

Slideshow: Functional Presentations

ROBERT BRUCE FINDLER

University of Chicago

(*e-mail*: robb@cs.uchicago.edu)

MATTHEW FLATT

University of Utah

(*e-mail*: mflatt@cs.utah.edu)

Abstract

Among systems for creating slide presentations, the dominant ones offer essentially no abstraction capability. Slideshow represents our effort over the last several years to build an abstraction-friendly slide system with PLT Scheme. We show how functional programming is well suited to the task of slide creation, we report on the programming constructs that we have developed for slides, and we describe our solutions to practical problems in rendering slides. We also describe experimental extensions to DrScheme that support a mixture of programmatic and WYSIWYG slide creation.

1 Abstraction-Friendly Applications

Strand a computer scientist at an airport, and the poor soul would probably survive for days with only a network-connected computer and five applications: an e-mail client, a web browser, a general-purpose text editor, a typesetting system, and a slide-presentation application. More specifically, while most any mail client or browser would satisfy the stranded scientist, probably only Emacs or vi would do for editing, L^AT_EX for typesetting, and Microsoft PowerPoint™ for preparing slides.

The typical business traveler would more likely insist on Microsoft Word™ for both text editing and typesetting. Computer scientists may prefer Emacs and L^AT_EX because text editing has little to do with typesetting, and these different tasks are best handled by different, specialized applications. More importantly, though, tools such as Emacs, vi, and L^AT_EX are programmable. Through the power of programming abstractions, a skilled user of these tools becomes even more efficient and effective.

Shockingly, many computer scientists give up the power of abstraction when faced with the task of preparing slides for a talk. PowerPoint is famously easy to learn and use, it produces results that are aesthetically pleasing to most audience members, and it enables users to produce generic slides in minutes. Like most GUI-/WYSIWYG-oriented applications, however, PowerPoint does not lend itself easily to extension and abstraction. PowerPoint provides certain pre-defined parameters—the background, the default font and color, etc.—but no ability to create new abstractions.

Among those who refuse to work without abstraction, many retreat to a web browser (because HTML is easy to generate programmatically) or the various extension of T_EX

(plus a DVI/PostScript/PDF viewer). Usually, the results are not as aesthetically pleasing as PowerPoint slides, and not as finely tuned to the problems of projecting images onto a screen. Moreover, novice users of \TeX -based systems tend to produce slides with far too many words and far too few pictures, due to the text bias of their tool. Meanwhile, as a programming language, \TeX leaves much to be desired.

Slideshow, a part of the PLT Scheme application suite (PLT, 2005), fills the gap left by abstraction-poor slide presentation systems. Foremost, Slideshow is an embedded DSL for picture generation, but it also provides direct support for step-wise animation, bullet-style text, slide navigation, image scaling (to fit different display and projector types), cross-platform consistency (Windows, Mac OS, and Unix/X), and PostScript output.

Functional programming naturally supports the definition of picture combinators, and it enables slide creators to define new abstractions that meet their specific needs. Even better, the Slideshow programming language supports a *design recipe* to help slide creators build and maintain animation sequences. Our design recipe guides the programmer from a storyboard sketch to an organized implementation, and it also suggests how changes in the sketch translate into changes in the implementation.

Even with the best of programming languages, some slide sequences can benefit from a dose of WYSIWYG construction. WYSIWYG tools should be part of the slide language’s programming environment—analogue to GUI builders for desktop applications. We have implemented extensions of the DrScheme programming environment (Findler *et al.*, 2002) that support interactive slide construction in combination with language-based abstraction.

In this paper—a revised and expanded version of our ICFP paper (Findler & Flatt, 2004)—we report on the Slideshow presentation system. Section 2 is a tour of Slideshow’s most useful constructs. Section 3 presents the Slideshow design recipe for picture sequences. Section 4 describes Slideshow’s implementation and addresses practical issues in rendering slides on different media and operating systems. Section 5 describes our prototype extension of DrScheme. Section 6 discusses some of our design decisions and compares to related work.

2 A Tour of Slideshow

A *pict* is the basic building block for pictures in Slideshow. Each *pict* consists of three parts:

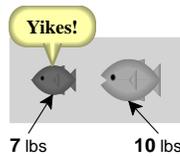
- A drawing procedure, which takes a primitive PLT Scheme drawing context and a location to draw the *pict*.
- A bounding box. A *pict*’s bounding box does not necessarily enclose all of the *pict*’s image, but usually it does. Besides a width and height, the bounding box includes upper and lower text baselines. When a *pict* contains a single line of text, the baselines typically coincide. When a *pict* contains no text, the baselines typically match the bottom edge of the bounding box.
- The identity and location of sub-*picts* (if any) that were used to compose the *pict*.

Slideshow includes a pre-defined set of *pict* constructors whose drawing procedures are implemented directly, and most other *picts* are created by combining them. Slideshow’s

pre-defined constructors include lines, boxes, circles, text, arrows, clouds, balloons, fish, and bitmap images in a variety of formats (including PNG, GIF, and JPEG).

A PLT Scheme drawing context produces output to either the screen, a bitmap, a printer, or a PostScript file. For this paper, we render example picts to embedded PostScript, and we scale the drawing context so that one drawing unit equals one point ($= 1/72$ of an inch) when printed. For a presentation, one drawing unit equals one projector pixel on a standard-size projector (but we discuss this more in section 4.2).

The remainder of this section demonstrates how to use Slideshow functions to generate images like the following, which might appear in a presentation about how fish gain weight when they eat other fish.



2.1 Generating Picts

The pre-defined pict constructor `filled-rectangle` takes a height and a width and produces a pict:

```
(filled-rectangle 20 10) 
```

Similarly, the `text` function takes a string, a font class, and a font size, and produces a pict:

```
(text "10 lbs" '(bold . swiss) 9) 10 lbs
```

The `standard-fish` function takes a height, width, direction symbol, color string, eye-color string, and boolean to indicate whether the fish's mouth is open:

```
(standard-fish 30 20 'left "darkgray" "black" #t) 
```

The `standard-fish` function's many arguments make it flexible. If we need many fish that differ only in color and in mouth position, we can define our own `fish` function that accepts only two arguments:

```
;; fish : str[color] bool -> pict
(define (fish color open?)
  (standard-fish 30 20 'left color "black" open?))
```

```
(define big-fish (fish "darkgray" #f)) 
```

```
(define big-aaah-fish (fish "darkgray" #t)) 
```

2.2 Adjusting Picts

If we need fish of different sizes after all, instead of adding arguments to `fish`, we can simply use Slideshow's `scale` function:

```
(define little-fish (scale (fish "dimgray" #f) 0.7)) 
```

We certainly want to place our fish into an aquarium, which we can draw as a light rectangle behind the fish. The `filled-rectangle` function does not accept a color argument; instead, it generates a rectangle that uses the default color, and the `colorize` function lets us adjust the default color for a pict.¹ Thus, we can create our rectangle as follows:

```
(colorize (filled-rectangle 20 10) "lightgray") 
```

2.3 Combining Picts

To create a pict with two fish, one way is to use `ht-append`:

```
(ht-append 10 little-fish big-aaah-fish) 
```

The first argument to `ht-append` is an amount of space to put between the pict. It is optional, and it defaults to 0. The `ht` part of the name `ht-append` indicates that the picts are horizontally stacked and top-aligned. Analogously, the `hb-append` function bottom-aligns pict. If, we want to center-align pict, we can use `hc-append`:

```
(define two-fish
  (hc-append 10 little-fish big-aaah-fish)) 
```

Now we are ready to place the fish in an aquarium. Our old aquarium,

```
(colorize (filled-rectangle 20 10) "lightgray") 
```

is not large enough to hold the two fish. We could choose a fixed size for the aquarium, but if we later change the size constants in the `fish` function, then the aquarium might not be the right size. We could scale the image, but a simpler strategy is to create a function `aq` that takes a pict and makes an aquarium for the pict:

```
;; aq : pict -> pict
(define (aq p)
  (colorize (filled-rectangle (pict-width p)
                              (pict-height p))
            "lightgray"))
```

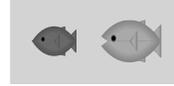
```
(aq two-fish) 
```

The `pict-width` and `pict-height` functions take a pict and produce its width and height, respectively. This aquarium is large enough to hold the fish, but it's tight. We can give the fish more room by adding space around `two-fish` with Slideshow's `inset` function, and then generate an aquarium pict. Finally, we put the fish and aquarium together using Slideshow's `cc-superimpose` function:

¹ A fish is drawn with multiple colors—the eye color, the body color, and intermediate shades—so its colors must be supplied to `fish`.

```
;; in-aq : pict -> pict
(define (in-aq p)
  (cc-superimpose (aq (inset p 10)) p))

(in-aq two-fish)
```



The leftmost argument to `cc-superimpose` is placed bottommost in the drawing, so the fish end up on top of the aquarium rectangle.

The `cc` part of the name `cc-superimpose` indicates that the picts are centered horizontally and vertically as they are stacked on top of each other. Slideshow provides a `-superimpose` function for each combination of `l`, `c`, or `r` (left, center, or right) with `t`, `c`, or `b` (top, center, or bottom).

One additional mode of alignment is useful for text. When combining text of different fonts into a single line of text, then neither `ht-append` nor `hb-append` produces the right result in general, because different fonts have different shapes. Slideshow provides `hbl-append` for stacking picts so that their baselines match.

```
;; lbs : num -> pict
(define (lbs amt)
  (hbl-append (text (number->string amt) '(bold . swiss) 9)
              (text " lbs" 'swiss 8)))

(define 10lbs (lbs 10))           10 lbs

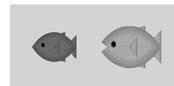
(define 7lbs (lbs 7))            7 lbs

(define 17lbs (lbs 17))         17 lbs
```

For multi-line text, `hbl-append` matches baselines for the bottommost lines. Slideshow provides `htl-append` to make baselines match for topmost lines. Naturally, Slideshow provides variants of `-superimpose` with `tl` and `bl`, as well.

