In *1999 IEEE Symposium on Visual Languages*
Tokyo, Japan, pp. 111-118

Visual Music in a Visual Programming Language

Fred Collopy
Case Western Reserve University
flc2@po.cwru.edu

Robert M. Fuhrer
IBM Watson Research Center
rfuhrer@watson.ibm.com

David Jameson
DigiPortal, Inc.

Abstract

Sonnet was designed as a visual language for implementing real-time processes. Early design and development of behavioral components has largely focused on the domain of music programming. However, Sonnet's architecture is well suited to expressing many kinds of real-time activities. In particular, Sonnet is easily extended with new kinds of data types and behavioral components.

We have developed a collection of visual output components for Sonnet, referred to collectively as Sonnet+Imager. Its design embodies aesthetically grounded representations of color, form, and rhythm, as well as dynamics for each. Moreover, its value is enhanced by a flexible, modular architecture that treats these graphic entities and operations as first-class objects.

1. Introduction

There is growing interest in the design of instruments to play graphics in the way that musicians play sound. Such visual instruments could assume a wide variety of forms and functions. Indeed, there is no reason to expect less variety than there is among musical instruments. When the visual instruments are computer-based, we refer to them as imagers. The pieces that are produced using them we refer to as lumia.[1]

Interest in instruments that could integrate graphics with music is not new. In the early 18th century Louis-Bertrand Castel produced his *clavecin oculaire*, a light organ inspired by Newton's work on color theory. Since, there have been numerous experiments by painters, composers and film-makers. At the beginning of this century, the physicist Albert Michelson wrote: "so strongly do these color phenomena appeal to me that I venture to predict that in the not very distant future there may be a color art analogous to the art of sound--a color music [9]".

Around the same time, composers and filmmakers experimented with, and invented the elements of, such an art. Among the interesting developments were those of members of the Bauhaus, where Wassily Kandinsky and Paul Klee speculated on similarities between music and painting and Moholy-Nagy built kinetic sculptures to explore light in motion. In the United States, modern artists Morgan Russell and Stanton Macdonald-Wright built kinetic light machines, which they saw as a natural extension of their interest in color and movement. Film-makers including Oskar Fischinger, John and James Whitney, Mary Ellen Bute, and Harry Smith explored some of the same issues in abstract films.

Powerful graphic computers, MIDI, and high-bandwidth distribution media have led to an increased interest in computer-based graphic instruments. Scott Draves' Bomb, Mark Dank's GEM, Sydney Fels, et al.'s MusiKalscope, Sandy Cohen's Bindhu, and Greg Jalbert's Bliss Paint represent modern attempts to integrate graphics and music.[2] These and similar programs each implements some aesthetic model that characterizes it.

Any instrument imposes certain features of an aesthetic on its users. One cannot do with a trumpet precisely what one can with a piano. One cannot achieve in oils the effects that can be produced easily using watercolors. Thus, an instrument's design both limits expression and favors particular types of expression. Instrument designers must identify which choices are passed to the player. Choices provide the potential for expressiveness, though generally at the cost of added complexity. In other words, a balance must be achieved. When expressive potential is enhanced, and the resulting complexity is not too burdensome, the instrument succeeds.

With Imager we have designed an environment in which a variety of graphic instruments can be built. Details about particular aesthetics and range of expressiveness are left to a circuit designer. This decoupling of aesthetic choices from the underlying rendering engine can be used to hide some complexity from the player. Nevertheless, we cannot shirk responsibility for having made certain aesthetic choices.

The architecture described in this paper builds upon ideas embodied in a previous version of Imager that was constructed by the first author. That version runs under MAX on Macintosh, and is largely implemented as a single monolithic object. Nonetheless, it is capable of

[1] This term was suggested by Thomas Wilfred in 1947 ([13]).

[2] URLs for these and similar developments can be found at http://imagers.cwru.edu

producing a significant variety of visuals, as illustrated in several color images located at http://imagers.cwru.edu. One of these is reproduced in black and white as Figure 1. That version also serves as a baseline for the aesthetic quality, variety of form, and range of expressiveness of the new version.

