Dynapad: A Brief Introduction

James D. Hollan

Distributed Cognition and HCI Laboratory
Department of Cognitive Science
University of California, San Diego

Lab Website: hci.ucsd.edu
Email: hollan@cogsci.ucsd.edu

Dynapad Website: hci.ucsd.edu:dynapad
Dynapad Email: dynapad@hci.ucsd.edu

1 Introduction

Dynapad[1] is the third generation of our multiscale interface and visualization software. It makes scale a first-class parameter of objects, supports navigation of multiscale workspaces, and provides special mechanisms to maintain interactivity while rendering large numbers of graphical items. Dynapad employs Scheme to provide a high-level programming interface to the multiscale graphical and interaction facilities in the C++ rendering substrate.

Dynapad implements multiscale graphical objects (e.g., rectangles, lines, text, and images) that are interactive (e.g., they can be scaled or moved via user interaction) and dynamic (e.g., they can have behaviors that result from running of attached code). Behaviors can be associated with an object, a class of objects, or a region of the workspace and are triggered by user actions, the behavior of other objects, events, or timer interrupts. Because Dynapad is based on a class system, objects in Dynapad can inherit characteristics and behaviors from other objects.

To create a Dynapad workspace simply type (*load* "workspace.ss") in Scheme. This assumes, of course, that both the version of Scheme we are using, PLT-Scheme[2], and Dynapad have been properly installed. The default workspace, *dynapad* is an instance of the **dynapad%**[3] class. You can send messages to

[1] The name *Dynapad* was chosen to reflect the software's heritage from our earlier Pad++ and STkPad software as well as ideas from Dynabook and Sketchpad.

[2] http://www.plt-scheme.org

[3] The convention is to end class names with a percent sign.

this instance to see or change various characteristics of a dynapad instance. For example, the following code sets the background color to be *red*, the default *font* to Haeberli, and the default *pencolor* to *black* with *width* set to 3:

```
(define (my-defaults)
  (send dynapad background "red")
  (send dynapad defaultfont "Haeberli")
  (send dynapad defaultpen "black")
  (send dynapad defaultpenwidth 3))
```

Also any of these messages without the value will return the current setting. For example, (**send** *dynapad* **defaultfont**) will return the current setting of the default font and (**send** *dynapad* **fontnames**) will list the names of all the available fonts[4].

Two Dynapad windows are depicted in Figures 1 and 2. They provide example views onto a multiscale workspace. Using the mouse, a view can be panned to any workspace location and zoomed to any scale. Similarly, a selected object or group of objects can be moved or scaled using the mouse. Locations of objects on a Dynapad worksurface are specified by an X-Y coordinate system. Similarly, a view has a location and scale. A view provides visual access to a portion of the worksurface. Its X-Y location indicates the center of the view and its scale specifies a magnification.

Every Dynapad object is located at an X-Y position and has a scale. X-coordinates increase to the right and Y-coordinates increase upward. Coordinates are stored as floating-point numbers and are independent of the current view. For example, if the view happens to have a magnification of 2.0 and you create an object in the view that is 50 pixels wide, it will appear 100 pixels wide within the view.

From Scheme there are messages to position and scale the view. To move the view to a particular location use **moveto**. For example, (**send** *dynapad* **moveto** '(0 0 1)) moves the center of the view to the default home location of the worksurface: x location 0, y location 0, and zoom of 1. To find the current view location, send a dynapad instance a **view** message: (**send** *dynapad* **view**). To determine the bounding box of the view: (**send** *dynapad* **bbox**). A bounding box[5] is represented as a list. The first two numbers are the x and y coordinates of the bottom left corner of the view and the last two numbers are the x and y coordinates of the top right corner.

[4] Dynapad supports any Type 1 Postscript font.

[5] Dynapad provides a set of routines to assist with bounding box computations. Eleven functions give access to the corners of a bounding box by compass locations, the center of the box, the centers of each side, and the height and width of the box. For example, *(bbsw bbox)* provides the coordinates of the southwest corner of the box. *(bbn bbox)* gives the coordinates of the center of the top of the bounding box. The functions are: bbsw, bbw, bbnw, bbn, bbne, bbe, bbse, bbs, bbcenter, bbheight, bbwidth.