Finally, Slideshow provides `vl-append`, `vc-append`, and `vr-append` to stack picts vertically with left-, center-, and right-alignment. To add the labels to our aquarium pict, we can use `vl-append` to first stack the aquarium on the “10 lbs” label, and then use `rbl-superimpose` to add the “7 lbs” label to the bottom-right corner of the pict:

```
(define two-fish+sizes
  (rbl-superimpose
    (vl-append 5
      (in-aq two-fish)
      7lbs)
    10lbs))
```



7 lbs 10 lbs

2.4 Finding Picts

Pict objects are immutable. A particular pict, such as `little-fish`, can be used in multiple independent contexts. Functions that adjust a pict, such as `scale`, do not change the given pict, but instead generate a new pict based on the given one. For example, we can combine `two-fish` and a scaled version of `two-fish` in a single pict:

```
(hc-append 10 two-fish (scale two-fish 0.5))
```



Picts nevertheless have an identity, in the sense of Scheme's `eq?`, and each use of a `Slideshow` function generates a pict object that is distinct from all other pict objects. This identity is useful for finding the location of pict within other pict.

For example, to add an arrow from "7 lbs" to the little fish, we could insert an arrow pict into the `vl-append` sequence (with negative space separating the stacked pict), but then adding an arrow from "10 lbs" to the big fish would be more difficult. Instead, `Slideshow` provides a way to add an arrow to the pict based on the relative location of the little-fish and "7 lbs" sub-picts.

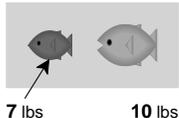
To locate a sub-pict within an aggregate pict, `Slideshow` provides a family of operations ending with `-find`. The prefix of a `-find` operation indicates which corner or edge of the sub-pict to find; it is a combination of `l`, `c`, or `r` (left, center, or right) with `t`, `tl`, `c`, `bl`, or `b` (top, top baseline, center, bottom baseline, or bottom). The results of a `-find` operation are the coordinates of the found corner/edge relative to the aggregate pict, where the aggregate pict's top-left corner is the origin, and x and y -coordinates increase going right and down.

A `-find` operation is often combined with `pin-over`, which takes a pict, coordinates, and a pict to place on top of the first pict. For example, we can create a connecting arrow with `pip-arrow-line` (which takes horizontal and vertical displacements, plus the size of the arrowhead) and place it onto `two-fish+sizes`:

```
(define-values (ax ay) (ct-find two-fish+sizes 7lbs))

(define-values (bx by) (cb-find two-fish+sizes little-fish))

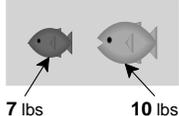
(pin-over two-fish+sizes
  ax ay
  (pip-arrow-line (- bx ax) (- by ay) 6))
```



Since we need multiple arrows, we abstract this pattern into a function:

```
;; connect : pict pict pict -> pict
(define (connect main from to)
  (define-values (ax ay) (ct-find main from))
  (define-values (bx by) (cb-find main to))
  (pin-over main
    ax ay
    (pip-arrow-line (- bx ax) (- by ay) 6)))

(define labeled-two-fish
  (connect two-fish+sizes
    7lbs little-fish
    10lbs big-aaah-fish))
```



`Slideshow` provides a function that is like `connect` called `pin-arrow-line`. In addition to the arguments of `connect`, `pin-arrow-line` accepts the `-find` functions for each sub-pict. Thus, `connect` can be implemented more simply as

```
;; connect : pict pict pict -> pict
(define (connect main from to)
  (pin-arrow-line 6 main from ct-find to cb-find))
```

Slideshow provides other libraries work well with finding pict. For example, the library "balloon.ss" provides `pip-wrap-balloon` for wrapping a pict with a cartoon balloon:

```
(define yikes
  (pip-wrap-balloon
    (text "Yikes!" '(bold . roman) 9)
    's 0 10)) ; spike direction and displacement

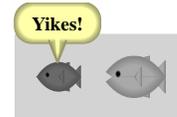
;; panic : pict -> pict
(define (panic p)
  (define-values (x y) (ct-find p little-fish))
  (pin-over p x y yikes))

(panic two-fish)
```



Each `pin-` operation preserves the bounding box of the first argument, instead of extending it to include the placed pict. This behavior is useful for adding arrows and balloons such that further compositions are unaffected by the addition. For example, we can still put the fish into an aquarium after adding the panic balloon.

```
(in-aq (panic two-fish))
```

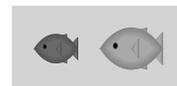


2.5 Ghosts and Laundry

We eventually want a slide sequence with variants of the aquarium pict. One variant should have the little and big fish together in the aquarium, as before:

```
(define both-fish
  (hc-append 10 little-fish big-fish))

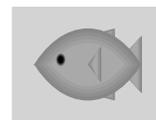
(in-aq both-fish)
```



Another variant should have just the big fish—now even bigger, since it has eaten the little fish. If we generate the pict as

```
(define bigger-fish
  (scale big-fish 1.7))

(in-aq bigger-fish)
```

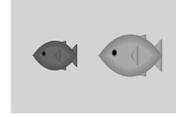


then our slides will not look right, because the aquarium changes shape from the first pict to the second.

We can avoid this problem by constructing a large enough aquarium for the first pict. Conceptually, we'd like to stack the large-fish pict on top of the pict with the two fish together, and then put the combined pict in the aquarium (so that it is large enough to fit both pict in both dimensions), and then hide the large-fish pict.

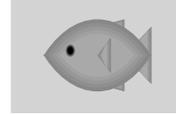
To hide a pict, the `ghost` function takes a pict and generates a pict with the same bounding box and sub-picts as the given one, but with no visible drawing. Thus, we can create the right initial pict as follows:

```
(define all-fish-1
  (in-aq (cc-superimpose
    both-fish
    (ghost bigger-fish))))
```



We can create the last slide by ghosting `both-fish` instead of `bigger-fish`:

```
(define all-fish-2
  (in-aq (cc-superimpose
    (ghost both-fish)
    bigger-fish)))
```

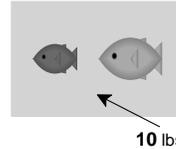


Since both `all-fish-1` and `all-fish-2` contain all three fish, they are guaranteed to be the same size.

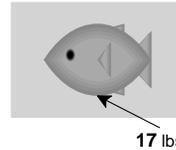
If we try to add a label and arrow for the big fish, however, something goes wrong:

```
;; add-big-label : pict pict -> pict
(define (add-big-label all-fish wt)
  (let ([labeled (vr-append 5 all-fish wt)])
    (connect labeled wt bigger-fish)))
```

```
(add-big-label all-fish-1 10lbs)
```



```
(add-big-label all-fish-2 17lbs)
```

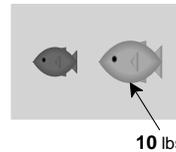


The problem is that `bigger-fish` is a scaled version of `big-fish`, and even though `bigger-fish` is ghosted in `all-fish-1`, `ghost` merely replaces the `big-fish` drawing procedure, but not its sub-picts. Specifically, `big-fish` exists twice in each `all-fish-` pict, and `add-big-label` finds the wrong one in `all-fish-1`.

To hide a sub-pict's identity, Slideshow provides `launder`² as a complement to `ghost`. The `launder` function takes a `pict` and produces a `pict` with the same dimensions and drawing, but with a new identity and without sub-picts. To ensure that `add-big-label` finds the right `big-fish`, we can both `ghost` and `launder` the `pict` to hide.

```
(define all-fish-1
  (in-aq (cc-superimpose
    both-fish
    (launder (ghost bigger-fish)))))
```

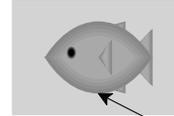
```
(add-big-label all-fish-1 10lbs)
```



² We use "launder" in the sense of "to launder money," which is to obscure the source of money without changing its face value.

```
(define all-fish-2
  (in-aq (cc-superimpose
        (launder (ghost both-fish))
        bigger-fish)))

(add-big-label all-fish-2 17lbs)
```



17 lbs

2.6 From Pictures to Slides

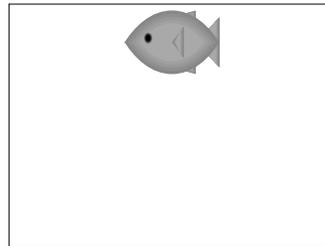
Pict-construction primitives are only half of Slideshow's library. The other half defines pict operations that support common slide tasks and that cooperate with a slide-display system. Common tasks include creating a slide with a title, creating text with a default font, breaking lines of text, bulletizing lists, and staging bullet lines. Cooperating with the display includes correlating titles with a slide-selection dialog and enabling clickable elements within interactive slides.

Ultimately, a slide is represented as a single pict to be drawn on the screen. The screen's drawing context is scaled so that the slide fits when it is 1024 units wide and 768 units high, which means that each drawing unit for a slide is one pixel on a typical projector.

2.7 Registering Slides

Since a slide presentation is a sequence of picts, a presentation could be represented as a list, and a Slideshow program could be any program that generates a list of picts. We have opted instead for a more imperative design at the slide level: a Slideshow program calls a `slide` function (or variants of `slide`) to register each individual slide's content:³

```
(slide
 (scale big-fish 10))
```

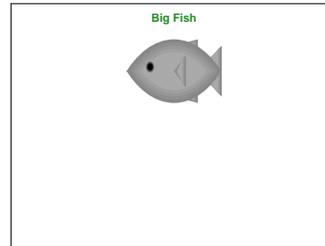


We choose imperative registration through `slide` because a slide presentation is most easily written as a sequence of interleaved definitions and expressions (invocations of `slide`), much like the examples in section 2, and much like any other Scheme program. To make the interface more functional, we might require a Slideshow program to produce a list of slides, but threading this list through the top-level sequence of definitions is awkward to read and maintain. The picts that are registered for slides remain purely functional (i.e., they cannot be mutated), so a small amount of imperative programming causes little problem in practice. A drawback of imperative registration is that Slideshow programs are less easily reused in other programs; so far, we have been content to manually refactor programs to support reuse.

³ We illustrate the effect of `slide` by showing a framed, scaled version of the resulting slide's pict.

