Affectations/Torso: A case study in future-proofing interactive computer music through robust code design

H. James Harkins
Modern Music Department, Xinghai Conservatory of Music, China
jamshark70 [at] dewdrop-world.net
http://www.dewdrop-world.net

This paper describes programming techniques used in my composition *Affectations/Torso* to organize large-scale form to facilitate future revision, including the complete replacement of its original performance interface with a gesture-driven, touchless interface. This article covers my approach to large-scale structure in interactive computer music before discussing code design errors, specifically *tight coupling* between components, that complicate interface changes. The issues are general to any programming environment for interactive sound art; examples are in the object-oriented programming language SuperCollider, and a final example in Pure Data shows how the same principles may apply in graphical patching environments.

Many interactive computer musicians use programming environments for music, such as SuperCollider, ChucK, Max/MSP and Pure Data. Some have prior programming experience, but it is more common to learn "on the job," picking up enough programming tricks to create interesting performances, but not necessarily enough to evaluate programming strategies and choose designs that are better able to absorb new demands in the future. Software design is not always necessary for a successful musical work; the larger the project, however, the more it benefits from "big-picture" code organization.

In my composition *Affectations*, I had created one of these forward-looking designs to handle one of the project's central requirements. Later, when I wanted to revise one movement into a solo work, it turned out that this design solved a number of problems essentially "for free." The design was so successful that I have already used it in subsequent work.

Affectations (2010-2011) is a 45-minute modern dance piece, with choreography by Laura Schandelmeier and Stephen Clapp of Dance Box Theater in Maryland, USA,[1] live visuals and motion detection by Lorne Covington,[2] and music by myself, written using SuperCollider Server software (McCartney 2002). *Torso* is the fourth of five movements. In 2013, I extracted this movement for separate performance and added a layer to control the performance by movement in front of the computer's webcam.

The original conception of *Affectations* used infrared cameras to track the dancers' movements and control the music's flow. The video-control plan was not fully realized; still, the musical code needed to be able to handle two control sources:

- Graphic User Interface (GUI) objects in SuperCollider itself, for testing;
- Open Sound Control messages from the video software, running in *vvvv*.[3]

To meet this requirement, I designed the original code to separate the musical sequence from the user-interface mechanisms. It is this separation that made it easy to add a new control mechanism after the fact.

This article describes this separated object design and illustrates how object-oriented design principles address problems that musician-programmers will face in larger projects sooner or later. It is not a programming tutorial; rather, it aims to expose musicians to some higher-level "programmers' secrets" that help to organize projects that are easier to maintain going forward, and even open up creative possibilities in the short term.

I will begin with some brief notes on object-oriented programming and on structuring interactive computer music in terms of multiple structural layers, and apply object-oriented design principles to the problem of incorporating user control into higher levels of musical structure. I will briefly cover the design of the video control components, and conclude with an example in Pure Data to demonstrate how similar design principles can work in graphical patching environments.

The complete code for *Affectations/Torso* is available for download from *https://github.com/jamshark70/affectations-torso*.[4]

Object-Oriented Programming concepts

SuperCollider is an *object-oriented programming* (OOP) language, and object-oriented design informs my solutions for large-scale form and external control. I do not assume readers will be deeply familiar with OOP, so I introduce the primary concepts here, as summarized in Table 1.

Term	Purpose
Class	Defines an object's *data structure* and *interface*.
Instance	Uses the class definition to store real data and do real work.
Interface	The list of an object's *methods*. In pure OOP, the public interface is the only way to use an object.
Method	Represents an action the object can perform. It takes *arguments* (input data) and returns a result.
Polymorphism	Allows the same method name to take different, sensible actions in different objects.
Inheritance	Allows a new class to be based on an existing one. The new class *inherits* data structure and methods from its superclass.
Design pattern	A way of structuring classes into replaceable, decoupled components that encourages code reuse and later extensibility.

Table 1. Key Object-Oriented Programming terms.

Phone	Direct connections needed	Unique connections	Hub connections needed
A	**A ⟷ B, A ⟷ C, A ⟷ D**	3	A ⟷ Hub
B	A ⟷ B, **B ⟷ C, B ⟷ D**	2	B ⟷ Hub
C	A ⟷ C, B ⟷ C, **C ⟷ D**	1	C ⟷ Hub
D	A ⟷ D, B ⟷ D, C ⟷ D	0	D ⟷ Hub
	Total	$\frac{4\cdot 3}{2} = 6$	4

Table 2. Connections required in a 4-telephone network, comparing direct connections against a hub. "Unique connections" does not count duplicates. A ⟷ B is counted in the A line, but not the B line.

OOP's principal innovation is to bundle a data structure and actions that can be performed on the data structure into a single package called an *object*. Every object belongs to a *class*; a class defines the variables representing the data structure and the *methods* that declare the actions. A new class can build on an existing class through *inheritance*, in which the new class has access to the entire data structure of its parent class, as well as the parent's set of methods (which it can use unchanged, or overwrite with a new definition specific to the new class).

This set of methods is called the object's *interface*, reflecting the object-oriented paradigm shift: focusing on *what an object can do* rather than *what kind of object it is*. This shift of focus allows one object to masquerade as another, by *polymorphism*. Two classes may have very different data structures, but implement some method names in ways that are compatible with each other. In SuperCollider, for instance, GUI objects change the displayed data using the `value_` method, but the visible impact is very different for buttons, number boxes, sliders, pop-up menus and so on.

Polymorphism makes it possible to build complex systems out of simpler, reusable components. Two objects with a common interface can do different work in response to the same orders, "plugging into" the same part of a larger system. For instance, in *Affectations/Torso*, each section needs to react differently to video data coming from the laptop's webcam. A straightforward implementation uses a different video-response object for every section. Each of these objects receives video data and triggers events in the same ways. Only the logic between input and output is different; that is, the objects communicate identically but do different work. Because of the identical communication, the first section can plug its video responder into the system and the program responds to one set of gestures. When that section ends, it unplugs the video responder; then the second section plugs in a different one, and the program now reacts to a different set of gestures.

Object design: *Tight* vs. *loose* coupling

Object design is the practice of deciding which objects handle which requirements and standardizing the interfaces by which these objects should cooperate. Object-oriented design draws an important distinction between *tight* and *loose* coupling.