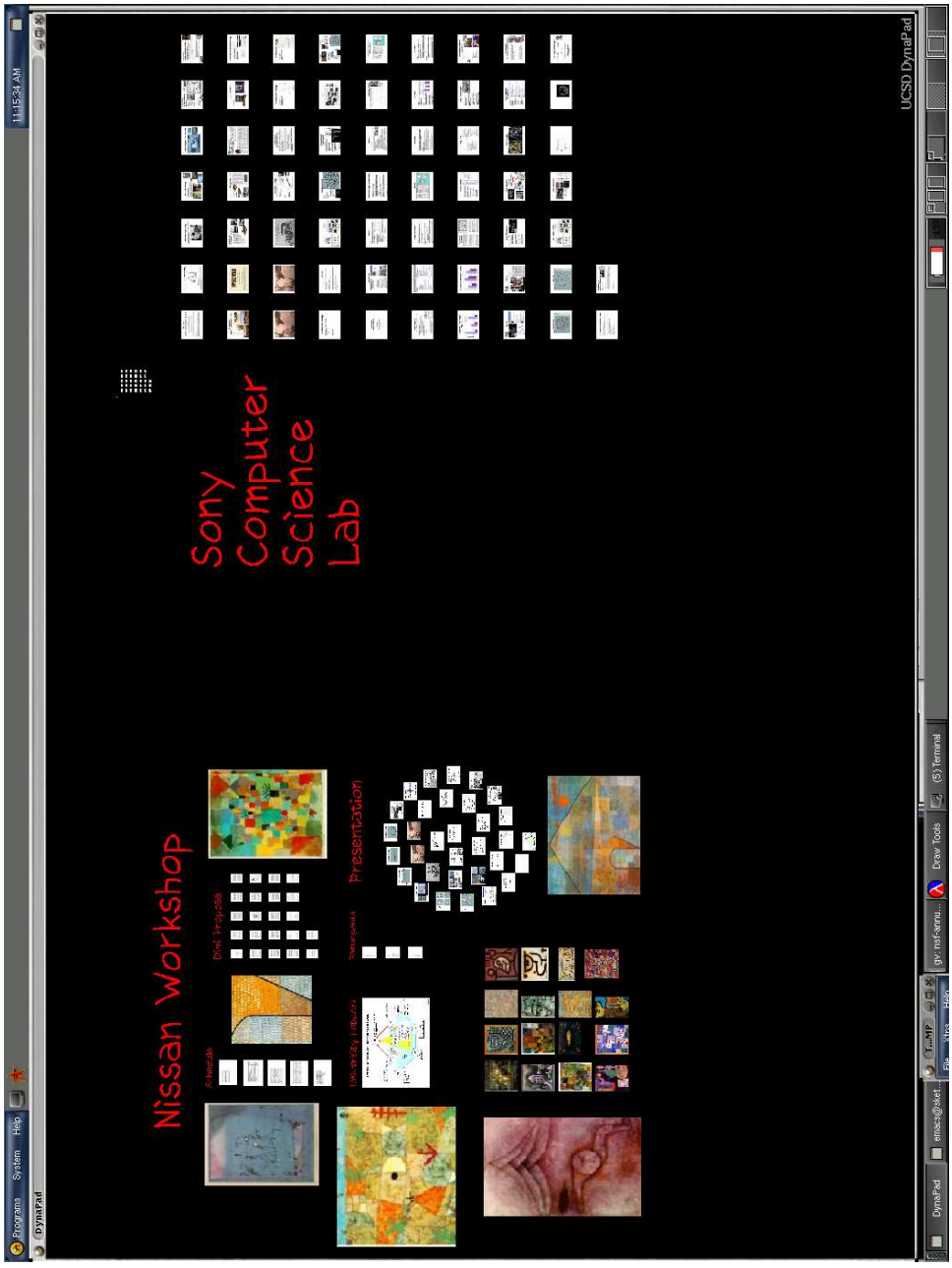

Figure 1: Example Dynapad Window. This shows a series of talks I recently gave in Japan. The spiral on the left, for example, is a powerpoint-like talk that zooms in and animates between slides.