The `slide/title` function is similar to `slide`, except that it takes an extra string argument. The string is used as the slide's name (for selecting the slide in an outline), and it is also used to generate a title pict that is placed above the supplied content pict. The title pict uses a standard (configurable) font and is separated from the slide content by a standard amount.

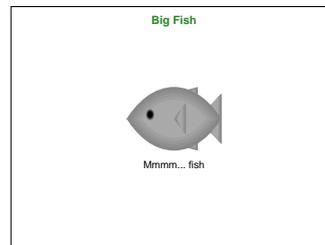
```
(slide/title "Big Fish"
 (scale big-fish 10))
```



In principle, the `slide` and `slide/title` functions simply register slides, and programmers could build more elaborate slide functions in terms of these functions. In practice, programmers will prefer to use the more elaborate functions, and part of Slideshow's job is to provide useful elaborations. Thus, Slideshow allocates the relatively short names `slide`, `slide/title`, etc. to functions that accept multiple pict.

When multiple picts are provided to the `slide` function, `slide` combines the picts with `vc-append` using a separation of `gap-size` (which is 24). The `slide` function then `ct-superimposes` the appended picts with a blank pict representing the screen (minus a small border). The `slide/center` function is like `slide`, except that it centers the slide content with respect to the screen. The `slide/title/center` function accepts a title and also centers the slide.

```
(slide/title/center "Big Fish"
 (scale big-fish 10)
 (text "Mmmm... fish" 'swiss 32))
```



The set of pre-defined `slide` layouts includes only the layouts that we have found to be most useful. Programmers can easily create other layouts by implementing functions that call `slide`.

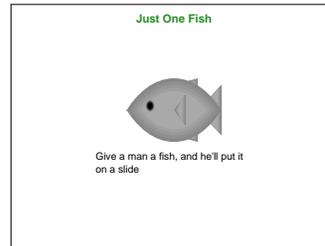
2.8 Managing Text

In the spirit of providing short names for particularly useful functions, Slideshow provides the function `t` for creating a text pict with a standard font and size (which default to a sans-serif font and 32 units, respectively). Thus, the label in the preceding example could have been implemented as `(t "Mmmm... fish")` instead of `(text "Mmmm... fish" 'swiss 32)`. The `bt` function is similar to `t`, except that it makes the text bold, and it makes text italic.

For typesetting an entire sentence, which might be too long to fit on a single line and

might require multiple fonts, Slideshow provides a `para` function. The `para` function takes a width and a sequence of strings and `picts`, and it arranges the text and `picts` as a paragraph that is bounded by the given width. In the process, `para` may break strings on word boundaries (it does not hyphenate words). The width argument to `para` is often based on the `client-w` constant, which is defined as the width of the entire slide minus a small margin.

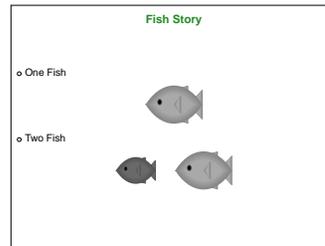
```
(slide/title/center "Just One Fish"
 (scale big-fish 10)
 (para (* 1/2 client-w)
  "Give a man a fish, and"
  "he'll put it on a slide"))
```



The `page-para` function is like `para`, but with `client-w` as the implicit first width.

Slideshow accommodates bulleted lists with the `item` function. It is similar to `para`, except that it adds a bullet to the left of the paragraph. In parallel to `page-para` and `para`, the `page-item` function is like `item`, but with `client-w` built in.

```
(slide/title/center "Fish Story"
 (page-item "One Fish")
 (scale big-fish 6)
 (page-item "Two Fish")
 (scale (hc-append 10
  little-fish
  big-fish)
  6))
```



Naturally, `item` is easily implemented in terms of `para` and a bullet `pict`:

```
;; item : num pict ... -> pict
(define (item w . picts)
  (htl-append (/ gap-size 2)
  bullet
  (apply para
  (- w
  (pict-width bullet)
  (/ gap-size 2))
  picts)))
```

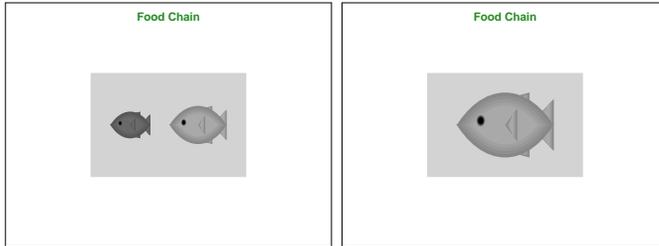
Just as Slideshow provides many `slide` variants, it also provides many `para` and `item` variants, including variants for right-justified or centered paragraphs and bulleted sub-lists. The `para*` function, for example, typesets a paragraph like `para`, but it allows the result to be narrower than the given width (in case no individual line fills exactly the given width).

2.9 Staging Slides

One way to stage a sequence of slides is to put one `pict` for each slide in a list, and then apply a slide-generating function to each element of a list of `picts`:

```
;; aq-slide : pict -> void
(define (aq-slide all-fish)
  (slide/title/center "Food Chain"
    (scale all-fish 6)))

(for-each aq-slide (list all-fish-1 all-fish-2))
```

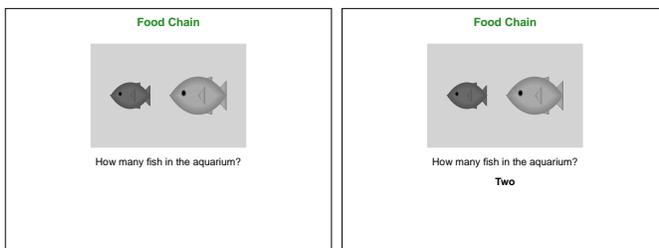


Interesting presentations usually build on this idea, and we discuss it more in section 3.

Much like `text` and `hbl-append` for typesetting paragraphs, however, this strategy is awkward for merely staging bullets or lines of text on a slide. For example, when posing a question to students, an answer may be revealed only after the students have a chance to think about the question.

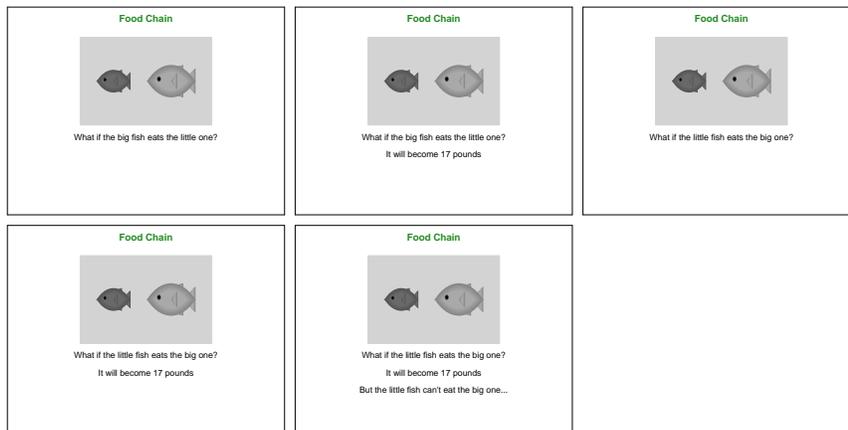
To support this simple kind of staging, the `slide` function (and each of its variants) treats the symbol `'next` specially when it appears in the argument sequence. All of the `picts` before `'next` are first used to generate a slide, and then the `picts` before `'next` plus the arguments after `'next` are used to generate more slides.

```
(slide/title "Food Chain"
  (scale all-fish-1 6)
  (t "How many fish in the aquarium?")
  'next
  (bt "Two"))
```



The `'next` symbol simplifies linear staging of `slide` content. The `slide` function also supports tree-like staging of content through the `'alts` symbol. The argument following `'alts` must be a list of lists, instead of a single `pict`. Each of the lists is individually combined with the preceding `picts` to generate a set of slides. The final list is further combined with the remaining arguments (after the list of lists). The `'next` and `'alts` symbols can be mixed freely to generate sub-steps and sub-trees.

```
(slide/title "Food Chain"
  (scale all-fish-1 6)
  'alts
  (list (list (t "What if the big fish eats the little one?")
             'next
             (t "It will become 17 pounds")))
        (list (t "What if the little fish eats the big one?")
             'next
             (t "It will become 17 pounds"))))
  'next
  (t "But the little fish can't eat the big one..."))
```



With `slide`, `page-item`, `'next`, `'alts`, etc., a programmer can build a text-oriented presentation in Slideshow almost as easily as in PowerPoint. Our overall goal, however, is not to encourage bullet-point presentations (Tufte, 2003), but instead to simplify the creation of interesting, graphical presentations. So, we now turn our attention to the *design* of slide presentations.

3 How to Design Slide Presentations

The strength of functional programming is that it supports and encourages systematic program design. We believe that Slideshow supports and encourages good *presentation* design, not only because it is based on functional programming, but because it directly supports a design recipe (Felleisen *et al.*, 2001) for image sequences.

Often, an image sequence resembles a scene in a play, where actors move together to illustrate some point in a presentation. Simple scenes can be implemented easily through ad-hoc parameterization of a `pict` expression, but assembling a complex scene requires more careful planning. The design recipe presented in this section distills our past success in building complex scenes. To demonstrate the recipe, we animate the scene from Section 2 showing conservation of mass in an aquarium: when one fish eats another, the total mass of the aquarium content does not change.

The first step in designing a scene is to create a storyboard. A storyboard sketches the sequence of individual frames, each of which corresponds to a slide. Our example storyboard is shown in figure 1.

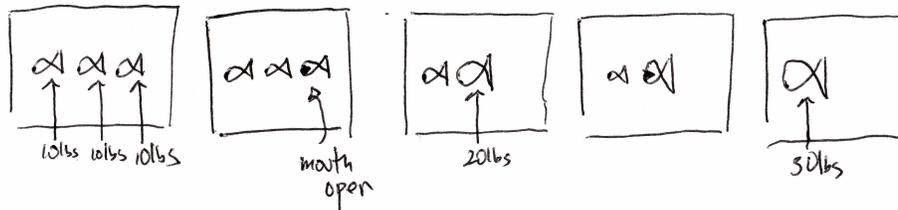


Fig. 1. Storyboard for the example talk

Once we have a storyboard, we must identify the *main characters*—that is, the elements of the picture that define the overall layout across frames. We call the rest of the frame elements the *supporting cast*; they will be layered on the basic frame as determined by the main characters.