Object coupling is an abstract concept; an analogy may clarify. One way to connect telephones in a wired network is to install physical lines between every pair of phones. This is *tight* coupling. It seems like the easiest way to go—what could be simpler than direct connections? However, it does not scale easily to larger networks. A network of only four telephones requires six direct connections (Table 2). For any number of telephones n, the number of connections is the sum of integers 1 to $n-1$: $\sum_{k=1}^{n-1} k$. This may be calculated more simply as $\frac{n(n-1)}{2}$: a quadratic calculation. Quadratic systems are especially inefficient. In a linear system, the cost of adding new components is roughly the same, no matter how many components there were to begin with. Quadratic growth means that the cost of one new component grows with the size of the system. Adding one telephone to a four-telephone network requires four new connections; adding one to a hundred-telephone network would need a hundred new connections, in addition

to the $\frac{100 \cdot 99}{2} = 4,950$ connections that already existed. Tight coupling in software, similarly, is a trivially simple design concept which becomes unmanageable in practice as the system grows.

The actual structure of telephone networks, with groups of endpoints feeding into hubs, reduces the number of connections dramatically. If four telephones connect to one hub, then you need only four connections. 100 phones need 100 connections (a considerable savings over the nearly 5000 direct connections that would be needed), and adding one new unit adds only one connection, whether the system is small or large. This is *loose* coupling, in which communication goes through an intermediary. The intermediary adds a small amount of complexity to the design structure—for the smallest networks, the structural overhead may not be worth it—but as the network grows, the structure vastly simplifies the shape of the network and makes the whole much easier to maintain.

In software, the nature of the communication between objects is at least as important as the number of connections. Two forms of tight coupling are direct references to objects and communicating specific actions to other objects.

Direct references to objects. *Problem:* The more direct references, the harder it is to keep them consistent with each other, especially if those connections are scattered among a large number of objects.

Solution: Keep references in *hubs*. With fewer places to handle references, the chance of mistakes is reduced. The triggering strategy in this paper uses exactly such a hub.

Communicating specific actions to other objects. *Problem:* It is not always possible to predict the specific responses required for one object's change of state. Coding toward specific actions locks the program into those specifics. Even very simple future requirements would require changes in the communication protocol.

Solution: Communicate *what happened* rather than *what to do*. If a synthesis parameter should be reflected in a graphical user interface (GUI), it should communicate only that its value changed. The GUI object would receive the notification and take its own appropriate action. Other objects are free to respond to in a different, equally appropriate, ways.

Decoupling is so important to the design of large programs that entire books are devoted to *design patterns*: object structures that promote reusability. The most notable of these, still influental after nearly twenty years, is *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma et al. 1995). The techniques presented here rely on two design patterns, related to the two forms of tight coupling listed above.

- A *Mediator* is a hub that connects any number of senders and receivers, without the senders having to be aware of the receivers and vice versa.
- An *Observer* registers with another object (called the "model" or "subject") to receive updates. The model broadcasts notifications to all registered observers. The messages to be broadcast are abstract, rather than specific commands.

The concrete benefits of these patterns will become apparent through the discussion of the implementation details of *Affectations/Torso*. This discussion focuses on triggering events, but the techniques apply generally to any communication between discrete components. For example, synchronizing parameter values with graphical displays and external control devices benefits from the *Model-View-Controller* architecture (Krasner / Pope 1988). A complete discussion of MVC is beyond this article's scope, but here is a brief summary:

- The *Model* represents the parameter's value. When it changes, it broadcasts a notification.
- The *View* is any reflection of that value that is perceptible to the user: a slider or knob on screen, audio synthesis, or display on an external device.
- The *Controller* is the Mediator that connects the Model to its physical View. It registers as an Observer of the Model, and manipulates the View in response to "changed" notifications. If the View is an interface object that the user can touch, the Controller sends the new information to the Model, whose notification pushes the new value to every other Controller.

It is slightly more complex initially to implement MVC rather than direct connections for parameter control, though it becomes easier with practice. It is worth the effort, however, as MVC makes it trivially easy to add new interface components. The Model is the hub previously mentioned. Other objects connect only to the hub, simplifying the channels of communication and reducing the likelihood of bugs.

Control over large-scale form

Musical compositions operate at multiple structural levels simultaneously; when the composition is expressed in code, it makes sense to organize these levels into distinct objects. Table 3 summarizes the levels at which I tend to work, beginning with the most detailed levels and proceeding to higher levels of organization.

For sonic events, I favor SuperCollider's built-in event and

Level	Responsible objects	Typical duration
Sonic events	Event	60ms - seconds
Gestures	Process; Pattern	Seconds
Sections	Section	Seconds, minutes
Whole composition	Section sequencer	Minutes, hours

Table 3. Hierarchical levels in my SuperCollider composition framework.

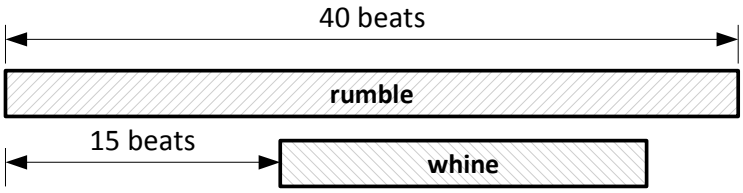

Figure 1. Graphical view of two overlapping processes.

```
TLSequenceIterator([
   bpCmd: (name: \rumble, dur: 40),
   15,
   bpCmd: (name: \whine, dur: 20)
]);
```

Listing 1. Simple sequencer, implementing the musical flow from Figure 1.

pattern framework (Kuivila 2011).[5] An Event object produces one sonic event, typically a note; Pattern objects generate sequences of Events. A pattern almost always depends on other resources: a unique audio bus for mixing at minimum, but possibly also including effect processors, buffers containing sampled audio or wavetables and the like. A musical *process* combines a pattern with all its resources into a single package (Harkins 2011). Playing a process produces one or more musical *gestures*; processes may naturally overlap to build complex textures. These objects represent the detail layers of musical structure.