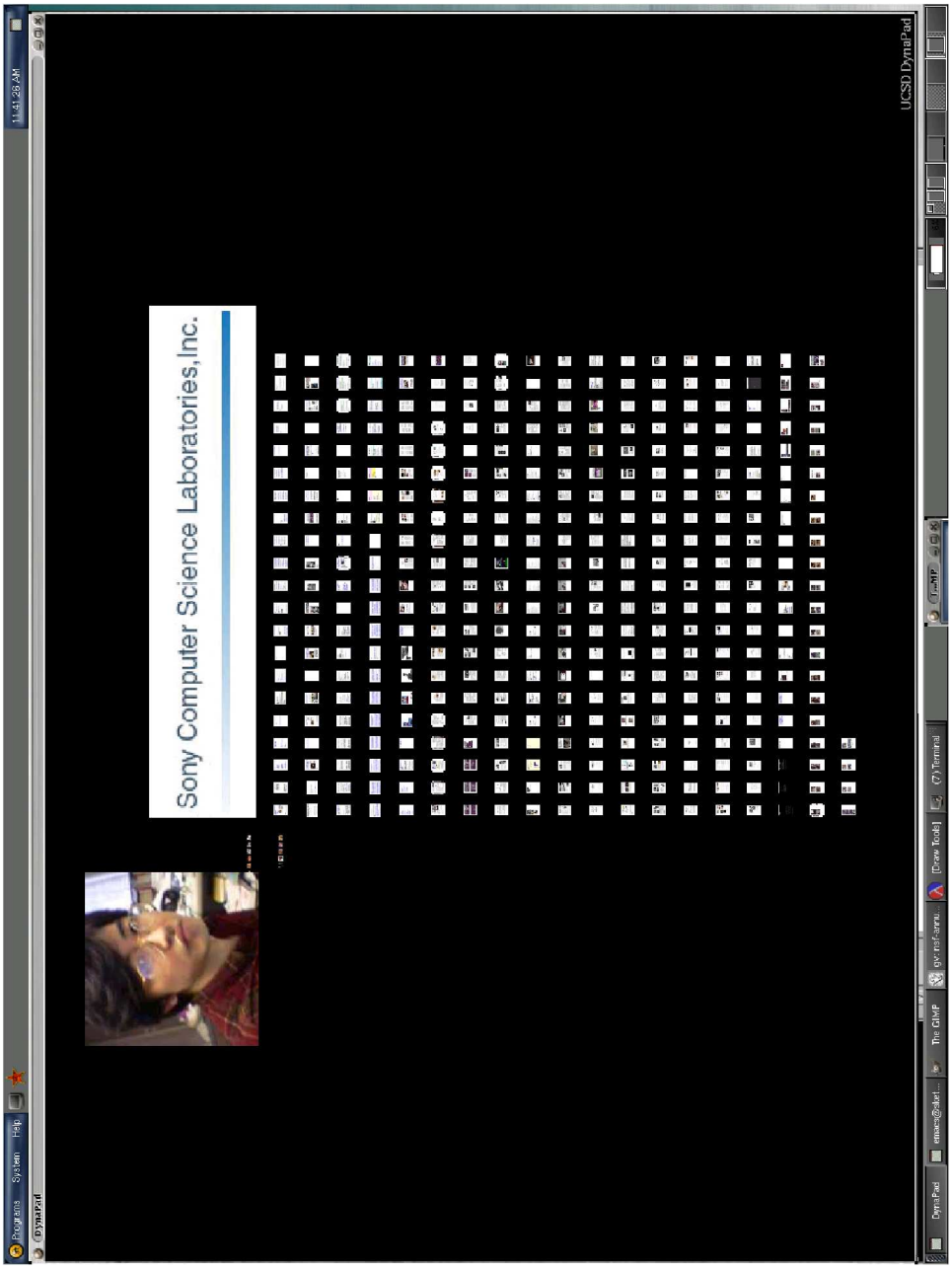

Figure 2: Zoomed View of a Portion of Previous Dynapad Window. This is zoomed in on what is only a few pixels in the Sony Computer Science area from the previous figure. It shows an image of Jun Rekimoto. The large collection of images at the bottom is a multiscale view of his whole web site.

A view can be zoomed by a *zoomfactor*[6]. For example, (**send** *dynapad* **zoom** 2) will zoom in, centered on the current view, by a factor of two. To return to the previous view requires halving the current zoom by zooming out by a factor of .5: (**send** *dynapad* **zoom** .5). An optional animation time can be included. (**send** *dynapad* **zoom** 2 1000) takes 1 second to zoom in by a factor of two. One also can specify a location to center the zoom around. Without such a specification, zooming is always centered on the current view. The message (**send** *dynapad* **zoom** 2 3000 10 10) will result in taking 3 seconds to zoom by a factor of two toward location (10, 10).