In this case, the fish are the main characters, because the layout of the fish determines the overall shape of each frame in the sequence. The water, weights, and arrows are the supporting cast, because we can draw the aquarium under the frame shaped by the fish, and we can draw the arrows and weight labels on top.

The programming task is to convert a scene and the characters into expressions that generate pict for the frames. We proceed in four steps:

1. Define an expression that produces the pict for each main character. A particular character may change shape over multiple frames, so design one expression per view of each character.

In this case, we have three characters, which are the three fish in the first frame. The first two fish have only a single view each. The last fish has five: small with its mouth open and closed, medium-sized with its mouth open and closed, and large with its mouth closed.

2. Define a single pict expression that includes all of the views of the main characters from all frames in the scene. This ensemble pict may include a character in multiple places if the character appears in different places in different frames. To generate a particular frame, we parameterize the ensemble pict expression to hide characters that do not appear in the frame.

In our running example, we must generate a pict that has the three fish in a line, plus different views of the last fish on top of those fish. For example, the largest fish will be on top of the first fish with their left edges touching.

3. Define pict expressions for the supporting cast. Unlike the main cast, each supporting cast pict should have the same dimensions as the entire frame, so it can be superimposed on its corresponding frame. Each supporting cast member is placed into an appropriately sized blank pict at a position that is determined by its location in the complete frame.

In our example, we design overlays for the arrows and for the weights. Each weight will be centered below its corresponding fish, and the arrows point from the top of the weight to the bottom of the fish.

- Define an expression that, for each frame, combines the main-character pict and the supporting-cast picts.

Each of the following sections implements one step in the recipe for our example scene.

3.1 Main Characters

Our expressions for main-character picts can re-use the `fish` function from section 2:

```
(define lunch-fish
  (fish "dimgray" #f))
```



```
(define dinner-fish
  (launder lunch-fish))
```



```
(define small-fish
  (fish "darkgray" #f))
```



```
(define small-aaah-fish
  (fish "darkgray" #t))
```



```
(define medium-fish
  (launder (scale small-fish (sqrt 2))))
```



```
(define medium-aaah-fish
  (launder (scale small-aaah-fish (sqrt 2))))
```



```
(define large-fish
  (launder (scale small-fish (sqrt 3))))
```



In general, each of the main characters in our scene must have a separate identity from each of the others and must not contain any of the others as sub-picts, so we launder all of the picts. Otherwise, the overlays we build in step 3 might find the wrong main character.

3.2 Frames

The first frame in our scene has three fish lined up in a row, so we begin by combining our fish with `hc-append`. We add space between the fish using a blank pict, instead of an initial number to `hc-append`, to make the space explicit:

```
(define spacer (blank 10 0))

(hc-append dinner-fish spacer
  lunch-fish spacer
  small-fish)
```



Next, we add the eating fish: `small-aaah-fish` through `large-fish`. The position of each eating fish is determined by one of the three initial fish. Since the nose of each eating fish, such as `medium-fish`, should line up with the nose of an existing fish, such as `lunch-fish`, we might try to add the eating fish into the scene with `lc-superimpose`:

```
(hc-append (lc-superimpose
  large-fish
  dinner-fish)
  spacer
  (lc-superimpose
  medium-fish
  medium-aaah-fish
  lunch-fish)
  spacer
  (lc-superimpose
  small-aaah-fish
  small-fish))
```



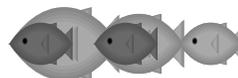
On close examination of the spacing, however, we can see that the medium and large fish have stretched the space between the original pict. In fact, the space between the dinner and lunch fish is no longer the same as the space between the lunch and small fish. To see the spacing problem clearly, we can ghost out the medium and large fish and compare it with the original pict that shows just the first frame:

```
(hc-append (lc-superimpose
  (ghost large-fish)
  dinner-fish)
  spacer
  (lc-superimpose
  (ghost medium-fish)
  (ghost medium-aaah-fish)
  lunch-fish)
  spacer
  (lc-superimpose
  small-aaah-fish
  small-fish))
```



We fix this problem by moving calls to `hc-append` inside calls to `lc-superimpose`:

```
(lc-superimpose
  large-fish
  (hc-append dinner-fish
    spacer
    (lc-superimpose
      medium-fish
      medium-aaah-fish
      (hc-append lunch-fish
        spacer
        (lc-superimpose
          small-aaah-fish
          small-fish))))))
```



Now, the large fish and the medium fish overlap with the fish to the right, but the storyboard shows that the overlapping fish will never appear together in a single frame. Meanwhile, this pict preserves the original spacing of the dinner, lunch, and small fish.

Once we have the basic layout designed, the design recipe tells us to parameterize it so we can hide certain fish. In this case, we parameterize the scene with a list of the fish that we want to show, and we define an `on` helper function to either show or hide each character in the scene:

```
;; main-characters : (listof pict[main-char]) -> pict[frame]
(define (main-characters active)
  (lc-superimpose
   (on active large-fish)
   (hc-append (on active dinner-fish)
              spacer
              (lc-superimpose
               (on active medium-fish)
               (on active medium-aaah-fish)
               (hc-append (on active lunch-fish)
                          spacer
                          (lc-superimpose
                           (on active small-aaah-fish)
                           (on active small-fish)))))))

;; on : (listof pict[main-char]) pict[main-char] -> pict
(define (on set pict) (if (memq pict set) pict (ghost pict)))
```

To assemble the main characters for each frame, we simply apply `main-characters` to a list of fish to show:

```
(define main1
  (main-characters (list dinner-fish
                        lunch-fish
                        small-fish)))
```



3.3 Supporting Cast

For the supporting cast, we start with the fish's weights. Typically, a supporting cast member's position depends on one or more main characters. As it happens, we can go even further for the weights by generating the label `pict` based on the area of a fish:

```
;; weight : pict[main-char] -> num
(define (weight fish)
  (inexact->exact (round (* (pict-width fish)
                           (pict-height fish)
                           1/60))))

(define small-lbs (lbs (weight small-fish))) 10 lbs
```

To place the weight of the fish directly below the fish, we first compute the position of the fish. Then, we can use `pin-over` to combine a frame with the weight. The second argument to `pin-over` is the horizontal position of the left edge of the weight, and we compute a position to center the weight below the fish. The third argument to `pin-over` is the vertical position, and we compute a position for the weight just below the aquarium.

```
(define-values (fx fy) (cc-find main1 small-fish))

(define main1/weight
  (pin-over main1
            (- fx
              (/ (pict-width small-lbs)
                 2))
            (+ (pict-height main1) 6)
            small-lbs))
```



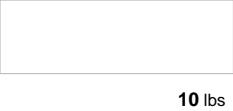
10 lbs

Combining the weight and the original scene in this manner helps us test our computation of the weight's coordinates, but the design recipe tells us to build an overlay pict that contains only the weight. Building the overlay as a separate pict enables us to add or remove it without disturbing the rest of the scene.

We can obtain a separate pict for the weight overlay by simply ghosting the input frame:⁴

```
;; weight-overlay : pict[frame] pict[main-char] pict[wt] -> pict
(define (weight-overlay frame fish weight)
  (define-values (fx fy) (cc-find frame fish))
  (pin-over (ghost frame)
            (- fx (/ (pict-width weight) 2))
            (+ (pict-height frame) 6)
            weight))

(add-bbox
 (weight-overlay main1
                 small-fish
                 (lbs (weight small-fish))))
```



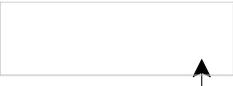
The other supporting cast members are the arrows. We use `pin-arrow-line` to draw an arrow from the small fish's weight to the small fish:

```
(pin-arrow-line 8
                main1/weight
                small-lbs ct-find
                small-fish cb-find)
```



Again, to satisfy the design recipe for this step, we must build a pict that draws only the arrow:

```
(add-bbox
 (pin-arrow-line 8
                 (ghost main1/weight)
                 small-lbs ct-find
                 small-fish cb-find))
```



Once we have designed the arrow overlay for a single frame, we can abstract over its construction, as with the weight overlay:

```
;; arrow-overlay : pict[frame] pict[main-char] pict[wt] -> pict
(define (arrow-overlay frame fish lbs)
  (pin-arrow-line 8
                  (ghost frame)
                  lbs ct-find
                  fish cb-find))
```

Now that we have defined the two overlays, we can write a single function that combines them. Since `pin-over` and `pin-add-line` do not adjust the bounding box of the ghosted frame when creating an overlay, we combine overlays with `cc-superimpose`:

⁴ We use `add-bbox`, which adds a gray box to the given pict, to show the location of the bounding box in a mostly blank image.

```
;; overlay : pict[frame] pict[main-char] -> pict
(define (overlay pict fish)
  (let* ([weight (lbs (weight fish))]
         [w/weight (weight-overlay pict fish weight)])
    (cc-superimpose
     w/weight
     (arrow-overlay w/weight fish weight))))
```



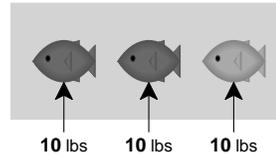
```
(add-bbox (overlay main1 small-fish))
```

3.4 Putting It All Together

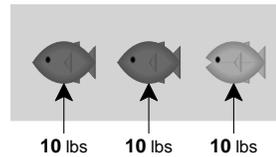
All that remains is to define a function that puts together the aquarium, the main characters, and the overlays, and then call the function once for each frame in the scene:

```
;; fish-scene : (listof pict[main-char]) -> pict
(define (fish-scene active-characters)
  (let ([frame (in-aq (main-characters active-characters))])
    (apply cc-superimpose
     frame
     (map (lambda (fish) (overlay frame fish))
          active-characters))))
```

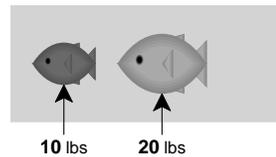
```
(fish-scene (list dinner-fish
                  lunch-fish
                  small-fish))
```



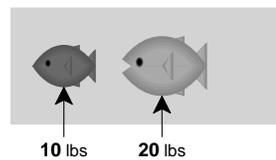
```
(fish-scene (list dinner-fish
                  lunch-fish
                  small-aaah-fish))
```



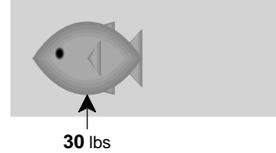
```
(fish-scene (list dinner-fish
                  medium-fish))
```



```
(fish-scene (list dinner-fish
                  medium-aaah-fish))
```



```
(fish-scene (list large-fish))
```



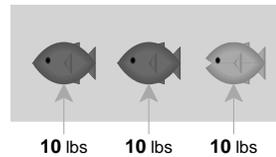
3.5 Exploiting the Design Recipe

By following the design recipe, we have established a clear connection between the scene's storyboard and its source code. If we imagine a change to the storyboard, the design recipe helps us determine how to modify the implementation.