In my solo performances, I manage higher levels of structure by controlling the behavior of process objects, by issuing text commands to the objects while onstage or using a MIDI controller. I could not be present for all performances of *Affectations*; thus, the software had to be able to do automatically what I would do by hand if I were onstage. For this purpose, I developed a timeline sequencer, published as an extension package called "ddwTimeline,"[6] which performs lists of *command* objects.[7]

For example, in a live performance, I might start a low rumbling sound, and add a high-pitched whine some seconds later (Figure 1). The sequencer expresses this simple case in a way that reflects the temporal layout, as shown in Listing 1. The first line inside the `TLSequenceIterator` is a `bpCmd`, or "Bound-process command," which plays the rumble process and stops it after 40 beats. After 15 beats, another `bpCmd` plays the whine for 20 beats. The commands are written sequentially but may overlap. The timing between them can be controlled by specific durations as here, calculated durations (which may introduce randomness), or by responding to some commands' state changes. The last of these supports user-trigger control, to be covered in the next section.

Above this, a *section* object can generate and play one or more sequences, and a list of sections constitutes the piece. If the sections are carefully written and correspond to rehearsal marks in a score, performers can start the electronics at any mark, facilitating rehearsal. Also, I find that the separation between processes, sequences and sections helps with the creative process. While working on higher-level form, I can focus on higher-level code that omits the details of musical gestures: a "zoomed-out" view. If a gesture needs some adjustment, I can drop down to the process level to zoom in on the details without being distracted by higher-level concerns.

Large-scale timing by triggers

In *Affectations* and other of my works, the durations of sections, and cues within sections, may vary according to the performers' actions. That is, the sequencer must run some processes and then wait for input before proceeding to the next command. The ddwTimeline framework uses *sync*

```
TLSequenceIterator([
   synthCmd: (name: \default, freq: 60.midicps, dur: 0.8),
   \cmdSync,
   synthCmd: (name: \default, freq: 64.midicps, dur: 0.8),
]).play;
```

Listing 2. Simple sync-keyword example in ddwTimeline.

```
(
~trigParms = (
   id: \button,
   setDoneSignal: {
      ~doneSignal = true;  // tells funcCmd to wait
      (inEnvir {
         var center = Rect.aboutPoint(Window.screenBounds.center, 50, 35);
         ~button = Button(nil, center)
            .states_([["GO"]])
            .action_(inEnvir { ~stop.() })
            .front;
      }).defer;
   },
   clearDoneSignal: {
      (inEnvir { ~button.close }).defer;
   }
);

t = TLSequenceIterator([
   synthCmd: (name: \default, id: \c, freq: 60.midicps),
   funcCmd: ~trigParms,
   \cmdSync,
   funcCmd: (func: { ~iterator.findActive(\c).stop }),  // stop previous note
   synthCmd: (name: \default, id: \e, freq: 64.midicps),
   funcCmd: ~trigParms,
   \cmdSync,
   funcCmd: (func: { ~iterator.findActive(\e).stop }),
   synthCmd: (name: \default, freq: 67.midicps, dur: 0.8),
]).play;
)
```

Listing 3. External trigger by a GUI button.

keywords for this. Sync keywords themselves do not incorporate any user action to resume the sequence. They simply pause the sequence until one or more previous commands finish:

- `\sync`: Pause the command sequence until all prior commands come to an end (note that it is the sequence of commands which pauses, while the commands themselves continue to operate);
- `\cmdSync`: Pause the sequence until the immediately prior command comes to an end, ignoring the status of other previous commands.

What does it mean to say that a command has come to an end? Figure 1 shows that commands occupy time: they begin, remain active for some time, and then stop. When a command stops, it communicates that fact back to the sequencer. The "stopped" notification from the command may clear the condition for which the sync keyword was waiting, allowing the sequence to continue. Listing 2 demonstrates. A `synthCmd` plays a single synthesis node on the server. The `\cmdSync` keyword takes effect immediately after the `synthCmd` starts, pausing the sequencer. The parameter `dur: 0.8` tells the command to stop itself after 0.8 beats. At that moment, `cmdSync`'s condition has been satisfied and the sequencer continues to the next `synthCmd`.

User-input triggers

Any command object can expose a user trigger by creating one or more objects that respond to user input. At

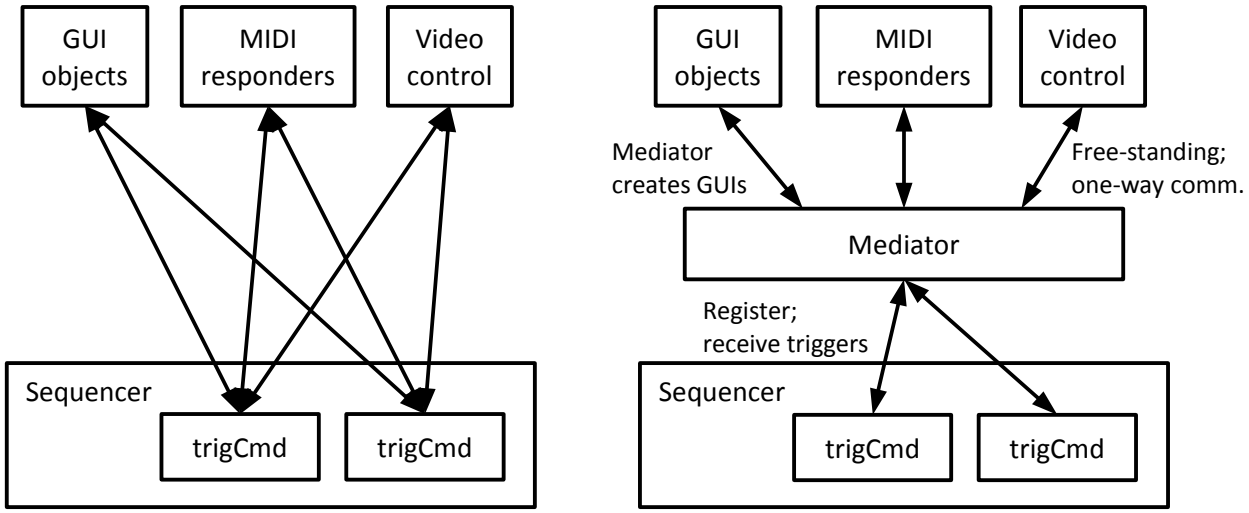

Figure 2. Schematic comparison of embedded vs. decoupled approaches.

a command's onset, a hook (`setDoneSignal`) allows the command to create additional resources; at the end, another hook (`clearDoneSignal`) releases them. The command should also set a variable, `~doneSignal`, so that the command will stay active until the user trigger arrives, or it runs out of work to do. An example may be found in Listing 3. This example installs a clickable `Button` as the user-input responder.[8] When clicked, the button runs an `action` function; here, the action stops the `funcCmd`: `~stop.()`. Once the `funcCmd` is finished cleaning up, `cmdSync` is free to allow the sequence to continue and play the second note.

