

# Documentation for the world.ss teachpack

The `world.ss` teachpack allows you to create animations. Some of the functions are explained here in detail. Full documentation of these and other functions can be found in the DrScheme Help Desk. More information on the Help Desk can be found on the “Helpful Tips” page of the course Web site.

In the same way that a film is a series of still shots seen very quickly in sequence, here an animation is a series of *scenes*, changing at each “tick” of the “clock”. There are two main ways to make animations, one using an idea called a *world* and the other using a list of scenes. Each program can execute at most one animation at any given time.

## 1 Creating images and scenes

Each scene in an animation is made from one or more images. Images can also be used as still illustrations. Images are formed by overlaying various shapes.

### 1.1 Basic shapes

The following functions produce images with shapes that are either solid (specified as `'solid`) or outlines (specified as `'outline`). In each you can choose a colour (e.g. `'red`, `'yellow`, `'blue`, `'green`, `'orange`, `'purple`, `'white`, `'black`, `'brown`) as well. The sizes given below are in pixels.

The first four functions below create, respectively, a rectangle of the given width and height, a circle of the given radius, an ellipse of the given width and height, and an upward-pointing equilateral triangle of the given edge length, each with the specified mode and colour. The star function creates a star with the specified number of points (at least two), each going from *radius-begin* to *radius-end*.

*(rectangle width height mode colour)*

*(circle radius mode colour)*

*(ellipse width height mode colour)*

*(triangle length mode colour)*

*(star points radius-begin radius-end mode colour)*

For example, the following code will define and then display five images, each with the same width.

```
(define red-rect (rectangle 40 20 'solid 'red))
(define blue-circ (circle 20 'outline 'blue))
(define green-tri (triangle 40 'solid 'green))
(define orange-ellipse (ellipse 40 20 'outline 'orange))
(define yellow-star (star 7 10 20 'solid 'yellow))
red-rect
blue-circ
green-tri
```

*orange-ellipse*  
*yellow-star*

## 1.2 Putting together shapes

We may wish to form a new image by putting one image in front of another. To do so, we need to be able to specify how the images should be aligned with each other. To handle this, each image has a *pinhole*. The pinhole of each shape is in the middle of the shape. The overlay function lines up the given images at their pinholes to form a new image; the last image listed is the one in the front.

```
:: overlay : image image image ... → image
```

Using the shapes we created before, the following lines of code will result in the triangle being in front, the circle being next, and the rectangle being all the way in the back.

```
(overlay red-rect blue-circ green-tri)
```

But what if we wish to align the shapes in a different way? One way is to use (*overlay/xy image1 x y image2*) to place image2 on image1 such that the pinhole of image2 is offset *x* pixels to the right (or left, if *x* is negative) and *y* pixels down (or up, if *y* is negative) of the pinhole of image1. Using the shapes we define above, the function applications

```
(overlay/xy red-rect 40 0 blue-circ)
```

```
(overlay/xy red-rect 0 (- 30) green-tri)
```

form images in which the blue circle is placed to the right of the red rectangle and the green triangle is placed above the red rectangle.

Another option is to create a new image with a pinhole in a different location. The `world.ss` teachpack includes functions that allow us to determine where the pinhole is and to create a new image with the pinhole in a different location. We use a coordinate system to refer to locations; the *x* coordinate is the number of pixels away from the left border of the image (so the *x* coordinate 0 is at the left border), and the *y* coordinate is the number of pixels away from the top border of the image (so the *y* coordinate 0 is at the top border). The location (0, 0) is at the top left corner of the image.

```
:: pinhole-x: image → num
```

```
:: Produces the x coordinate of the pinhole of an image.
```

```
(pinhole-x animage)
```

```
:: pinhole-y: image → num
```

```
:: Produces the y coordinate of the pinhole of an image.
```

```
(pinhole-y animage)
```

```
:: put-pinhole: image num num → image
```

```
:: Produces a new image with the pinhole at
```

```
:: (xcoord, ycoord).
```

*(put-pinhole animage xcoord ycoord)*

;; move-pinhole: image num num  $\rightarrow$  image  
;; Produces a new image with the pinhole moved right (or left  
;; if the number is negative) and down (or up if the number  
;; is negative) by the xcoord and ycoord, respectively, from  
;; positions in animage.

*(move-pinhole animage xcoord ycoord)*

;; image-width: image  $\rightarrow$  num  
;; Produces the width of animage, in pixels.