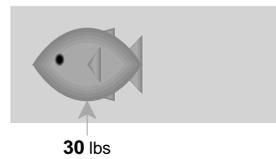
For example, suppose we decide to color the arrows that connect the weights to the fish. The corresponding code is the `arrow-overlay` function. Simply adding a `colorize` expression to its body produces colored arrows.

```
;; arrow-overlay : pict pict pict -> pict
(define (arrow-overlay frame fish lbs)
  (colorize
    (pin-arrow-line 8
      (ghost frame)
      lbs ct-find
      fish cb-find)
    "darkgray"))
```

```
(fish-scene (list dinner-fish
              lunch-fish
              small-aaah-fish))
```

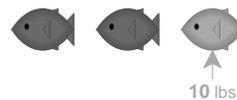


```
(fish-scene (list large-fish))
```



This modification relies on implementing the supporting cast as overlays. If we had not followed the design recipe and placed the arrow directly on the scene, `colorize` would have colored the weights as well as the arrows.

```
;; !! Incorrectly colors the weight !!
(colorize
  (pin-arrow-line 8
    main1/weight
    small-lbs ct-find
    small-fish cb-find)
  "darkgray")
```



For a second example, suppose we want to make the fish grow pudgier as it eats, instead of growing equally in both dimensions. This change to the storyboard affects only the main characters. Correspondingly, we need to modify only the definitions of the main characters:

```
;; pudgy : pict num -> pict
(define (pudgy p n) (scale p (expt n 1/4) (expt n 3/4)))
```

```
(define medium-fish
  (launder (pudgy small-fish 2)))
```



```
(define medium-aaah-fish
  (launder (pudgy small-aaah-fish 2)))
```

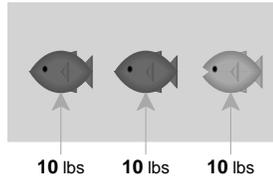


```
(define large-fish
  (launder (pudgy small-fish 3)))
```

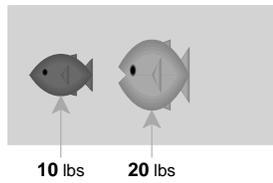


The aquarium and arrows adapt automatically to the new fish dimensions:

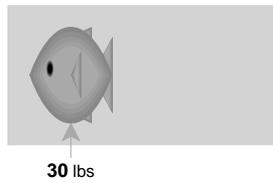
```
(fish-scene (list dinner-fish
  lunch-fish
  small-aaah-fish))
```



```
(fish-scene (list dinner-fish
  medium-aaah-fish))
```



```
(fish-scene (list large-fish))
```



Finally, imagine generalizing from three fish to an arbitrary number of fish. Although this change is more complex, the design recipe suggests how to analyze these changes, and it motivates a set of modifications to the code.

The first step is to adapt our definitions of the main characters. We can organize them into three lists: the fish that are eaten, the fish that do the eating with their mouths closed, and the fish that do the eating with their mouths open.

```
(define num-fish 9)

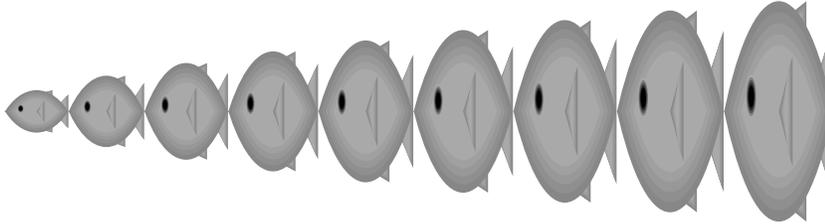
(define meal-fish
  (build-list (- num-fish 1) (lambda (i) (launder lunch-fish))))

(apply hc-append meal-fish)
```



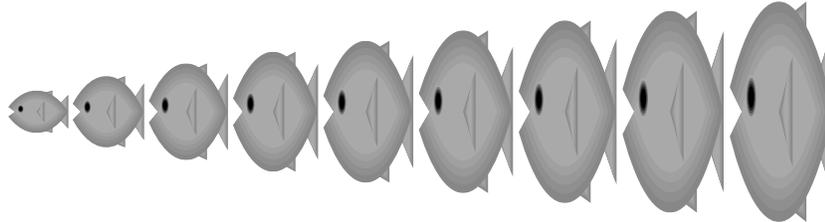
```
(define eating-fish
  (build-list num-fish
    (λ (i) (launder (pudgy small-fish (+ i 1))))))

(apply hc-append eating-fish)
```



```
(define eating-aaah-fish
  (build-list num-fish
    (λ (i) (launder (pudgy small-aaah-fish (+ i 1))))))

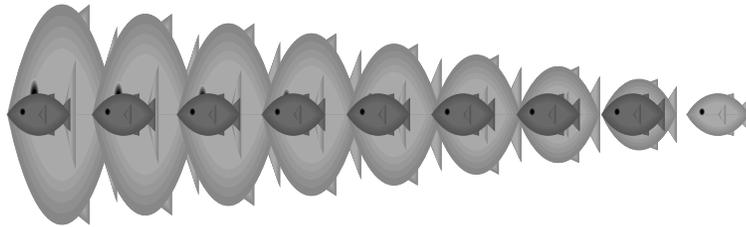
(apply hc-append eating-aaah-fish)
```



This modification also affects the second step of the design recipe: the placement of the main characters. We must re-define `main-characters` to generalize from just three fish to an arbitrary number of fish. With the exception of the first eating fish, each eating fish is `lc-superimposed` onto one of the food fish, and the rest of the scene is placed just behind with `spacer` in between. The procedure in the body of `main-characters` constructs this portion of the scene. Using `foldl`, we iterate over the three lists of fish. The second argument to `foldl` is the initial `pict`, which contains only the first eating fish. The last three arguments of `foldl` correspond to lists of arguments to the folding procedure.

```
;; main-characters : (list-of pict[main-char]) -> pict
(define (main-characters active)
  (foldl (λ (meal-fish eating-fish eating-aaah-fish so-far)
          (lc-superimpose
            (on active eating-fish)
            (on active eating-aaah-fish)
            (hc-append (on active meal-fish) spacer so-far)))
        (cc-superimpose (on active (car eating-fish))
                        (on active (car eating-aaah-fish)))
        meal-fish
        (cdr eating-fish)
        (cdr eating-aaah-fish)))

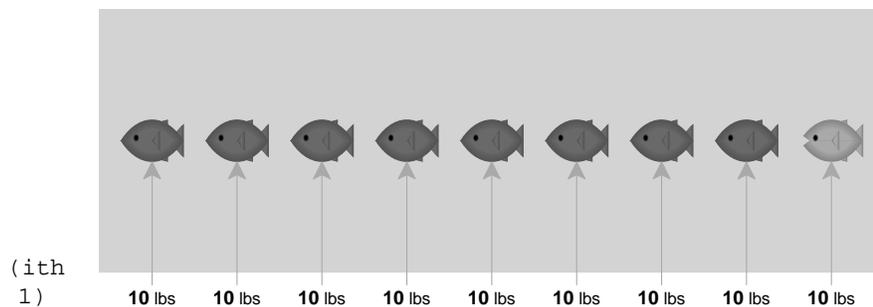
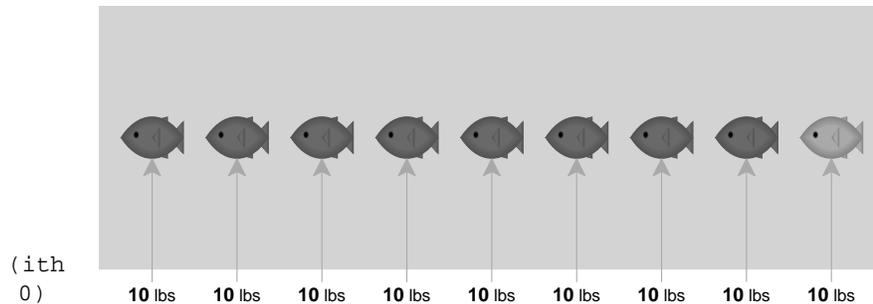
(main-characters (append meal-fish eating-fish eating-aaah-fish))
```