In a composition, the commands between user triggers would be far more complex, controlling process objects instead of single notes. Also, the command may use any kind of object that responds to external information (typically MIDI or OSC receivers). The triggering principle remains the same in all cases.

Embedded triggers: Problem

The above design *embeds* the user interface objects into the trigger commands. Each trigger command must create and destroy all the user interface objects it needs. Figure 2a illustrates the tight coupling this structure implies. The riskiest element is that every user interface object must know the specific trigger command to stop. If this information is faulty, or an interface object was not removed properly (pointing to a stale command), unpredictable behavior ensues.

Listing 3 inadvertently makes it difficult to fire the trigger by any means other than the GUI button. The GUI button stops the enclosing `funcCmd`; if we can stop the same command in code, then any trigger source could do the same. The `funcCmd` is marked with an ID `\button`, and the sequencer can locate command objects by ID: `t.findActive(\button).stop`.

This approach is neither convenient nor elegant. It is odd to make the sequence *go* forward by telling something to *stop*. Surely it would be more logical to tell some object to *do a trigger* instead. In fact, this is a common problem in object-oriented programming: a programming interface (the set of methods required to accomplish a task) is designed for one job, but later pressed into service for a different job. The old programming interface is often inappropriate for the new task.

Solution: The *Mediator* design pattern

The typical object-oriented solution is to create a new class of object whose interface is designed specifically for the task at hand. For this situation, the new object should:

- Provide a central location toward which user-interface objects can direct triggers;
- Forward triggers to any number of objects that are expecting a trigger.

The *Mediator* design pattern stands between related objects, providing a standardized channel of communication

```
(
Proto {
   ~prep = {
      ~registered = IdentitySet.new;
      ~makeResponder.();
      Environment.current
   };
   ~freeCleanup = { ~clearResponder.() };
   ~makeResponder = {
      (inEnvir {
         ~button = Button(nil, Rect(Window.screenBounds.right - 100, 0, 100, 70))
            .states_([["GO"]])
            .enabled_(false)
            .action_(inEnvir { ~doTrigger.() })
            .front;
      }).defer;
   };
   ~clearResponder = {
      (inEnvir { ~button.close }).defer;
   };
   ~addClient = { |client|
      if(~registered.includes(client)) {
         "BP(%): Tried to re-register a previously-registered object"
         .format(~collIndex.asCompileString).warn;
      } {
         ~registered.add(client);
         ~addClientHook.();
      };
   };
   ~addClientHook = {
      (inEnvir { ~button.enabled = ~registered.isEmpty.not }).defer;
   };
   ~removeClient = { |client|
      if(~registered.includes(client)) {
         ~registered.remove(client);
         ~removeClientHook.();
      };
   };
   ~removeClientHook = { ~addClientHook.() };
   ~doTrigger = { |trigId|
      ~registered.do { |client|
         client.doTrigger(trigId);
      }
   };
} => PR(\trigMediator);
)
```

Listing 4. A simple trigger mediator.

```
(
PR(\funcCmd).clone {
   ~doneSignal = true;
   ~mediator = \trigMediator;
   ~setDoneSignal = {
      BP(~mediator).addClient(currentEnvironment);
   };
   ~clearDoneSignal = {
      BP(~mediator).removeClient(currentEnvironment);
   };
   ~doTrigger = { ~stop.() };
} => PR(\trigCmd);
)
```

Listing 5. A timeline-command to wait for a signal from the mediator.

```
(
// Create the mediator and MIDI controller once.
BP(\trigMediator).free; PR(\trigMediator) => BP(\trigMediator);

~pedalCtl.free;
~pedalCtl = MIDIFunc.cc(inEnvir { |val|
   if(val > 0) { BP(\trigMediator).doTrigger; };
}, 64);
)

(
TLSequenceIterator([
   { "one".postln; 0 },
   \trigCmd,
   \cmdSync,
   { "two".postln; 0 },
   \trigCmd,
   \cmdSync,
   { "three".postln; 0 },
   \trigCmd,
   \cmdSync,
   { "stop".postln; 0 }
]).play;
)
```

Listing 6. Using the mediator in a sequence, with additional MIDI control.

so that the objects on either side can come and go independently. Figure 2b illustrates the Mediator design pattern applied to to this situation.

Listing 4[9] defines a simple mediator for triggers. It addresses all the problems of embedded triggers noted above. Stale user interface objects are not a problem; in the example, the mediator creates one button that is reused for all triggers. The button does not have to know which commands are waiting for triggers. It simply calls the mediator's central triggering location: `doTrigger`.

As in Listing 3, a specific command object must appear in the sequence wherever the flow should pause for a trigger. A suitable command definition is found in Listing 5,[10] and its usage is demonstrated in Listing 6. The mediator will call `doTrigger` on it, so the command must implement this method name in a way that is appropriate, namely `~stop.()`. The command must also tell the mediator that it is interested in triggers, by registering with the trigger object, using `addClient` and unregistering itself after receiving the trigger, by `removeClient`. Where `setDoneSignal` and `removeDoneSignal` respectively created and removed a `Button` in Listing 3, now they should simply connect to the mediator.

Returning to the issue at the heart of this article: Does this structure simplify the addition of a new control mechanism? Indeed, it does: in Listing 6, it requires a mere four

lines of code to add a MIDI foot switch controller (see `pedalCtl`). Like the button, the pedal control needs only to call `doTrigger` on the mediator when the pedal is pressed (`val > 0` tests whether the pedal has been pressed or released). Both the button and the foot switch go through `doTrigger`; thus the response to the pedal is exactly the same as clicking the button with the mouse. There is no need to modify the sequence or the definitions of the mediator or trigger command; the sequence will run equally well with or without MIDI.

This design is why it was easy to add video control to *Affectations/Torso*. The video control logic was not simple, but connecting it to the trigger mediator was precisely as easy as in Listing 6's `MIDIFunc`. This eliminated a whole class of code-integration problems and allowed me to focus on the video-analysis problems.

Video control implementation

Space does not permit a complete description of the video-control system in *Affectations/Torso*. I will present an overview, however.