*(image-width animage)*

;; image-height: image  $\rightarrow$  num  
;; Produces the height of animage, in pixels.

*(image-height animage)*

### 1.3 Useful functions for scenes for animations

An animation can be made with any images, but due to the placement of the pinhole, it is often more convenient to think of forming a series of rectangular scenes, each of which is just a special type of image.

Often the animation will start out with an empty scene. This can be created using the function *empty-scene*, where

*(empty-scene width height)*

creates a blank rectangle of dimensions  $width \times height$ , placing the pinhole in the top left corner, that is, position  $(0, 0)$ .

To use an image in a scene, the function application

*(place-image image x y scene)*

creates the scene formed by placing the supplied image with its pinhole at position  $(x, y)$  in the supplied scene.

## 2 Creating animations using worlds

In order to create an animation, you will need to provide a way of changing one world into another (where your worlds can be numbers, symbols, or more complicated entities such as structures or lists) and a way of generating a scene from a world. For example, if your world is a number, you can change the world by increasing it at each tick, and you can generate a circle using the world as the radius. You also need a way to start the clock ticking and a way to bring the animation to an end.

## 2.1 Starting the animation

The animation starts with a call to *big-bang*, in which you can specify the size of the “canvas” on which images will be drawn, the speed of the clock, and the first world. The function application (*big-bang width height ticklength firstworld*)

prepares to draw scenes of dimensions  $width \times height$ , creates a clock that ticks every *ticklength* seconds, and makes *firstworld* the first world. It evaluates to *true*, which will appear in your Interactions window.

For example, if our worlds are numbers, we might start the animation as:

(*big-bang* 100 100 (/ 1 25) 1)

## 2.2 Changing worlds

The function application

(*on-tick-event world-to-world-fun*)

specifies that at each clock tick, the function *world-to-world-fun* is applied to the current world to produce the next world. The function *on-tick-event* evaluates to *true*.

For example, if the world is a number, the function *advance-world* might be defined to increase the number by 1.

```
;; advance-world: num  $\rightarrow$  num
;; Produces one greater than nbr.
;; Examples: (advance-world 2)  $\Rightarrow$  3
;; (advance-world 3)  $\Rightarrow$  4
(define (advance-world nbr)
  (+ 1 nbr))
```

To stop the animation, the function application (*stop-when last-world?*) is used to stop the clock. This means that it is necessary to write a function *last-world?* that consumes a world and produces true if the world is the last one to be displayed. For example, if we wish to stop when the number reaches 50, we would have the following function:

```
(define (last-world? num)
  (equal? num 50))
```

## 2.3 Translating a world into a scene

The function application

(*on-redraw world-to-scene-fun*)

is used to tell DrScheme to apply the function *world-to-scene-fun* at each tick; it evaluates to *true*. You will specify a function that translates a world into a scene. For example, if the world is a

number, the following function can be used to convert that number into a drawing of a red circle of radius world:

```
(define (num-to-circle num)
  (place-image (circle num 'outline 'red) 50 50
    (empty-scene 100 100)))
```

The example given throughout this section is summarized below; it creates a circle that increases in radius until it fills the scene. It uses the functions *advance-world*, *num-to-circle*, and *last-world?* defined above.

```
(big-bang 100 100 (/ 1 25) 1)
(on-tick-event advance-world)
(on-redraw num-to-circle)
(stop-when last-world?)
```

### 3 Creating an animation using lists of scenes

Another option is to create a list of images and then to use (*run-movie list-of-images*) on the list of images to obtain an animation. It is not necessary to use scenes instead of general images, but for this to work successfully, the images should be the same size and should each have their pinhole in position (0, 0).

The following incompletely-documented example shows how to animate the growing circles in a different way.

```
;; make-circle: num → image
;; Produces the image of a circle of radius num.
(define (make-circle num)
  (place-image (circle num 'outline 'red) 50 50 (empty-scene 100
    100)))
```

```
;; make-circle-list: num num → (listof image)
;; Produces a list of images of circles of radius num to maxsize.
(define (make-circle-list num maxsize)
  (cond
    [(= num maxsize) empty]
    [else (cons (make-circle num) (make-circle-list (+ 1 num)
      maxsize))]))
```

```
;; We can create a movie from the list of circles by the following:
(run-movie (make-circle-list 1 50))
```