However, its monolithic design has several shortcomings. First, it is not easily extended with new capabilities (e.g., new kinds of forms, motions, and other dynamic behavior). This problem stems in part from the fact that the design does not incorporate a modular architecture that encourages the development of reusable components. Second, several of its features force hard-coded constraints on the number and behavior of various kinds of objects.

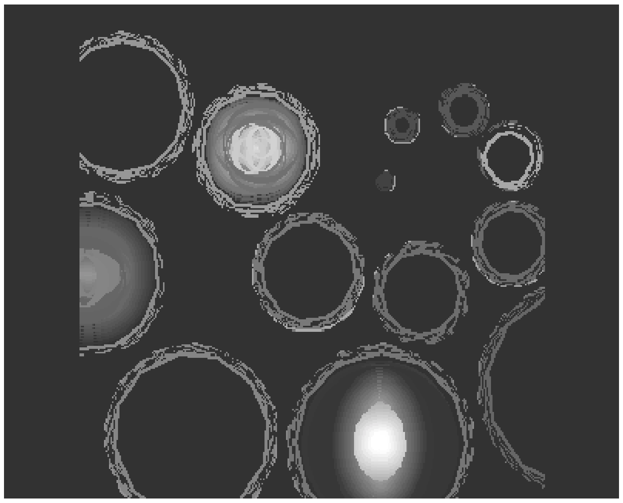
The version presently under construction represents an effort to overcome these limitations. In particular, our new design focuses on: 1) devising a more coherent, modular and robust architecture for building alternative instruments, and 2) providing the artist with a more useful environment by exposing key aesthetic properties in the form of intuitive and powerful controls.

The remainder of the paper is structured as follows. First, we discuss the critical role of visual aesthetics in devising an architecture for producing visual music. Then, we briefly introduce the Sonnet programming environment, upon which Imager is built. In Section 4 we provide a detailed description of our architecture. Some extensions that aid in realizing complex compositions are briefly presented in Section 5. Section 6 discusses how Sonnet+Imager differs from other dynamic visual generation systems. Finally, we briefly identify some future directions and make some concluding remarks.

2. The role of visual aesthetics in defining an architecture

Many approaches can be taken to devising an aesthetic for visual experience. One that has a close historical link with efforts to integrate visual and sonic experiences is the constructivist aesthetic employed by such modern artists as Kandinsky, Klee, Max Bill, and Karl Gerstner. For them and other modern artists, *form* and *color* assumed preeminent roles in expressiveness. Form was built up from simple elements, such as points, lines, planes, angles, and conic sections. Color was of interest in its own right, not merely as a way of rendering objects in the world.

Given that Imager is intended to deal with color and form in time, it is necessary to add a third element to describe changes in these two. In 1914, the painter Leopold Survage wrote about a new visual art in time, an art of "colored rhythm and of rhythmic color ([11], p. 36)." He believed that colored visual form could play a role analogous to that of sound in music and that these



forms could be described by three factors: color, the visual form proper, and rhythm. So, we add this third element to the constructivists' notions of color and form. Through rhythm, graphics and music can become linked, compositionally and improvisationally. These three elements (form, color and rhythm) completely describe the space of dynamic visuals.

For each of these three domains, color, form, and rhythm, it is necessary to make some choices about how composers and players will control them. In effect, one needs to decide what knobs will be available. In making these decisions, we have been guided by aesthetic considerations wherever possible. Our choice of the hue, saturation, and value (HSV) color model serves as an illustration.

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 for color theory. Indeed, the very language we use to denote colors is associated primarily with their hues. A hue is denoted 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-behaved color wheel, complementary colors would 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 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 choice of the right color model makes achieving certain aesthetic choices, such as surface textures, lighting effects, and perspective, much easier. We have chosen the hue, saturation, value (HSV) color model because those concepts and the related concepts of tint, tone, and shade have artistic meaning. We have taken a similar approach to the design of Imager's other components.

The design of Sonnet, however, is such that a choice of a particular model does not preclude the use of alternative models. For example, a lumianist could use components to transform to and from RGB or CYMK space.

3. The Sonnet Environment

With the design of Imager, we are interested in providing artists with the ability to define instruments that can be used to play with color and form as musicians play with sounds. Artists will wish to bring ideas and forms of their own to this process. Enabling that is achieved by embedding Imager's facilities in a programming environment.