One can center the view on an object or list of objects. If the value of *objs* is a list of dynapad objects, (**send** *dynapad* **center** *objs*) will center the view so that all the objects referred to by *objs* are within the view. The scale of the objects will determine if they are visible. One can also control the time to animate to a given location. For example, (**send** *dynapad* **center** *objs* 2000) will take 2 seconds to animate to a location that centers the view on the set of objects in the *objs* list. One can also specify centering to be done in two steps. With a two-step animation[7], the view zooms out so that both the start point and end point of the animation are visible and then zooms to the endpoint.

One can also use two-step with **moveto**. In the following code, the view will first move using the two-step technique to x-y location (100, 200) at a zoom of 10 and then center on an object *foobar* also using a two-step technique. Each move of the view will be done over 1 second.

```
(send dynapad moveto '(100 200 10) 1000 #t)
(send dynapad center foobar 1000 #t)
```

Two frequently used messages to a dynapad instance are **objects** and **selected**. The former returns a list of all the objects in the workspace and the latter returns a list of the currently selected objects. The following function finds all the objects on a worksurface and then centers each of the objects in the view for 3 seconds and takes two seconds to animate between objects:

```
(define (view-all)
  (let ((objs (send dynapad objects)))
    (for-each (lambda (obj)
                (send dynapad center obj 2000)
                (sleep 3))
              objs)))
```

One can also send a center message to a dynapad object. In the case of a thumbnail image, centering also replaces the thumbnail with the high-resolution image from which the thumbnail was derived. If the images in the previous

[6]Note that a *zoomfactor* is a multiplicative factor rather than an absolute setting.

[7]A two-step animation uses a slow-out and slow-in technique.

example are thumbnails, changing the code to send the center message to the object rather than the dynapad instance will result in each thumbnail image being replaced by its high-resolution version as it is centered:

```
(define (view-all)
  (let ((objs (send dynapad objects)))
    (for-each (lambda (obj)
                (send obj center 2000)
                (sleep 3))
              objs)))
```

Exchanging high-resolution and thumbnail images makes extremely efficient use of memory and enables workspaces with very large numbers of images.

As mentioned earlier there are also messages to scale and position Dynapad objects. For example, if *image1* is a dynapad object, (**send** *image1* **position**) will return the position of *image1* and (**send** *image1* **position** '(100 100 2)) will move the object to location (100, 100) with a scale of 2. (**send** *image1 scale* 3) will change its scale by a factor of 3 from the current scale. A **slide** message can be used to slide an object from its current location by some number of pixels. (**send** *image1* **slide** 10 20) will slide the images 10 pixels to the left and 20 pixels up. In the next section, we introduce some details about PLT-Scheme classes.

2 Dynapad Classes

Objects in Dynapad are defined using the PLT-Scheme class system. A class specifies a collection of fields (sometimes called *instance variables*) and messages (sometimes called *methods*). Each class is defined in terms of a *superclass*, from which it inherits fields and messages. The base class for all PLT-Scheme classes is *object%*. The base class for Dynapad objects is **dynaobject%**. It is at the root class of the Dynapad inheritance hierarchy and provides messages such as **center** to all instances of the class.

The functions **make-object** and **instantiate-object** are used to make instances of a class. For example, (**define** *don* (**make-object image%** *dynapad* "images/knuth.gif")) will load an image of Don Knuth) at the default home location of (0 0 1). To load it at another location, say (100 200 2), (**make-object image%** *dynapad* "images/knuth.gif" '(100 200 2)).

One can also *override* an inherited message in a class to provide a different behavior for that message. For example, we might want to define a new image class but have instances of the class do something a bit different when sent a **center** message. Suppose we want them to center themselves, zoom up by a factor of 1.5 for a period and then return to the normal centered position.