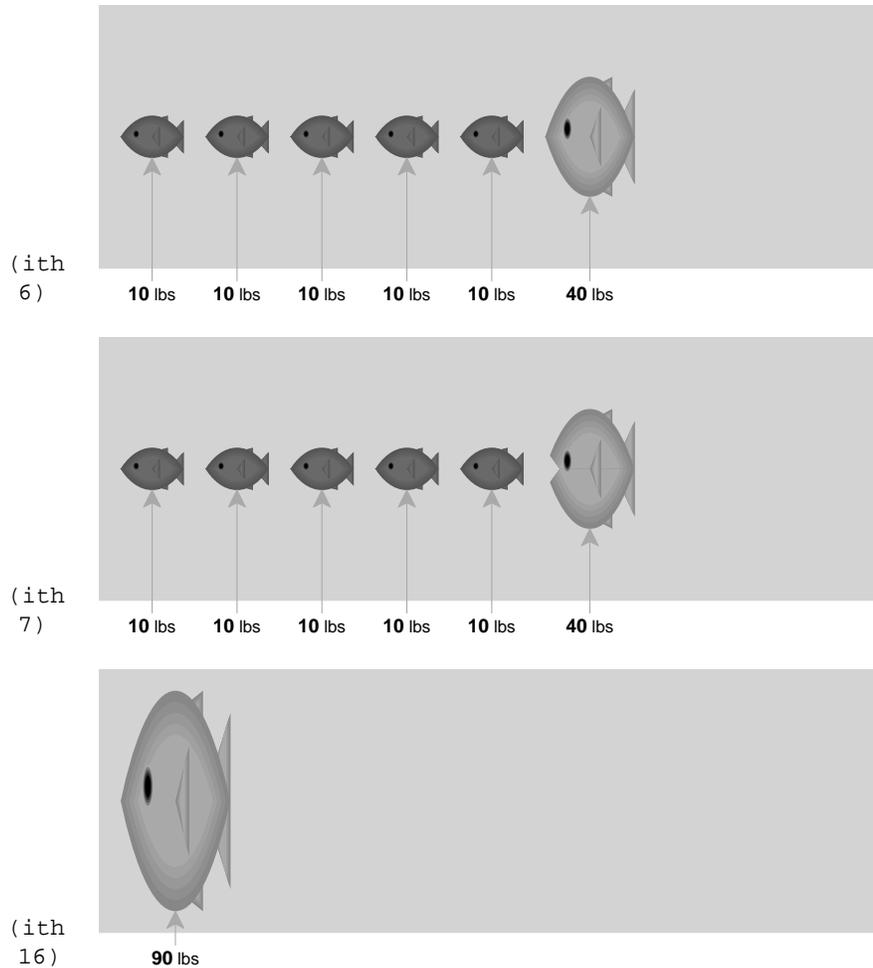


The weight and arrow overlays remain the same, per fish, in the storyboard, so the code from the third step of the design recipe does not change.

To follow the recipe's fourth step and complete the modification, we define `ith` to accept a frame number and produce the corresponding pict, using our earlier definition of `fish-scene`.

```
;; ith : num -> pict
(define (ith i)
  (fish-scene
    (cons (list-ref (if (even? i)
                      eating-fish
                      eating-aaah-fish)
                  (quotient i 2))
          (sublist meal-fish (quotient i 2) (- num-fish 1))))))
```





Overall, generalizing to an arbitrary number of fish required us to abstract over the fish construction and to write some list-processing functions. We sometimes use the same ideas to demonstrate the behavior of an algorithm in a presentation. After designing a scene that represents a snapshot of an algorithm, animating the algorithm is merely a matter of adding a call in the algorithm's body to generate snapshot pict.

4 Implementation

Slideshow is implemented as a PLT Scheme application (PLT, 2005). PLT Scheme provides the primitive drawing context for rendering pict, as well as the widget toolbox for implementing the Slideshow presentation window and other interface elements.

Slideshow is designed to produce consistent results with any projector resolution, as well as when generating PostScript versions of slides. The main challenges for consistency concern pict scaling and font selection, as discussed in the following sections. We also comment on Slideshow's ability to condense staged slides for printing, to pre-render slides to minimize delays during a presentation, and to trigger interactions during a presentation.

4.1 Drawing Picts

The `dc` function connects PLT Scheme's low-level drawing facilities with the world of `picts`. The `dc` function takes an arbitrary drawing procedure and a height, width, ascent, and descent for the `pict`. When the `pict` is to be rendered, the drawing procedure is called with a target drawing context and offsets.

For example, the `blank` `pict` constructor can be implemented as

```
;; blank : num num -> pict
(define (blank w h)
  (dc (λ (dest x y)
      (void))
      w h h 0))
```

More interestingly, we can define a `hook` function that is implemented using the methods `draw-spline` and `draw-line` of a primitive drawing context.

```
;; hook : num num -> pict
(define (hook w h)
  (dc (λ (dest x y)
      (let ([mid-x (+ x (/ w 2))]
            [mid-y (+ y (/ h 2))]
            [far-x (+ x w)]
            [far-y (+ y h)]
            [d (/ w 3)])
        (send dest draw-spline
              x y x far-y mid-x far-y)
        (send dest draw-spline
              mid-x far-y far-x far-y far-x mid-y)
        (send dest draw-line
              far-x mid-y (- far-x d) (+ mid-y d))))
      w h 0 h))
```

```
(define go-fish
  (ht-append 5 (hook 5 12) lunch-fish))
```



The primitive drawing context is highly stateful, with attributes such as the current drawing color and drawing scale. Not surprisingly, slides that are implemented by directly managing this state are prone to error, which is why we have constructed the `pict` abstraction. Nevertheless, the state components show up in the `pict` abstraction in terms of attribute defaults, such as the drawing color or width of a drawn line. In particular, the `linewidth`, `colorize`, and `scale` primitives change the line width, color, and scale of a `pict` produced by `hook`:

```
(scale
  (colorize (linewidth 1 go-fish) "darkgray")
  2)
```



Although the underlying drawing context includes a current font as part of its state, a `pict`'s font cannot be changed externally, unlike the `pict`'s scale, color, or line width. Changing a `pict`'s font would mean changing the font of sub-`picts`, which, in turn, would cause the bounding box of each sub-`pict` to change in a complex way, thus invalidating computations based on the sub-`pict` boxes. We discuss this design constraint further in section 4.3.

4.2 Scaling

Since 1024x768 displays are most common, Slideshow defines a single slide to be a `pict` with a bounding box of 1024x768 units, which makes drawing units match pixels on most projectors.⁵ Depending on the display at presentation time, the `pict`'s drawing context is scaled; for example, it is scaled by a factor of $\frac{25}{32}$ for an 800x600 display. If the display's aspect ratio is not 4:3, then scaling is limited by either the display's width or height to preserve the `pict`'s original 4:3 aspect ratio, and unused screen space is painted black. Unused space typically appears when viewing slides directly on a desktop machine, such as during slide development.

A Slideshow presentation is a program, and `pict` layouts are computed every time the presentation is started. (This computation is rarely so expensive that it interferes with interactive development.) Consequently, the target screen resolution is known at the time when slides are built. This information can be used, for example, to scale bitmap images to match the display's pixels, instead of units in the virtual 1024x768 space. Furthermore, information about the display is also useful for just-in-time font selection.

4.3 Fonts

Fonts are not consistently available across operating systems, or even consistently named. To avoid platform dependencies, Slideshow presentations typically rely on PLT Scheme's mapping of platform-specific fonts through portable "family" symbols, such as `'roman` (a serif font), `'swiss` (a sans-serif font, usually Helvetica), `'modern` (a fixed-width font), and `'symbol` (a font with Greek characters and other symbols). PLT Scheme users control the family-to-font mapping, so a Slideshow programmer can assume that the user has selected reasonable fonts. Alternately, a programmer can always name a specific font, though at the risk of making the presentation unportable.

Since specific fonts vary across platforms, displays, and users, the specific layout of `picts` in a Slideshow presentation can vary, due to different bounding boxes for text `picts`. Nevertheless, as long as a programmer uses `pict-width` and `pict-height` instead of hard-wiring text sizes, slides display correctly despite font variations. This portability depends on computing `pict` layouts at display time, instead of computing layouts in advance and distributing pre-constructed `picts`.

Text scaling leads to additional challenges. For many displays, a font effectively exists only at certain sizes; if a `pict` is scaled such that its actual font size would fall between existing sizes, the underlying display engine must substitute a slightly larger or smaller font. Consequently, a simple scaling of the bounding box (in the 1024x768 space) does not accurately reflect the size of the text as it is drawn, leading to overlapping text or unattractive gaps. Slideshow avoids many text-scaling problems by checking the expected scaling of slides (based on the current display size) when generating text `picts`. The programmer can optionally control this text-scale correction directly.

⁵ The default border introduced by `slide` leaves a 984x728 region for slide content.

4.4 Printing Slides

A drawing context in PLT Scheme is either a bitmap display (possibly offscreen) or a PostScript stream. Thus, printing a Slideshow presentation is as simple as rendering the slides to a PostScript drawing context instead of a bitmap context.

Slideshow provides a “condense” mode for collapsing staged slides. In condense mode, Slideshow automatically ignores `'next` annotations; a programmer can use `'next!` instead of `'next` to force separate slides in condense mode. In contrast, `'alts` annotations cannot be ignored, because each alternative can show different information. A Slideshow programmer can indicate that intermediate alternatives should be skipped in condense mode by using `'alts~` instead of `'alts`.

Slideshow synchronizes page numbering in condensed slides with slide numbering in a normal presentation. In other words, when `slide` skips a `'next` annotation, it also increments the slide number. As a result, a condensed slide’s “number” is actually a range, indicating the range of covered slides.

Programmers can use the `condense?` and `printing?` predicates in `pict`-generating expressions to further customize slide rendering. A `skip-slides!` function allows the programmer to increment the slide count directly, in case a customization for condense mode skips a slide.

4.5 Pre-rendering Slides

The time required to render a slide is normally imperceptible, but since a programmer can create arbitrarily complex `picts` or write arbitrarily complex code that uses the drawing context directly, the rendering delay is sometimes significant. To ensure instantaneous response in the common case, Slideshow pre-renders the next slide in the presentation sequence while the speaker dwells on the current slide. If the speaker requests a slide change within half a second, the slide is not pre-rendered, because the speaker is likely stepping backward through slides.

4.6 Display Interaction

In addition to creating pictures for the screen, slide presenters must sometimes interact more directly with the display system.

Commentary: A slide author might wish to attach a commentary to slides, for the benefit of the speaker or for those viewing the slides after the talk. Slideshow provides a `comment` constructor that takes a commentary string and produces an object that can be supplied to `slide`. When the `slide` function finds a comment object, it accumulates the comment into the slide’s overall commentary (instead of generating an image). The Slideshow viewer displays a slide’s commentary on demand in a separate window.