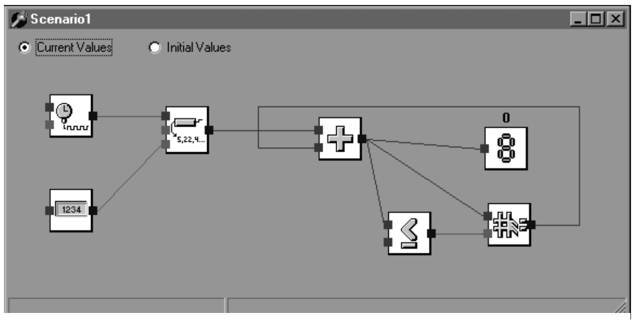
Sonnet was originally designed as a visual environment for associating runtime actions with running programs. It has since evolved into a visual programming language for the rapid development of real-time applications [6]. Sonnet uses a circuit metaphor, and embodies event-flow semantics. Sonnet behavior is expressed in two forms: as primitive "components", and as "circuits", which are interconnections of components. We often refer to Sonnet programs and Sonnet circuits interchangeably. The programming activity in Sonnet consists in 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. An input port may be designated a "trigger"; it then causes the `Execute()` method to be invoked whenever a data packet arrives on that input.

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 inputs or to have no outputs (as is often the case with interface components).

A circuit can be collapsed into a single component, known as a "chip", and used in the same manner as a primitive component. Chips allow the designer to structure Sonnet programs hierarchically.

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 present, each component's implementation is trusted to conform to its declared type signature.

Data packet types are arranged in a hierarchy. Thus, each port is compatible with its specified type and all of that type's ancestors in the hierarchy. At the root of the hierarchy is the "Generic" type, which is compatible with all types.

Sonnet is very modular, a fact which manifests in several ways. First, everything in Sonnet is represented by a software object. Data exists as quanta known as "packets". Components are also objects that support a certain set of pre-defined interfaces.

Second, the key language semantics are primarily defined by a replaceable module called the "semantic policy module". Thus, although the standard module implements event-flow semantics, it is a straightforward matter to replace this with one that implements pure data flow semantics.

Third, the scheduler is also a replaceable module. Currently, we have implemented simple round-robin scheduling. However, we recognize that specific real-time applications require other algorithms, such as rate-monotonic, earliest deadline first, best effort, and so forth. We have therefore designed in the appropriate hooks for other algorithms.

Finally, the Sonnet execution engine is defined and implemented as a server. As a result, the GUI may be replaced, or may run on a different machine. In fact, no GUI need exist; the engine can run "headless".

The Sonnet environment supplies a basic set of components for control, logic, and mathematics, and for string, 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 TCP/IP communication, file I/O, and GUI controls (pushbuttons, checkboxes, sliders, and so on).

Sonnet is easily extended by adding new component types and/or data packet types. No data types or component types are treated specially by the Sonnet infrastructure (with the exception of chip I/O ports).

Components are implemented as objects that support a small number of standard interfaces. Most interfaces are supported by base classes, so that essentially all that a component writer needs to supply is an appropriate `Execute()` method.

Likewise, data packets are implemented as objects that support a simple, standard interface. Beyond this, data packets typically support at least one additional interface to grant access to the particular kind of data that the packet holds.

4. Visual Compositions in Sonnet+Imager

Visual compositions in Sonnet+Imager have four major components: visual objects, rhythm, interaction, and orchestration. Each of these components builds on the previous one to provide larger-scale compositional elements. The components are treated in the four subsequent sections.

4.1. Visual objects: form and color

In the aesthetic model we have chosen, the abstract elements of mathematics are also the formal elements of 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 [7]. He, Klee [8], Gerstner [5], and others wrote 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 strategy of constructing works from simple elements is well-suited to computer-based art.