We can define a new class *myimage%* that has **image%** as its superclass and override the **center** message:

```
(define myimage%
  (class image%
         (init pad hirespath location)
         (override center)
         (define (center)
           (send dynapad center this 1000)
           (send dynapad zoom 1.5 1000)
           (send dynapad center this 1000))
         (super-instantiate (pad hirespath location))))
```

Then we could make an instance of this new class which would have the new center behavior but inherit all the other **image%** messages (e.g., **position**):

```
(define don (make-object myimage% dynapad "images/knuth.gif" '(200 200 2)))
(send don center)
(send don position)
```

One might also want to permit an optional zoom factor and zooming time to be specified:

```
(define myimage%
  (class image%
         (init pad hirespath location)
         (init-field (factor 1.5) (time 1000))
         (override center)
          (define (center)
            (send dynapad center this time)
           (send dynapad zoom factor time)
            (send dynapad center this time))
         (super-instantiate (pad hirespath location))))
```

In the example above we replaced the center method with a new method by overriding it. Sometimes we want to change an inherited method rather than totally replace it. We can accomplish this by renaming the method we inherit from our superclass. This allows continued access to the inherited method after we override it. In the code below, we modify the center message for the **image%** class. We use the inherited method to center the image but add a zoom behavior. When an image is sent **center** it will be centered in the view, then zoom up by the specified factor, and finally return to being centered in the view at the same scale as when it was first centered.

Figure 3: Spiral Layout. Using a list of 376 images, *art-examples*, this view is the result of *(arrange-in-spiral-in-current-view art-examples)*.

```
(define myimage%
  (class image%
          (init pad hirespath location)
          (init-field (factor 1.5) (time 1000))
          (rename (super-center center))
          (override center)
           (define (center)
             (super-center time) ;why this rather than (send this super-center)
             (send dynapad zoom factor time)
             (super-center time))
          (super-instantiate (pad hirespath location))))
```

2.1 Layout Routines

Dynapad provides routines to lay out images in the current view as well as behind or on other objects. Laying out images behind an object creates a mul-

tiscale layout in which zooming into the object reveals the images behind it. The layout functions assume that all images are the same size. Typically images are thumbnails created with their larger dimension set to the same size (commonly 125 bits). The function **create-thumbs** is available to create thumbnails. For example, (**create-thumbs** "/usr/tmp/images" 125) will create thumbnails for the images in the */usr/tmp/images/thumbs/125* directory. Figure 3 depicts a spiral layout.

The following layout functions, among others, are available:

arrange-in-grid-behind-object	**arrange-in-grid-in-current-view**
arrange-in-grid-onto-object	**arrange-in-spiral-behind-object**
arrange-in-spiral-in-current-view	**arrange-in-spiral-onto-object**
arrange-in-pile-in-current-view	

Below is example code to recursively descend a directory and draw subdirectories as nested boxes laid out in a grid. Each directory is labeled with its directory path and all images contained within it are laid out as a grid within the associated box.

```
(define (draw-subdirectories directory)
  (let ((save-view (send dynapad view)))
    (draw-subdirectories-helper directory)
    (send dynapad moveto
          (list (car save-view) (cadr save-view) (* .75 (caddr save-view))))))
(define (draw-subdirectories-helper directory)
  (let ((box (make-object rect% dynapad (send dynapad bbox)))
        (path (make-object text% dynapad directory))
        (directories (filter
                      (lambda (item)
                        (directory-exists?
                         (build-path directory item)))
                      (directory-list directory)))
        (dlist '())
        (images (hires-list directory))
        (ilist '()))
    (for-each
     (lambda (image)
       (set! ilist (cons
                    (make-object image% dynapad
                                 (build-path directory image)
                                 (send dynapad view)) ilist)))
     images)
    (for-each
     (lambda (dir)
```

```
    (set! dlist (cons
                  (make-object rect% dynapad
                               (send dynapad bbox)) dlist)))
  directories)
 (if (or (not (null? dlist))
         (not (null? ilist)))
     (arrange-in-grid-onto-object (append dlist ilist) box))
 (send path position
       (list
        (car (send box bbox))
        (cadr (send box bbox))
        (/ (bbwidth (send box bbox)) (bbwidth (send path bbox)))))
 (for-each (lambda (dir box)
             (send box center 1 #f .5 .5 1.0)
             (draw-subdirectories-helper (build-path directory dir)))
           directories dlist)))
```