Multiple displays: If a speaker’s computer supports multiple displays, the speaker might like to see comments and a preview on a private display. Thus, in addition to the separate commentary window, Slideshow provides a preview window that shows the current slide and the next slide in the presentation.

Demo hyperlinks: For presentations that involve demos, the speaker might like hyperlinks on certain slides to start the demos. Slideshow provides a `clickback` operator that

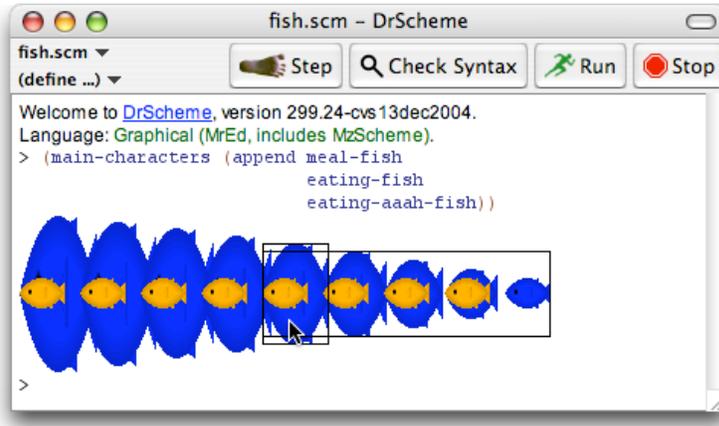


Fig. 2. Picts in DrScheme’s REPL

takes a pict and a procedure of no arguments; the result is a pict that displays like the given one, but that also responds to mouse clicks by calling the given procedure. (In this case, we exploit the fact that slide generation and slide presentation execute on the same virtual machine.)

Animation: Although many “animations” can be implemented as multiple slides that the speaker advances manually, other animations should be entirely automatic, such as moving an imagine across the screen in real time. Currently, Slideshow provides only minimal support for such animations through an imperative `scroll-transition` function that registers a scroll animation over the previously registered slide. (This feature has been used mainly to animate an algebraic reduction, making the expression movements easier to follow.) In the future, we might enrich the pict abstraction to support more interesting kinds of animation, as discussed in section 6.3.

5 Environment Support

Slideshow programs can be implemented using the DrScheme programming environment (Findler *et al.*, 2002), since Slideshow is a library for PLT Scheme. All of DrScheme’s programming tools work as usual, including the on-the-fly syntax colorer, the syntax checker, the debugger, and the static analyzer. Non-textual syntax can be used in a Slideshow program, such as a test-case boxes, comment boxes, or XML boxes (which support literal XML) (Clements *et al.*, 2004). More importantly, we can use DrScheme’s extension interface to add new tools to DrScheme that specifically support slide creation.

5.1 REPL Picts

In support of interactive slide development, DrScheme’s read-eval-print-loop (REPL) prints pict as they are drawn in a slide. Figure 2, for example, shows a call to the function `main-characters` and the resulting pict.

Each pict result in DrScheme’s REPL is itself interactive. When the programmer moves

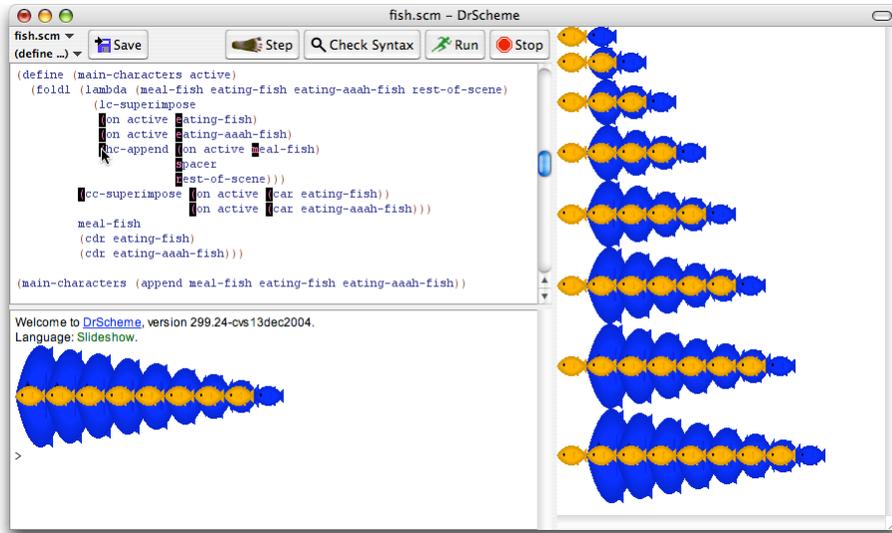


Fig. 3. Mousing over expressions to see resulting pics

the mouse pointer over a pict, DrScheme shows sub-pict bounding boxes that enclose the pointer. In the figure, the narrow bounding box corresponds to the blue fish under the pointer, and the wide bounding box corresponds to an intermediate pict from `hc-append`.

By default, DrScheme does not show bounding boxes that contain other bounding boxes around the pointer (so the visible bounding boxes are minimal), but pressing the `Alt` (or `Meta`) key exposes all of the bounding boxes that enclose the pointer. Pressing the `Control` key shows all bounding boxes in the entire pict. Pressing the `Shift` key toggles the color of each bounding box between black and white.

5.2 Tracing Picts

Figure 3 displays a screen dump for another tool. This tool records all pict(s) that are generated during the run of a Slideshow program. It then highlights each expression that produced a pict by drawing a black box at the start of the expression. A programmer can move the mouse over a box to see the pict(s) generated by the expression.⁶

In the screen dump, the mouse is over the start of an `hc-append` call. The right-hand side of the window shows one pict for each time that the expression was evaluated during the program run. Specifically, the first pict shows the result of the first evaluation, where `rest-of-scene` contains only one eating fish (including its `aaah` variant, superimposed), so the result of `hc-append` contains two fish. As the program continued, this result was further extended with an eating fish, and then it became `rest-of-scene` for

⁶ The `lc-superimpose` call in the third line has no black box because it is in tail position. To preserve tail-calls (Steele, 1977; Clinger, 1998), our prototype tool does not record the values of expressions that are syntactically in tail position.

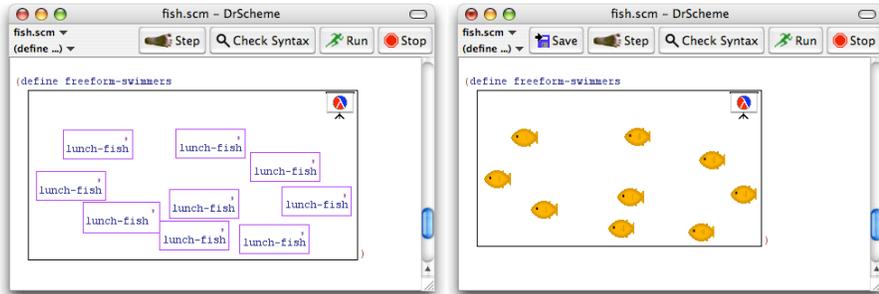


Fig. 4. A pict box containing Scheme boxes

the evaluation of the `hc-append` expression, and so on. Overall, the expression under the mouse is evaluated eight times when the program is run, so eight results are shown for the expression.

5.3 WYSIWYG Picts

The left-hand side of figure 4 shows a program that uses WYSIWYG layout for a composite pict. The box with a projector-screen icon in its top-right corner is a *pict box*. Inside the pict box, each box with a top-right comma (suggestive of `unquote`) is a *Scheme box*. A programmer can create any number of Scheme boxes within a pict box, and then drag the Scheme boxes to position them within the pict box.

A pict box is an expression in graphical syntax, and its result is always a pict value. When the pict box is evaluated, its Scheme-box expressions are evaluated to obtain sub-picts, and these sub-picts are combined using the relative positions of Scheme boxes in the source pict box. In this example, the program defines `freeform-swimmers` as a pict that contains a school of manually positioned fish.

The right-hand side of figure 4 shows the same program, but after the programmer has toggled the box to preview mode. In preview mode, each Scheme box is replaced with the image of the pict that resulted from the most recent evaluation of the Scheme box. (Preview mode is not available until after the program has been run once.) In this way, a programmer can arrange the content of a pict box using actual images, instead of just Scheme expressions.

Pict boxes illustrate how a programming environment can provide WYSIWYG tools that complement the language's abstraction capabilities. A pict box can be placed in any expression position, and Scheme boxes are lexically scoped within the pict-box expression. For example, a pict box can be placed into a function, and then a Scheme box within the pict box can access arguments of the function. In this way, a programmer can use both WYSIWYG editing and programmatic construction of pict, instead of choosing one or the other.

6 Design Choices and Related Work

Slideshow is by no means the first language designed for drawing pictures, nor is it the first environment for programmatic creation of slides. Many of our design choices for Slideshow are best explained through comparisons to other systems.

6.1 Picture Constructs

Slideshow’s definition of a `pict`—a drawing procedure, a bounding box, and coordinates for sub-picts—closely resembles aspects of in Henderson’s functional pictures (Henderson, 1982), PostScript (Adobe Systems Incorporated, 1999), MLgraph (Chailloux *et al.*, 1997), Pictures (Finne & Peyton Jones, 1995), Functional PostScript (Shivers & Sae-Tan, 2004), FPIC (Kamin & Hyatt, 1997), `pic` (Kernighan, 1991), MetaPost (Hobby, 1992), and other systems. Compared to Slideshow, the picture operators of MLgraph, Pictures, Functional PostScript, and FPIC include a richer set of transformation operations (mainly because they all build on PostScript), while Slideshow provides a richer set of text-formatting operations. Slideshow’s combination of the `-find` operations, `ghost`, and `launder` appears to be unique. These operations reflect generally the way that Slideshow is tailored for slide creation, and specifically how Slideshow supports a design recipe for slide sequences.

Among the above systems, the primitives for creating the drawing procedure differ, but in all cases, drawing sizes and locations are represented by concrete numbers. T_EX (Knuth, 1990), IDEAL (Van Wyk, 1981), and Juno-2 (Heydon & Nelson, 1994), in contrast, define pictures in terms of constraints. In T_EX, `glue` allows a box to grow or shrink, depending on the box’s context in the final picture. In IDEAL, programmers define pictures by describing constraints, such as “arrow X’s endpoint is attached to box Y’s right edge.” Thus, picture sizes and locations are represented as constraints instead of as numbers.