The choice of a model for defining and representing form is one of the most fundamental that must be faced in the design of a graphic instrument. Some designers have elected to base instruments on a single family of forms, such as kaleidoscopic imagery [4]. 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 [12]. Some systems simply expose the forms provided by the underlying computer graphics toolkit, such as QuickDraw or OpenGL, leaving it to the composer to build up higher-level forms. Our approach calls for building a vocabulary from the simple geometric elements of abstract art. Interfaces are provided to critical parameters, such as the number of sides for polygons, or the eccentricity for 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. Given Sonnet's structure, other aesthetics could replace and extend the ones we have provided.

Three foundation classes of forms that Imager provides are regular polygons, irregular polygons, and conic sections. 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.

We have already described the basic color model that Imager uses. Other issues arise in controlling color. In particular, it is necessary to provide high-level constructs that make it easy to use color to achieve tensions, resolutions, harmonies, and similar affects. Color can then be used to connect musical and visual rhythms.

Once we have a color model that corresponds well to the way the lumianist thinks about color, we can define interfaces to manage it. To do this effectively, we must identify appropriate manners in which the instrument permits the lumianist to control an item's color (beyond directly specifying its hue, saturation, and value).

One kind of control was suggested by the work of color theorists. Ogden Rood, for example, suggested that harmonious color combinations are found in pairs separated by 90 degrees on the color wheel, as well as by triads of colors that are 120 degrees apart [10]. Joseph Albers noted that strong complementary colors (those separated by 180 degrees) produce after-images and vibrations [1]. Faber Birren observed that people find pleasure in harmonies of color based on analogy (adjacent hues) and on extreme contrast (complementary hues) [2]. 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 improvisationally, 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 objects' colors in just such a manner. For example, a line's and a ring's colors can be controlled by a Sonnet circuit so as to maintain a complementary relationship. When the ring's hue is changed from red to yellow, the line's hue goes from blue to purple.

In fact, Sonnet+Imager allows for maintaining many kinds of relationships among visual forms. The circuit structure described above can also be applied to shape, thickness, and texture. These interactions form a basis for compositional structure. As a result, our architecture supports both establishing and controlling the structure and mood of visual compositions.

4.2. Rhythm: navigating parameter spaces

Because Imager is intended to facilitate the creation of dynamic graphic images that interact with musical performances, the temporal dimension plays an important role in its architecture.

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

Our model of visual rhythm has three components: *where*, *when* and *what*. By decomposing visual rhythm into these three components, we gain significant flexibility and opportunity for reuse.

The first component, w*here*, refers generically to a path through N-dimensional space for any collection of N visual parameters. Any path through such a space can form the basis of a visual rhythm. Paths need not be smooth or continuous in the mathematical sense. Rather, a path is represented by an arbitrary mapping from real numbers on the closed interval [0, 1] onto points in N-space.

This model constitutes a very general and powerful view of rhythm. For example, the visual parameters hue, thickness, and x-coordinate form one such 3-dimensional space. The individual dimensions need not be real-valued; strings and symbol sets also define valid dimensions.

In practice, many interesting traversals of N-space can be implemented using a set of N distinct functions. E.g.:

$$< z_1, z_2, z_3 >= f(t) = \left\langle kt, \sin(t), \cos(t) \right\rangle$$

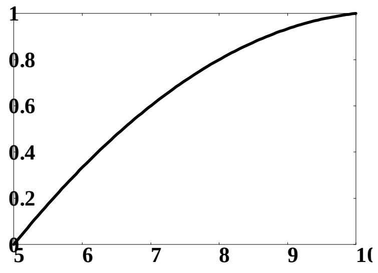
describes a path through 3-space in which the x coordinate follows a linear path, while the y and z coordinates follow phase-inverted sinusoidal paths.

The second component of rhythm, "when," associates temporal information with the path, thereby specifying dynamics. In particular, a distinct function maps a portion of the temporal dimension onto the parameter of the path-mapping function mentioned above. A trivial example linearly maps the time interval [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

$$t = f(x) = -2.67 + .7x - .0333x^2$$

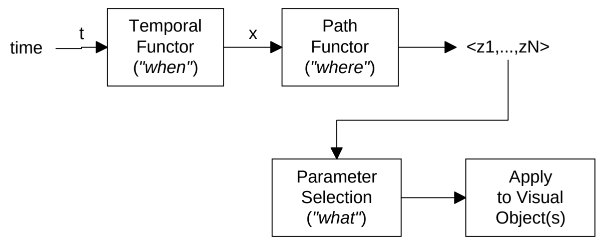
This has the effect of dilating time so that it flows more quickly in the early portion of the time interval than in the latter portion, as shown in Figure 3.