Flow of data

Figure 3 illustrates the data flow graphically. SuperCollider does not feature built-in video input, so I used GEM (Graphics Environment for Multimedia) objects in Pure Data to process webcam images. The Pure Data patch uses a simple frame-difference technique to identify areas of the frame that are in motion.[11] The patch then divides each image into a 3x3 matrix of sub-frames, and sends an OSC message to SuperCollider for each sub-frame, containing the *centroid* coordinates[12] and a relative measure of the amount of movement. One additional message gives the same statistics for the entire frame. The performance display is shown in Figure 4.

In SuperCollider, a `VideoListener` object receives the OSC messages and, in another use of the Observer design pattern, notifies any registered clients that a new set of analysis data are ready. In *Affectations/Torso*, the clients watch for specific triggering conditions and call into the trigger mediator to fire. As already shown, this clears any active sync keywords, allowing the sequence to advance.

Video trigger objects

The performance would be boring to watch if every video trigger followed the same gesture; so, each section needed its own unique analysis functions. Recall that objects are interchangeable if they have compatible interfaces; so, each section's video logic should be wrapped into objects with consistently named methods. In Listing 7, `prRespond` is the main hook; when the `VideoListener` broadcasts that a new frame's worth of data are ready, the video analyzer object calls `prRespond`, which then delegates to other functions (e.g. `nextSegCheck` and `respond`) to do the work. Replacing these lower-level functions with other logic will change the behavior of the video response, without changing the standard entry point.

It is simple, then, to have different analysis behaviors in each section by removing one analyzer object and adding another. In my compositional framework, a *section* object can create and remove any resources it needs at transition points. Listing 8 shows where the video analysis objects come and go: the first item in the sequence creates the analyzer; the `onStop` function, executed when the section ends, removes it. Thus, only one analyzer exists at any time, and the analysis logic can change without affecting any other components.

The analyzers use a variety of techniques to fire triggers: watching for the centroid to move to a specific part of the frame, comparing the amount of movement against a threshold, or even, in the final section, tracking the centroid to see when it moves from the right side of the frame to exit past the left edge.

Video data always have some jitter above and below a general trajectory. Tests for triggers should therefore cover several data points: one data point meeting a condition may be an aberration, while five or six consecutive matches are unlikely to be accidental. For example, some of *Torso*'s sections begin when one hand moves toward the top of the frame. If everything else is more-or-less still, the centroid will follow the moving hand, causing its y coordinate to decrease (centroid coordinates are normalized: $-1 \leq y \leq 1$). The top third of the frame is $y < -0.33$; however, y may jump into that range briefly even though the "true" centroid is closer to the frame's center. The analyzer should ignore such brief glitches. So, it specifies a number of consecutive "hits" (`segReqd`); when the y condition is met, the analyzer counts down from this number and only fires when it reaches 0. If the condition fails, the counter resets and there is no trigger.[13]

This type of logic figured into so many sections that I divided the test among several functions, allowing me to override parts of the test while reusing the overall logical form (Listing 7): `nextSegCheck` and `resetSegCheck` handle the counting, and `nextSegCondition` is the specific test based on the video data. In addition to checking the y coordinate, the `normmag` test in `nextSegCondition` ensures that there is enough movement in the frame to be meaningful. (Without enough frame movement, the centroid position is based on video noise rather than a solid reading, and is thereby unpredictable.) To match a different gesture, I need only to replace `nextSegCondition` with a different one, e.g. `~model.centroid.y > 0.33`

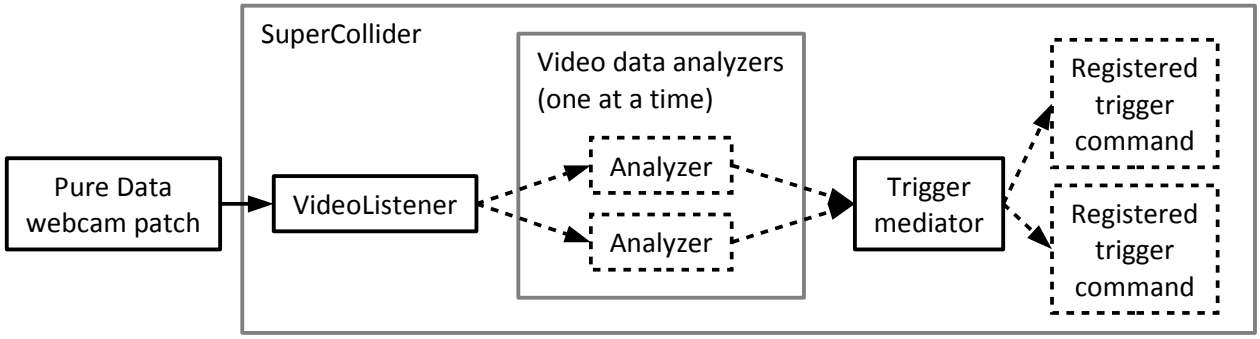

Figure 3. Flow of data in *Torso*'s video control. Dashed boxes indicate temporary objects.