We have opted for a direct functional style in Slideshow, instead of a constraint-based language, because we find that many patterns of constraints are easily covered by basic combinators (such as `v1-append`), while other patterns of constraints (like adding lines to connect nodes) are easily abstracted into new functions.

Constraints, however, support certain operations that concrete sizes do not. In particular, a `pict`’s font cannot be changed externally in Slideshow, because changing the font could change the bounding boxes of sub-picts non-uniformly, which would invalidate computations based on those bounding boxes’ dimensions. In a constraint-based system, such invalidated computations can simply be re-executed. In Slideshow and other systems the offsets are computed by arbitrary code, so that automatic re-calculation is not always possible.

Functional reactive programming (FRP) (Elliott & Hudak, 1997) offers an attractive alternative to both constraints and fixed-size attributes, and an interesting evolutionary path from Slideshow’s current design. With FRP, a picture’s attributes can be expressed as number-valued *signals* instead of plain numbers. FRP systems provide variants of arithmetic operations that consume and produce number-valued signals instead of plain numbers, so programs in an FRP-based Slideshow would have the same intuitive form as programs in the current version of Slideshow. We hope to explore this possibility for future versions of Slideshow using FrTime (Cooper & Krishnamurthi, 2004).

6.2 Programming and Presentation Environment

Unlike $\text{T}_{\text{E}}\text{X}$, `pic`, MetaPost (but like `MLgraph`, etc.), Slideshow builds on a general-purpose programming language, so it can support modular development, it allows programmers to write maintainable code, libraries, and tests, and it is supported by a programming environment.

The environment most similar to Slideshow is Slithy (Zongker, 2003), which is a Python-based tool for building slide presentations. Slithy provides a programmatic interface for slide construction, manages the display of slides directly, and supports an explicit set of design principles for presentation authors.

Skribe (Seranno & Gallezio, 2005; Seranno & Gallezio, 2002) is like Slideshow in that it builds on Scheme to support programmatic document creation in a general-purpose functional language. Skribe’s architecture targets mainly the creation of articles, books, and web pages. Since Skribe includes a $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ -output engine, appropriate bindings could be added to Skribe to create slides through $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ -based packages, or a new back-end could be developed for Skribe to target other display applications.

Countless packages exist for describing slides with an HTML-like notation. Such packages typically concentrate on arranging text, and pictures are imported from other sources. Countless additional packages exist for creating slides with $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$, including `foiltex` and `Prosper` (Van Zandt, 2005). With these packages, $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ remains well adapted for presenting math formulae and blocks of text, but not for constructing pictures, and not for implementing and maintaining abstractions.

Unlike Slideshow, most slide-presentation systems (including all based on PostScript, PDF, and $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$) treat the slide viewer as an external tool. Separating the viewer from the slide-generation language makes display-specific customization more difficult, and it inhibits the integration with a programming environment described in section 5. An integrated viewer, meanwhile, can easily support executable code that is embedded within slides. Embedded code is particularly useful in a presentation about programming or about a software system, since an “eval” hyperlink can be inserted into any slide. More generally, Slideshow gives the presentation creator access to the complete PLT Scheme GUI toolbox, which the programmer can use to enrich a presentation with interactive graphical elements.

6.3 Animation

Slithy (Zongker, 2003) is designed specifically to support movie-like animation, whereas Slideshow mainly supports slide-stepping animation. Slithy’s slide-construction combinators seem more primitive than Slideshow’s; most of the work to implement a Slithy presentation is in building animations directly, much like building pictures with Slideshow’s `dc` primitive. A `pict`-like layer of abstraction would support our design recipe in Slithy, but more work is required to adapt the `pict` constructs to fully support movie animation. In particular, key points and times, as in SVG animation (World Wide Web Consortium, 2003), may provide in the time dimension something like of sub-`pict` location in the space dimension. Naturally, FRP is a promising direction for animation support in Slideshow.

Somewhat like our design recipe, the Slithy authors have identified principles for slide animation (Zongker & Salesin, 2003) to guide the design of presentations with anima-

tions. Whereas our design recipe focuses on the steps required to implement a maintainable scene, their principles are more general, and less tied to the task of implementation.

7 Conclusion

In only the last few years, laptop-projected slides have become the standard vehicle for delivering talks, and tools other than PowerPoint are still catching up. We offer Slideshow as a remedy to PowerPoint's lack of abstraction, HTML's lack of flexibility, and \LaTeX 's lack of maintainability. We also offer a design recipe for slide sequences that is supported by Slideshow's set of primitives.

Programmatic construction of pictures and slides is probably not for everyone, even with powerful programming-environment tools. For various reasons, many people will prefer to create pictures and slides in PowerPoint and without significant abstraction, no matter how nice the language of picture construction.

For the authors' tastes and purposes, however, programmatic construction works well, and we believe that it appeals to many programmers. In our slides, with proper code abstraction, we can quickly experiment with different configurations of a picture, add slide-by-slide animation, and evolve ever more general libraries to use in constructing talks. Many tasks can be automated entirely, such as typesetting code and animating reduction sequences. As others begin to use Slideshow, we look forward to contributed libraries for typesetting mathematical formulae and plotting charts.

All of the figures in this paper are generated by Slideshow's `pict` library, using exactly the code as shown.⁷ In fact, like many other picture languages, the core `pict` language works equally well on paper and on slides. A picture language alone is not enough, however; most of our effort behind Slideshow was in finding appropriate constructs for describing, staging, and rendering slides.

For further information on using Slideshow and for sample slide sets—including slides for conference talks and two courses—see the following web page:

<http://www.plt-scheme.org/software/slideshow/>

References

- Adobe Systems Incorporated. (1999). *PostScript Language Reference*. Third edn. Addison-Wesley.
- Chailloux, Emmanuel, Cousineau, Guy, & Suárez, Ascánder. (1997). *The MLgraph System*.
- Clements, John, Felleisen, Matthias, Findler, Robert Bruce, Flatt, Matthew, & Krishnamurthi, Shriram. (2004). Fostering little languages. *Dr. Dobbs' Journal*, Mar., 16–24.
- Clinger, William D. (1998). Proper tail recursion and space efficiency. *Pages 174–185 of: Proc. ACM Conference on Programming Language Design and Implementation*.
- Cooper, Gregory, & Krishnamurthi, Shriram. (2004). *FrTime: Functional reactive programming in PLT Scheme*. Computer science technical report. Brown University. CS-03-20.
- Elliott, Conal, & Hudak, Paul. (1997). Functional reactive animation. *Pages 263–273 of: Proc. ACM International Conference on Functional Programming*.

⁷ We used Slideshow version 299.107, and we re-defined the `slide` operations to produce boxed `picts`.

- Felleisen, Matthias, Findler, Robert Bruce, Flatt, Matthew, & Krishnamurthi, Shriram. (2001). *How to Design Programs*. Cambridge, Massachusetts: The MIT Press. <http://www.htdp.org/>.
- Findler, Robert Bruce, & Flatt, Matthew. 2004 (Sept.). Slideshow: Functional presentations. *Pages 224–235 of: Proc. ACM International Conference on Functional Programming*.
- Findler, Robert Bruce, Flatt, Matthew, Felleisen, Matthias, & Krishnamurthi, Shriram. (2002). The design and implementation of DrScheme. *Journal of Functional Programming*, **12**(2), 159–182.
- Finne, Sigbjorn, & Peyton Jones, Simon. 1995 (July). Pictures: A simple structured graphics model. *Proc. Glasgow Functional Programming Workshop*.
- Henderson, Peter. (1982). Functional geometry. *Pages 179–187 of: Proc. ACM Conference on Lisp and Functional Programming*.
- Heydon, Allan, & Nelson, Greg. (1994). *The Juno-2 constraint-based drawing editor*. SRC research report. Systems Research Center. 131a.
- Hobby, John D. (1992). *A User's Manual for MetaPost*. Computer science technical report. AT&T Bell Laboratories. CSTR-162.
- Kamin, Samuel N., & Hyatt, David. 1997 (Oct.). A special-purpose language for picture-drawing. *Pages 297–310 of: Proc. USENIX Conference on Domain-Specific Languages*.
- Kernighan, Brian W. (1991). *PIC — a graphics language for typesetting, user manual*. Computer science technical report. AT&T Bell Laboratories. CSTR-116.
- Knuth, D. E. (1990). *The TeX-book (revised)*. Addison Wesley.
- PLT. (2005). *PLT Scheme*. www.plt-scheme.org.
- Seranno, Manuel, & Gallezio, Erick. 2002 (Oct.). This is Scribe! *Pages 31–40 of: Proc. Workshop on Scheme and Functional Programming*.
- Seranno, Manuel, & Gallezio, Erick. (2005). *Scribe home page*. <http://www.inria.fr/mimosafp/Scribe>.
- Shivers, Olin, & Sae-Tan, Wendy. (2004). *Functional PostScript: Industrial-strength 2D functional imaging*. In preparation. <http://www.scs.hawaii.edu/resources/fps.html>.
- Steele, Guy Lewis. (1977). Debunking the “expensive procedure call” myth, or procedure call implementations considered harmful, or LAMBDA, the ultimate GOTO. *Pages 153–162 of: Proc. ACM Conference*.
- Tufte, Edward R. (2003). *The Cognitive Style of Powerpoint*. Graphics Press.
- Van Wyk, Christopher J. (1981). *IDEAL user's manual*. Computer science technical report. AT&T Bell Laboratories. CSTR-103.
- Van Zandt, Timothy. (2005). *Prosper*. prosper.sourceforge.net.
- World Wide Web Consortium. (2003). *Scalable vector graphics (SVG) 1.1*.
- Zongker, Douglas. (2003). *Creating animation for presentations*. Ph.D. thesis, University of Washington.
- Zongker, Douglas, & Salesin, David. (2003). On creating animated presentations. *Proc. Eurographics/SIGGRAPH Symposium on Computer Animation*.