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.



Finally, the "what" component associates the dynamic 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 parameter, without modification.

This micro-architecture provides considerable potential for reuse, an essential aspect of any toolbox. Any particular path through N-space is trivially substituted for any other, given that the number and types of the dimensions match. Thus, a sinusoidal path in 2-space can be used to affect hue and saturation, or x- and y-coordinates, and so on. Likewise, one can vary the rate of traversal of a path by substituting another temporal component. The architecture is illustrated in Figure 4.


time
t
Temporal Functor ("when")
x
Path Functor ("where")
<z1,...,zN>
Parameter Selection ("what")
Apply to Visual Object(s)

Because our objective is to create a platform for devising a *variety* of dynamic and interactive visual instruments, it is important to have 'knobs' that are at once simple to understand and use, and powerful. We achieve this by expressing all of the above functions using the same construct -- functors.

4.2.1. Functors. Functors are software objects that encapsulate arbitrary functions. A functor's clients need know only 1) its domain and range, and 2) how to ask it to compute its value given a value in that domain. In this manner, the specific function embedded in the functor (say, a linear curve, piecewise-linear envelope, or sine wave) is hidden.

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.

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

Figure 5. A simple interface defining a class of functors

A particular functor simply needs to support the appropriate interface (such as `OneArgFunctor` above), and to support interfaces that provide access to manipulate the function's definition. An example of a linear functor that uses `OneArgFunctor` appears in Figure 6.

```
class LinearFunctor: 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

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 (in this case, 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 2D space can be used by any client that conforms to the evaluation interface. 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 $f(t_1, t_2) = k_0 t_1 + k_1 t^2$.

4.2.2. The Functor Toolbox. The Imager toolbox furnishes a set of simple functors, including one-parameter functors such as polynomial (linear, quadratic, etc.), transcendental (sine, cosine), as well as various algebraic two-parameter functors, used for combining functors in intuitive ways.

Perhaps the most flexible type of functor is the *envelope functor*. This type of functor encapsulates a set of samples of a function. Its appeal derives in part from the fact that the function can be drawn graphically (in "freehand"), 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.

4.2.3. Visual Representation of Functors in Sonnet+Imager. In the context of Sonnet's visual language, functors appear in three forms: as individual primitive components, as circuits or chips, and as packets.

This representation has two advantages. First, the circuit metaphor employed by Sonnet allows functors to be "hooked up" to suitable clients using simple "wires". No code needs to be written. Type safety is guaranteed by Sonnet's normal type-checking mechanism.

Second, more elaborate functors 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 Fermat's spiral functor) with an additive composition functor, as shown in Figure 7. The packet produced at the adder's output is itself a functor, which is called by an additional component at the appropriate times to produce its result.

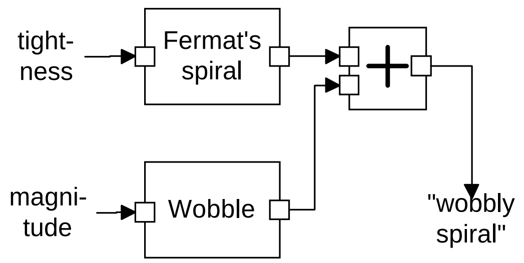


Figure 7. Compound trajectory built from simple components

This circuit can now be collapsed into a chip, and reused as a single component.

4.3. Interaction: input and synchronization

Imager's improvisational performance capability relies on the ability to accomplish two things based on a performer's input: 1) modulating visual parameters, and 2) triggering visual rhythms. Likewise, a significant aspect of any integrated musical and visual composition lies in the interplay between the rhythms in the two domains. The key to establishing this interplay is the ability to synchronize musical and visual rhythms.

Sonnet furnishes an ideal 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 flows from external sources to modulate visual parameters. Likewise, because data packets can trigger component execution, these same interface components act to trigger Imager visuals or to synchronize activity between Imager and external sources (e.g. of MIDI music content).