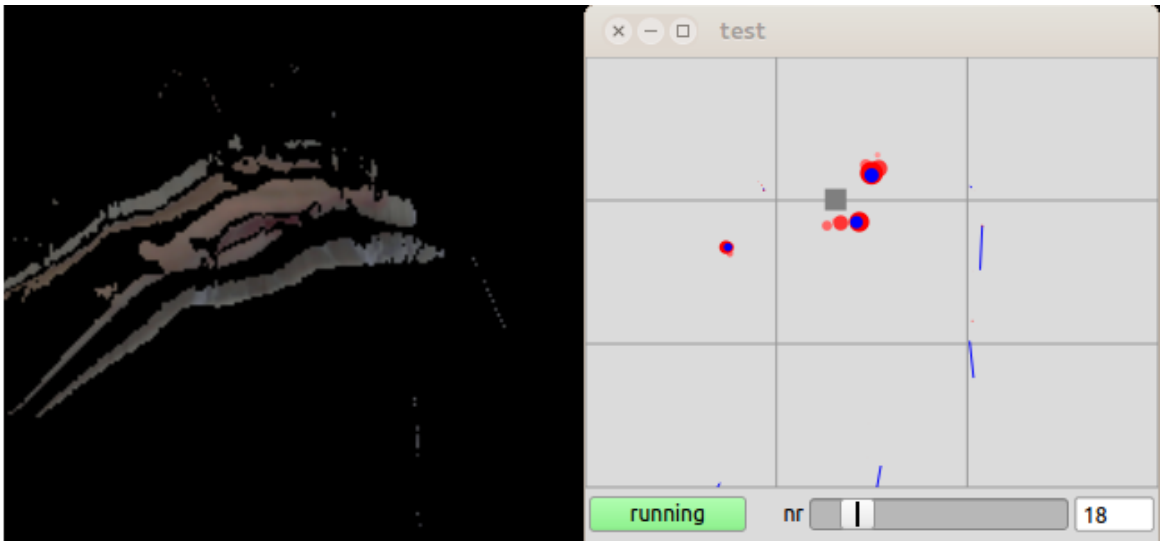

Figure 4. Display of video performance interface. To the left is the GEM window in Pure Data, showing the frame difference, where lighter pixels indicate more movement. At right is a representation of the video data received in SuperCollider. I am standing still and moving my hand in the middle of the frame. The gray rectangle, just above center in the right window, shows the centroid (center of movement).

```
Proto({
   // ... initialization methods omitted ...

   ~segReqd = 6;  // reqd for transition to next section
   ~segTrigCount = 0;
   ~segRange = [0.008, 0.25];
   ~nextSectionThresh = -0.33;
   ~segTrigCount = ~waitBeforeAdvance;

   // called when a new analysis frame is ready
   ~prRespond = { |obj, what, moreArgs|
      switch(~nextSegCheck.())  // perform the test: 3 outcomes
      { \goAhead } {            // Got a trigger: fire!
         BP(\segTrig).doTrigger;
      }
      { \respond } {            // Trigger failed; reset the test
         ~resetSegCheck.();
         ~respond.valueArray(obj, what, moreArgs);
      }
      { \notYet } {             // Trigger *might* complete, so don't reset
         ~respond.valueArray(obj, what, moreArgs);
      };
   };

   // If condition is true, count down; trigger when it reaches 0.
   // If a test fails, count will reset.
   ~nextSegCheck = { |obj, what, moreArgs|
      if(~nextSegCondition.()) {
         ~segTrigCount = ~segTrigCount - 1;
         if(~segTrigCount == 0) { \goAhead } { \notYet }
      } {
         \respond
      };
   };

   // You can override this to use a different test.
   ~nextSegCondition = {
      ~model.normmag.inclusivelyBetween(*~segRange)
         and: { ~model.centroid.y < ~nextSectionThresh }
   };
   ~resetSegCheck = { ~segTrigCount = ~segReqd };
}) => PR(\abstractVizTrig);
```

Listing 7. A trigger based on a threshold, using a counter to eliminate false positives.

to catch the centroid moving into the bottom third of the frame.

Continuous parameter control

In some sections of the piece, video data also control compositional or synthesis parameters, such as the relative speed of notes or a frequency modulation index. In Listing 7, prRespond calls an optional function labeled respond, which handles any arbitrary reaction to the video data, including parameter mapping. This function is called whenever video data come in but the section trigger should not fire. (If the trigger fires, then this analyzer object will be removed and any mapping within respond would be invalid; there is no need to call it in that case.)

Mapping video data to a parameter may be as simple as converting the incoming data from one range to another and setting the parameter. SuperCollider has a number of methods for range mapping: `linlin` to map a linear range onto another linear range, `linexp` for linear-to-exponential, and so on. Recall that the centroid coordinates range from -1 to 1; `linexp` can easily convert this to, e.g., a filter cutoff, as in `~model.centroid.y.linexp(-1, 1, 16000, 100)`. -1 is the top of the video frame and cor-

```
PR(\tlSection).copy.putAll((
   name: "T2030",
   // ... initialization and cleanup ...
   seqPattern: {
      Pn((
         sequence: [
            {
               // create this section's video analyzer:
               if(topEnvironment[\useVideo]) {
                  Fact(\t2030trig) => BP(\t2030trig);
               };

               // ... reset processes to this section's state ...

               0
            },
            // ... commands for this section ...
         ],
         // 'onStop' occurs when the section stops,
         // removing the video analyzer
         onStop: { BP(\t2030trig).free },
         // ... additional sync information ...
      ), 1)
   }
))
```

Listing 8. Outline of a section object, showing where video analyzer objects are created.

responds to the higher cutoff frequency of 16000 Hz. Then the `respond` function can set any type of control based on the new value. Because the `respond` function can hold any logic, mapping may be far more complex than this

Trigger dispatch in Pure Data

Graphical-patching environments, such as Max/MSP by Cycling '74 and the open-source project Pure Data, are widely used for interactive audio art. Their design, featuring graphical object boxes connected by virtual "patch cables" connecting outputs from one object to inputs of another, is radically different from that of object-oriented code; still, the principle of decoupling to support future revision holds true. The remainder of the article describes a patch in Pure Data that decouples user interface objects from musical flow. The patch translates easily to Max/MSP.

Three main concerns guided the design of the dispatcher in Pure Data:

- Reusability: It should be possible to use the trigger-dispatch mechanism in many compositions, with only minimal adjustments.
- Scalability: It should be able to handle 5 or 50 triggers, without cluttering the screen with too many patch cables.
- Unified programming interface: Just as Listing 4 allows triggers to be directed to a single point, `doTrigger`, the Pure Data mechanism should provide a single trigger location, addressable from anywhere in the patch.

For reuse, the triggering logic is encapsulated into an *abstraction* (graphical-patcher terminology for a patch saved on disk that may be embedded in another patch). The entire abstraction, shown in Figure 5, is analogous to the mediator in the SuperCollider examples. It presents a small interface with three control buttons and a display of the current section's name, using Pure Data's "Graph On Parent" feature.[14] It cannot assume that every composition will contain the same sections, so it provides an `[inlet]` at the top to receive a list of section names. A composition may send the list of sections to the inlet while loading the patch by triggering a message box using `[loadbang]`, as shown in Figure 6b. This also supports *scalability*, as the section list may be arbitrarily long.

