and trigger visual rhythms. A significant aspect of many integrated musical and visual compositions lies in the rhythmic interplay between the two domains. One key to establishing this interplay is the ability to synchronize musical and visual rhythms.

We achieve these two performance goals using Sonnet's innate abilities as the platform upon which to build and combine the necessary components. In particular, interface components can be constructed (e.g. for MIDI, data glove, dance suits) to allow data and event flows from external sources to modulate visual parameters. Because data packets can trigger component execution, these same interface components can act to trigger Sonnet + Imager visuals or to synchronize activity between Sonnet + Imager and external sources (e.g. of MIDI music content).

12. Composition: Orchestrating performances

Orchestrating visual and musical segments and moods is central to organizing a coherent performance from basic visual forms and rhythms. Satisfying this need generally

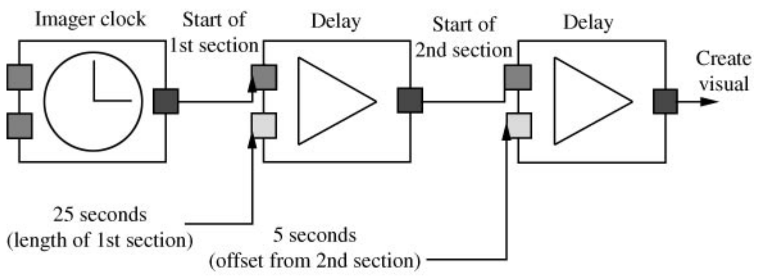

Figure 10. A simple sequencing circuit

requires real-time facilities. In particular, where visual rhythms are triggered sympathetically by musical events, real-time support is necessary for a timely response. Orchestrating segments of the visual performance, on the other hand, requires both high-level and fine-grained sequencing support. For example: 'show this visual five seconds into the second section of the piece'.

Sequencing at both of these levels is accomplished using Sonnet + Imager's event flow and real-time mechanisms to create and propagate events that trigger activity at the appropriate times. Specifically, Sonnet + Imager's clock and delay components and Sonnet's MIDI input components are used to create a stream of temporal events on the composition's time-base that is appropriate to the structure of the composition. Circuits built using these components create a cascade of events that mirrors the temporal structure of the composition.

The example of the above paragraph can be implemented by the circuit shown in Figure 10. The Imager clock component emits a packet when the composition begins playing. This packet triggers the delay component, whose output signifies the beginning of the second section. This in turn triggers another delay component, whose output is fed to a set of visual form constructor components, that make some set of visuals appear.

By simply substituting a MIDI input component (part of the standard Sonnet component suite) for the clock component, the circuit now causes visuals to appear when, e.g. a MIDI note-on event is received. Likewise, the MIDI input component can be used to receive data-carrying 'MIDI controller events', which in turn can be used to supply, say, the hue parameter of a color modulation sub-circuit.

13. Conclusion

We have described an architecture that integrates Sonnet and Imager, resulting in a system that is capable of generating a wide variety of dynamic visuals. Our architecture is based on strong aesthetic principles, which should make the system a powerful tool in the hands of artists and musicians.

In doing so, we have addressed each of the previously stated design objectives. First, we have created a simple yet powerful conceptual framework for rhythmic variation in terms of time, space and aspect. Relative to the second objective, this framework and the corresponding software architecture facilitate the creation of customized, intuitive and aesthetically useful controls over the entities in a visual composition. This is manifested in the progression from the design and organization of the visual primitives (shape, color and movement), to providing a composition mechanism for specifying and encapsulating visual objects with user-specified controls. We achieved the third objective by giving time an explicit role in our model of rhythmic variation. We achieved the fourth objective by extending Sonnet's set of components to include visual ones, and using functors both as primitive objects and as powerful building rules.

The value of this flexible and rich framework would clearly be augmented by a user-interface that is more imitative of painting and similar modes of interaction than the circuit metaphor. This is an important area for future research. However, we believe that, unlike competing systems, Sonnet + Imager's modular architecture provides a platform that is uniquely suited to achieving the long-standing objective of creating a graphic art that is like music in its impact on audiences.

References

1. M. S. Kushner (1990) *Morgan Russell.* Hudson Press, New York.
2. M. Graves (1951) *The Art of Color and Design.* McGraw-Hill, New York.
3. R. Russet & C. Starr (1988) *Experimental Animation: Origins of a New Art, 2nd edn.* DaCapo Press, New York.
4. D. Jameson (1996) Building real-time music tools visually with sonnet. *Second IEEE Real-Time Technology and Applications Symposium*, Boston MA.
5. O. Rood (1897) *Modern Chromatics With Application to Art and Industry.* D. Appleton and Company, New York.
6. J. Albers (1975) *Interaction of Color.* Yale University Press, New Haven, CT.
7. F. Birren (1987) *Principles of Color.* Schiffer Publishing Ltd, Atglen, PA.
8. S. Fels, K. Nishimoto & K. Mase (July–Sept 1998) MusiKalscope: a graphical musical instrument. *IEEE Multimedia*, pp. 26–35.
9. S. R. Wagler (1974) Sonovision: a visual display of sound. In *Kinetic Art: Theory and Practice* (F. J. Malina, ed.). Dover Publication, New York, pp. 162–164.
10. M. Danks (1996) The graphics environment for MAX. *International Computer Music Conference*, pp. 67–70.
11. W. Kandinsky (1994) *Point and Line to Plane*, 1926. In *Kandinsky*: *Complete Writings on Art* (K. C. Lindsay & P. Vergo, eds). DaCapo Press, New York, pp. 527–699.
12. P. Klee (1981) In: *The Thinking Eye* (Jurg Spiller, ed.). George Wittenborn, New York.
13. K. Gerstner (1986) *The Forms of Color*: *The Interaction of Visual Elements.* MIT Press, Cambridge, MA.
14. T. Wilfred (June 1947) Light and the artist. *Journal of Aesthetics and Art Criticism* **V**, 247–255.