4.4. Orchestration

The orchestration of visual (and musical) segments and modalities is central to organizing a coherent, structured performance from basic visual forms and rhythms .

Satisfying these needs generally requires real-time facilities. In particular, where visual rhythms are triggered sympathetically by musical events, real-time support is necessary for timely response. Orchestrating segments of the visual performance, on the other hand, requires both high-level and fine-grained sequencing support (e.g., “show this visual five seconds into the second section of the piece”). Sequencing at both of these levels is accomplished using Sonnet’s event flow and real-time support to create and propagate events that trigger activity at the appropriate times.

5. Tools for Complex Compositions

Sonnet+Imager offers three additional tools for dealing with complex compositions. The first two of these aid the navigation of long compositions; the last allows artists to construct performances containing arbitrarily complex visuals, regardless of their computational complexity.

5.1. Navigating long compositions

Sonnet+Imager offers two techniques for navigating longer compositions: variable-speed fast-forward, and random access (seek). This is accomplished by triggering Imager’s display refresh from a *virtual clock*, whose rate can be controlled using a Sonnet+Imager component. Under ordinary circumstances (e.g. during performances), the virtual clock runs in real-time. During editing, the clock can be sped up (and the normal retrace interlocking bypassed) to effect fast-forward. If the rendering is already running at full speed, then the clock can be made to advance virtual time in larger increments.

The second feature, seek, is trivially implemented by setting the Imager virtual clock to the desired time (relative to the beginning of the performance).

Our current design for the seek feature poses a minor problem, however. Specifically, we have found that a significant class of visuals derives value from “artifacts” that are produced as moving visual objects interact in various ways, as shown in Figure 8. These artifacts are critically dependent on the history of pixel-level drawing operations used to render the visual objects. Thus, moving directly to a different temporal location elides the intermediate drawing operations, thus losing the artifacts. As a result, continuing the performance from that point will produce somewhat (or perhaps radically) different results from those achieved when starting from the beginning of the composition. As of this time, we have no solution to this problem.

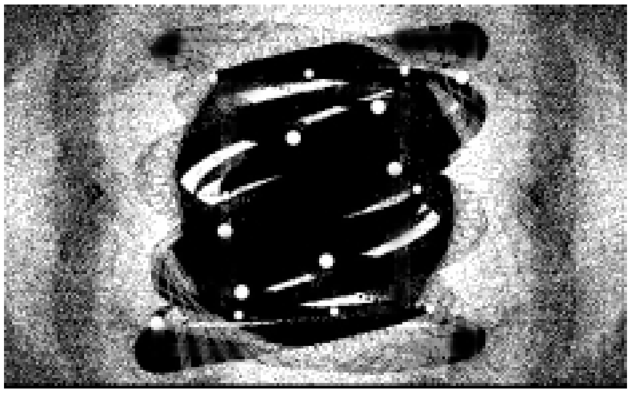


Figure 8. A visual exhibiting textural artifacts

5.2. Rendering computationally demanding visuals

We anticipate that artists will at times desire visuals whose computational complexity exceeds that which can be rendered in real-time with the available processing power. To cope with this eventuality, we provide a simple mechanism that allows Sonnet+Imager to render complex visuals in an off-line (non-real-time) mode, for later playback.

Specifically, the Imager virtual clock enters a mode in which it only advances time when the current frame is finished rendering, no matter how long that takes. Likewise, the Imager engine enters a mode in which it records each frame (once fully rendered), along with its virtual timestamp (typically a single frame’s worth of time). The resulting performance is a sequence of frames, suitable for compression by any of the various available techniques (e.g. QuickTime, MPEG). Note that no other portion of the architecture needs modification to support this mechanism.

With this mechanism in place, the artist gains access to a larger space of visuals, at the expense of the ability to see the performance in real-time. Obviously, this makes *developing* complex performances harder. To compensate for this, we propose that computationally expensive visual objects support a “fast rendering mode” that trades image quality for run-time overhead. More work is needed on this problem.

If interaction is designed into the performance, clearly only that portion of the performance that does not rely on interaction can be rendered off-line. Our system allows for this circumstance by rendering the static portion of the performance, and allowing for simultaneous playback of the “recorded” performance and arbitrary additional visual material in real-time.