Both the decoupling and the single location for triggers come from `[send]` and `[receive]` pairs. Normally, messages travel along visible "patch cables," while `[send]` and `[receive]` transmit messages "wirelessly"; `[send]` broadcasts a message to a specific name, and all `[receive]` objects with the same name get that message. This mechanism is inherently more loosely coupled than patch cables, because a new sender or receiver may be added anywhere

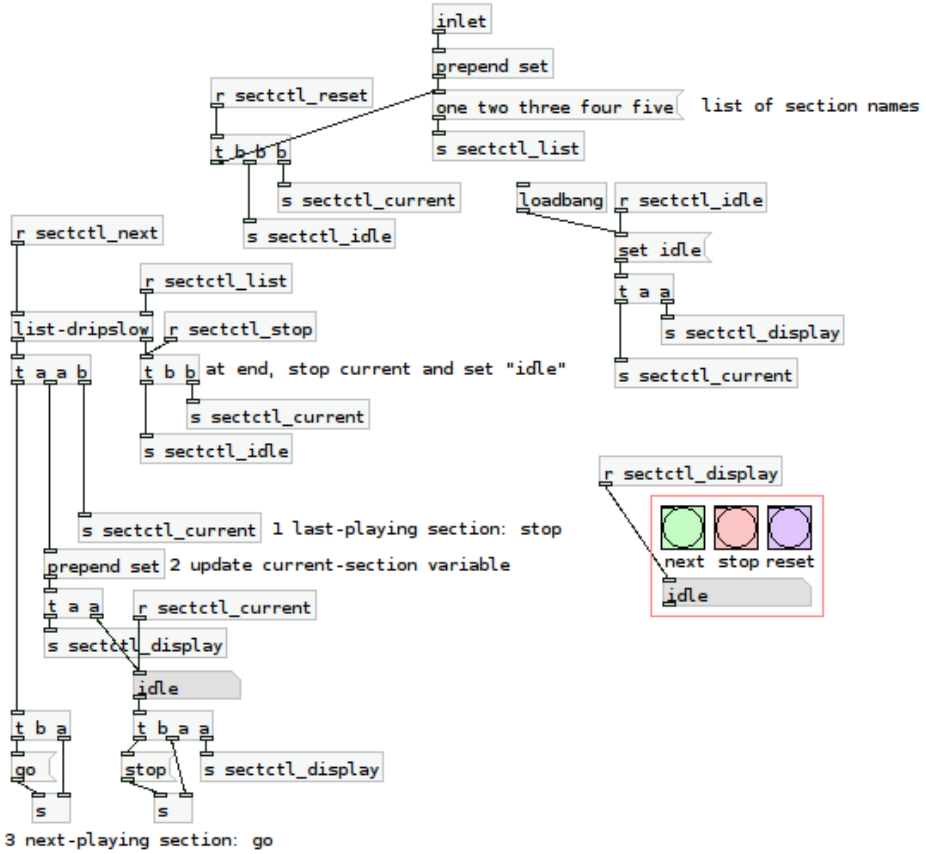

Figure 5. Dispatcher abstraction implemented in Pure Data.

a. Subpatch: Play one note.

b. Main patch: Complexity is hidden in the triggering abstraction (left) and subpatches.

Figure 6. Simple examples using the dispatcher abstraction.

in the patch (even in different abstractions or subpatches) with no need for an explicit connection.

The central triggering location is a receiver: [receive sectctl_next]. The "next" button sends a bang to this receiver, and any other control mechanism can do likewise. A simple way to add video control would be to add a subpatch, such as [pd video_control] in Figure 6b, which contains the video analysis logic and outputs a bang to feed to the sectctl_next receiver. Another type of control could be implemented in the same way, without modifying the triggering abstraction or any musical logic.

The triggering mechanism must also communicate with the piece's sections. The strategy from SuperCollider, in which triggering commands register with the mediator, does not apply in Pure Data because all [receive] objects are active all the time; there is no way to unregister them. Instead, a seldom-used feature of [send] allows it to change its name to a symbol received in its right-hand input.[15] Thereby, one [send] object can, at one time, direct messages toward a section called "one," and later toward a different section called "two." The list of section names should feed this right-hand input, one section at a time. Normally an entire list travels in a single message. A [list-dripslow] object parcels out the list items one by one, and receives its bang from the central triggering location.[16]

This example defines a minimal protocol, sending a go message to a section name when the section starts and stop when it should end. (The stop message is not optional. Rehearsals will often stop the piece in the middle, calling for a message distinct from "go to the next section.") The logic underneath [list-dripslow] ensures that the currently-playing section gets its stop message at the same time the next section receives go. If a composition requires a different messaging protocol, it may be implemented here. Sections may be created as subpatches, using [route go stop] to start or stop the section, as in the very simple example in Figure 6a. The section subpatches may be arbitrarily complex.

Depending on the needs of a specific composition, it may be necessary to adjust the logic. Still, the fundamental design separates triggering from musical logic and, just as in SuperCollider, exposes areas within the patch to add new mechanisms.

Conclusion

Programming for interactive compositions or sound art is similar to other kinds of programming in an important respect. Structures that are too simple (e.g. too many direct connections) work well for very simple behaviors. Moving toward more complex behaviors quickly reveals limitations, in particular, the risk that one change may force several other changes, which may call for further changes *ad infinitum*. Foresight in the early stages of creation helps to bypass these limitations.

In this regard, digital artists are at a disadvantage compared to conventional software developers. Traditional, top-down models of software development begin with "gathering requirements" before drawing up specifications for program behavior—all before writing any code. Careful specifications make it easier to plan object designs to meet not only current requirements but also allow room to handle future needs.

Digital artists are more likely to *discover* the requirements by experimenting with incomplete code. The work of art is the system's behavior, and we do not know how the system should behave until we interact with rudimentary prototypes. (In fact, my own creative process begins with poorly-structured code, but I move quickly to package interesting behaviors into process objects. These processes allow continued experimentation and at the same time support object-oriented modeling, making it easy to introduce loose-coupling strategies as soon as they are needed.) At this extremely simple stage, the benefits of loose coupling are not apparent, and the cost of creating Observers, Mediators and other extensible components can appear to be a waste of time. This article has demonstrated, on the other hand, how an open-ended object design for external event triggers facilitated a new version of *Affectations/Torso*, carrying the work forward into the future in a way that would have been considerably more difficult had the components been tightly coupled.

Restrictive code designs may impose a hidden cost during initial creation as well. As noted, the creative process begins with simple prototypes of behaviors that will grow in complexity before arriving at a compelling result. Design shortcuts (tight coupling, direct connections) are tempting, but they concretely impede the "free play" of ideas that that is essential in creative work. Whenever one abandons a musical impulse because it would be too difficult to adapt the simple prototype to it, a creative choice has been lost. If musician-programmers have extensible design templates in mind, ready to apply at a moment's notice, the distance between inspiration and implementation decreases and more interesting musical behaviors come within reach. Instrumentalists practice scales so that they can think in terms of groups of notes while sight-reading or improvising. Similarly, computer musicians can "practice" decoupling and come to think in terms of constellations of loosely coupled objects. Ideally, computer musicians would become so comfortable with better programming practices that they reach for extensible code structures as a matter of habit, rather than effort, expanding the available range of musical possibilities.