2.2 Event Bindings

It is easy to associate a function with a dynapad object so that when the user interacts with that item (via a mouse button press or key press) the associated function is evaluated. This is implemented with the *bind* command. For example, the following code creates a red square. Clicking on it will make it move randomly.

```
(define rect1 (make-object rect% '(0 0 50 50)))
(send rect1 fill "red")
(send rect1 bind "<ButtonPress>" (lambda (obj str e)
                                   (send obj slide
                                   (if (b(.pad 'moveto 0 0 2 1000)))
(.pad 'create 'rectangle 0 0 50 50 :tags 'rect1 :fill 'black)
(.pad 'create 'rectangle 100 0 150 50 :tags 'rect2 :fill 'white)
(.pad 'bind 'rect1 "<ButtonPress>" (lambda () (.pad 'moveto 0 0 2 1000)))
(.pad 'bind 'rect2 "<ButtonPress>" (lambda () (.pad 'moveto 0 0 1 1000)))
```

The last number in the **moveto** command indicates that the movement should take a 1000 milliseconds to accomplish.

2.3 Regions

A region is a worksurface area in which a particular physics of behavior is operational. An instance of a **region%** class specifies the boundaries of the

region, the physics that is imposed, and a set of messages to allow control of the physics.

One region class provided in Dynapad is a **layout%** class that controls the arrangement and scale of objects contained within the region.

```
;; Should objects know what dynapad they are on? Should it be an instance ;;
variable?
(define region%
  (class rect%
         (super-instantiate ())))
(define layout%
  (class region%
         (define (arrange type lst)
           (case type
             ((in-grid-in-current-view) (arrange-in-grid-in-current-view lst))
             ((in-pile-in-current-view) (arrange-in-pile-in-current-view lst))
             ((in-spiral-in-current-view) (arrange-in-spiral-in-current-view lst))
             ((in-grid-behind-object) (arrange-in-grid-behind-object lst))
             ((in-grid-onto-object) (arrange-in-grid-onto-object lst))
             ((in-spiral-behind-object) (arrange-in-spiral-behind-object lst))
             ((in-spiral-onto-object) (arrange-in-spiral-onto-object lst))
             (else (error "Unknown message to arrange: " type))))
                 (super-instantiate ())))
```

3 Powerpoint Presentation in Dynapad

In order to load a Powerpoint presentation, you need to save the presentation as images and then create thumbnails of the images. For example, given that the exported images are in the directory */home/hollan/talks/image-browsing*, the following code creates thumbnails[8] and loads them into Dynapad:

```
(create-thumbs "/home/hollan/talks/image-browsing" 125)
(load-and-link-images "/home/hollan/talks/image-browsing")
```

The thumbnails of the slide images are loaded and arranged by default in a grid. On each images three invisible hyperlink areas are created. To traverse the links you need to double click. Clicking on the top left quadrant of a slide will move to the previous slide and center it in the view, clicking on the top right quadrant will move to and center the next slide, and clicking on the bottom half of a slide will center it in the view. When an image is centered the high-resolution version is loaded. It is unloaded when another image is centered. This lets one

[8] In the example, thumbnails are created of size 125 bits on the longest axis.

use high-resolution versions of the slides with minimal impact on Dynapad since only one high-resolution image is in memory at a time.

Here are the relevant code from the file *slides.ss*:

```
;;
;; Load and link powerpoint slides exported as images
(define (load-and-link-images dir)
  (let* ((images (arrangedir dir))
         (first (car images))
         (last-two (link-to-next first images)))
    (fixup-last first last-two)))
(define (fixup-last first last-two)
  (let ((areas (add-click-areas (cadr last-two))))
    (send (car areas) link (car last-two))
    (send (cadr areas) link first)
    (send (cadr last-two) link (cadr last-two))))
(define (link-to-next last l)
  (cond ((null? (cdr l))
         (list last (car l)))
        (else
         (let ((areas (add-click-areas (car l))))
           (send (car areas) link last)
           (send (cadr areas) link (cadr l))
           (send (car l) link (car l)))
         (link-to-next (car l) (cdr l)))))
;;
;; Click-Areas for Slides
(define (add-click-areas obj)
  (let* ((bb (send obj bbox))
         (right (make-object rect% dynapad bb))
         (left (make-object rect% dynapad bb)))
    (send* right (fill "black") (scale .5) (transparency .001) (anchor "sw"))
    (send* left (fill "black") (scale .5) (transparency .001) (anchor "se"))
    (make-object group% dynapad (list right left obj))
    (list left right)))
```