6. Relationship to other efforts

Instruments for playing graphics with music can be grouped into two categories. The first consists of what might be termed image sequencers. These programs typically play back sequences of images that have been created elsewhere, sometimes providing transitional and other effects. X<>pose and Director are examples of such programs.

The second category is that of image generators. These programs permit the player to create some kind of graphic, sometimes specifically to accompany music. Bomb, GEM, MusiKalscope and BlissPaint are examples of this type of program.

Imager is designed to function in both categories. In this section we briefly consider how it differs from some of these other programs.

Bomb and MusiKalscope represent good illustrations of programs that implement a particular graphic algorithm or family of algorithms. Bomb’s algorithms, for example, are based primarily on ideas from artificial life research. MusiKalscope’s are based on kaleidoscopic imagery. The knobs that these programs provide to the player directly reflect the structure of their algorithms. Although these controls can be used to produce some exciting and useful effects, the models are not constructive, and thereby extensible, in the sense that Imager’s graphic model is. Moreover, because of the close association between the controls and the algorithmic structures, the controls do not map readily into the kinds of understandable aesthetic choices that artists are used to making.

BlissPaint provides a library of animated shapes and patterns, called scribblers, and permits you to determine where these shapes are drawn using distributors. A sequencer allows you to script the scribblers, and a color synthesizer can be used to control them in real-time. Though BlissPaint provides a substantial set of primitives, it does not provide a programming language with which to mix and arrange them. Where BlissPaint uses fairly static arrangements of coarse-grained primitives (that is, each encapsulates relatively large amounts of behavior), Imager provides more fine-grained primitive elements and a powerful programming environment that enables the artist to combine and control them.

GEM [3], like Imager, is embedded in a real-time, visual programming environment (Miller Puckette’s Pd). To first order, GEM’s architecture simply exposes the various primitives present in OpenGL. In contrast, Imager’s primitives embody models derived from the work of painters and graphic artists. Therefore, we believe it will be easier to produce non-trivial works of high artistic quality using Imager.

7. Conclusions

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

The value of this very 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 other competing systems, Sonnet+Imager’s modular architecture provides a platform that is uniquely suited to achieving this long-standing objective.

References

[1] Albers, J., *Interaction of Color*, New Haven: Yale University Press, 1975.

[2] Birren, F., *Principles of Color*, Atglen, PA: Schiffer Publishing Ltd., 1987.

[3] Danks, M. “The Graphics Environment for MAX,” *International Computer Music Conference*, 1996, 67-70.

[4] Fels, S., K. Nishimoto and K. Mase, “MusiKalscope: A Graphical Musical Instrument,” *IEEE Multimedia*, July-Sept 1998, 26-35.

[5] Gerstner, K., *The Forms of Color: The Interaction of Visual Elements*, Cambridge, MA: MIT Press, 1986.

[6] Jameson, D., “Building Real-Time Music Tools Visually with Sonnet,” *2nd IEEE Real-Time Technology and Applications Symposium*, Boston MA, 1996.

[7] Kandinsky, W., *Point and Line to Plane*, 1926 in K. C. Lindsay and P. Vergo [eds.], *Kandinsky: Complete Writings on Art*, New York: DaCapo Press, 1994, 527-699.

[8] Klee, P., *The Thinking Eye*, Jurg Spiller [ed.], New York: George Wittenborn, 1981.

[9] Michelson, A. A., *Light Waves and Their Uses*, 1899.

[10] Rood, O., *Modern Chromatics With Application to Art and Industry*, New York: D. Appleton and Company, 1897.

[11] Russet, R. and C. Starr, *Experimental Animation: Origins of a New Art (2nd edition)*, New York: DaCapo Press, 1988.

[12] Wagler, S. R., “Sonovision: A Visual Display of Sound,” in F. J. Malina [ed.], *Kinetic Art: Theory and practice*, New York: Dover Publication, 1974, 162-164.

[13] Wilfred, Thomas, “Light and the Artist,” in *Journal of Aesthetics and Art Criticism*, (V) June 1947, 247-255.