References

Gamma, E. / Helm, R. / Johnson, R. / Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, Mass.: Addison-Wesley.

Harkins, H. James. (2009). A Practical Guide to Patterns. Retrieved from *http://doc.sccode .org/Tutorials/A-Practical-Guide/PG_01_Introduction .html*

Harkins, H. James. (2011). Composition for Live Performance with dewdrop_lib and chucklib. In: Wilson, S. / Cottle, D. / Collins N. [eds.] *The SuperCollider Book* pp. 589–612. Cambridge, Mass.: MIT Press.

Krasner, G. / Pope, S. (1988). A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming* 1/3: 26–49.

Kuivila, Ronald. (2011). Events and Patterns. In: Wilson, S. / Cottle, D. / Collins N. [eds.] *The SuperCollider Book* pp. 179–205. Cambridge, Mass.: MIT Press.

McCartney, James. (2002). Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal* 26: 61–68.

---

[1] Dance Box Theater: *http://www.danceboxtheater.org.*

[2] Lorne Covington: *http://noirflux.com*.

[3] More information about *vvvv* may be found at *http://vvvv.org*.

[4] The site is a *git repository* containing the entire history of the revision process. Readers unfamiliar with *git* may download only the latest revision using the "Download ZIP" button on this page.

[5] Events and Patterns were designed by James McCartney, and come to SuperCollider Server from SuperCollider 2, the previous major version. In addition to Ronald Kuivila's excellent overview in *The SuperCollider Book* (2011), I have written extensive documentation on this framework, which is included in SuperCollider's documentation (Harkins 2009).

[6] The extensions described in this article are part of SuperCollider's Quarks package system, described in the "Using Quarks" help file in the SuperCollider distribution: *http://doc.sccode.org/Guides/UsingQuarks.html*.

[7] The actions in a sequence follow the *Command* design pattern. A Command represents an action in progress (Gamma et al. 1995). By making the action into an object, the program can store actions and pass them around to different parts of the system. In my framework, for instance, a sequencer can come to an end while one or more commands are still active. Those commands will be passed back to the object that started the sequencer. Then, this parent object can decide what to do with them: stop them, or pass them into a new sequencer.

[8] Listing 3 includes the potentially confusing construction `(inEnvir { ... do something ... }).defer`. This is an implementation detail that does not affect the code structure. SuperCollider requires any GUI manipulation to be deferred onto a lower-priority scheduler, so that GUI operations will not interfere with musical timing. Commands in my framework are encapsulated in SuperCollider `Environments`. The command's environment is in force at the time of running `setDoneSignal`, but the current environment switches back to the previously active environment before the deferred function wakes up. To be sure that the command's environment will be active inside the deferred function, `inEnvir` connects the function to the environment.

[9] The control mechanism described in this paper uses an alternate style of OOP called *prototype-based programming*. I needed to customize certain objects' behavior more than conventional OOP classes permit. Object prototypes support the major features of OOP (inheritance, polymorphism) and also allow radical changes in the structure of an object at runtime: adding new data variables and methods, and even rewriting methods. I have described my object-prototyping framework, and its syntax differences from standard SuperCollider class definitions, in an earlier article (Harkins 2011).

[10] The command definition is very short because of another OOP feature, *inheritance*. My prototype-based programming framework handles inheritance using the `clone` method. `PR(\funcCmd).clone { ... }` makes a copy of the original `funcCmd` object prototype and then modifies it according to the clone function, which adds two variables and three method definitions to the variables and methods inherited from `funcCmd`. The inherited code handles playing and stopping, and is in turn inherited from `PR(\abstractTLCommand)`.

[11] The *frame-difference technique* is based on the fact that a moving object in the frame will cause individual pixels (which are stationary) to change color. A change in color may be measured by subtracting a pixel's color in one frame from its color in the previous frame. Where there is no motion, the difference will be small and the pixel will go dark; thus, lighter image areas show where movement is taking place. GEM's `[pix_movement]` object implements this.

[12] A centroid is, generally, a center of mass. Here, considering brighter areas of the image to be "heavier" in terms of amount of motion, the centroid is the location in the sub-frame where the motion is centered. If most of the movement is in the bottom right corner of the sub-frame, the centroid will have large x and y coordinates.

[13] Note, in `prRespond`, that the trigger condition has three possible outcomes: `goAhead` to fire the trigger, `respond` when the test fails, and `notYet` when the test was successful but the countdown is still in progress.

[14] The corresponding Max/MSP object is `[bpatcher]`.

[15] The corresponding Max/MSP object is `[forward]`.

[16] Max/MSP's `[coll]` object can be used for the same purpose.

[Abstract in Korean | 국문 요약]

꾸밈/몸통: 강건 코드 설계robust code design를 통한 미래 지향형 상호작용 컴퓨터 음악에 대한 사례 연구

에이치 제임스 하킨즈

본 연구는 필자의 컴퓨터 음악 작품《꾸밈/몸통Affectations/Torso》에 사용된 프로그래밍 기법에 대한 것으로, 대규모 형태를 향후 수정이 용이하게 하기 위해 조직화하는 기술로써 원곡의 인터페이스를 무접촉-동작감지 인터페이스로 완전히 대체하는 것을 포함한다. 본 연구에서는 상호작용 컴퓨터 음악에서 대규모 구조에 대한 접근 방식에 대해 밝히고, 인터페이스 교체를 복잡하게 만드는 코드 설계 에러, 특히 구성 요소들 간의 밀착 결합에 대해 논의한다. 이 연구의 주제는 상호작용 소리 예술을 위한 프로그래밍 환경에서 매우 일반적으로 다루어지는 주제인데, 객체지향 프로그램 언어인 슈퍼콜라이더SuperCollider를 사용하여 그 예를 들고 있으며, 퓨어데이터Pure Data를 통한 마지막 예시에서는 그래픽 패칭patching 환경에서 동일한 원리들이 적용될 수 있다는 가능성을 보여 주고 있다.