HtD Worlds

Help

The How to Design Worlds process provides guidance for designing interactive world programs using `big-bang`. While some elements of the process are tailored to `big-bang`, the process can also be adapted to the design of other interactive programs. The wish-list technique can be used in any multi-function program.

World program design is divided into two phases, each of which has sub-parts:

1. Domain analysis (use a piece of paper!)
    1. Sketch program scenarios
    2. Identify constant information
    3. Identify changing information
    4. Identify big-bang options
2. Build the actual program
    1. Constants (based on 1.2 above)
    2. Data definitions (based on 1.3 above)
    3. Functions
        1. main first (based on 1.3, 1.4 and 2.2 above)
        2. wish list entriesfor big-bang handlers
    4. Work through wish list until done

Phase 1: Domain Analysis

Do a domain analysis by hand-drawing three or more pictures of what the world program will look like at different stages when it is running.

Use this picture to identify constant information such as the height and width of screen, color of the background, the background image itself, the length of a firework's fuse, the image for a moving cat and so on.

Also identify changing information such as the position of a firework, the color of a light, the number in countdown etc.

Identify which `big-bang` options the program needs.

If your program needs to:	**Then it needs this option:**
change as time goes by (nearly all do)	`on-tick`
display something (nearly all do)	`to-draw`
change in response to key presses	`on-key`
change in response to mouse activity	`on-mouse`
stop automatically	`stop-when`

(There are several more options to `big-bang`. Look in the DrRacket help desk under `big-bang` for a complete list.)

Phase 2: Building the actual program

Structure the actual program in four parts:

1. Requires followed by one line summary of program's behavior
2. Constants
3. Data definitions
4. Functions

The program should begin with whatever require declarations are required. For a program using `big-bang` this is usually a require for `2htdp/universe` to get `big-bang` itself and a require for `2htdp/image` to get useful image primitives. This is followed by a short summary of the program's behavior (ideally 1 line).

The next section of the file should define constants. These will typically come directly from the domain analysis.

This is followed by data definitions. The data definitions describe how the world state — the changing information identified during the analysis — will be represented as data in the program. Simple world programs may have just a single data definition. More complex world programs have a number of data definitions.

The functions section should begin with the `main` function which uses `big-bang` with the appropriate options identified during the analysis. After that put the more important functions first followed by the less important helpers. Keep groups of closely related functions together.

Template for a World Program

A useful template for a world program, including a template for the main function and wish list entries for tick-handler and to-draw handler is as follows. To use this template replace WS with the appropriate type for your changing world state. You may want to give the handler functions more descriptive names and you should definitely give them all a more descriptive purpose.

```
(require 2htdp/image)
(require 2htdp/universe)

;; My world program  (make this more specific)

;; =================
;; Constants:



;; =================
;; Data definitions:

;; WS is ... (give WS a better name)

;; =================
;; Functions:

;; WS -> WS
;; start the world with ...
;;
(define (main ws)
  (big-bang ws                   ; WS
            (on-tick   tock)     ; WS -> WS
            (to-draw   render)   ; WS -> Image
            (stop-when ...)      ; WS -> Boolean
            (on-mouse  ...)      ; WS Integer Integer MouseEvent -> WS
            (on-key    ...)))    ; WS KeyEvent -> WS

;; WS -> WS
;; produce the next ...
;; !!!
(define (tock ws) ...)



;; WS -> Image
;; render ...
;; !!!
(define (render ws) ...)
```

Depending on which other big-bang options you are using you would also end up with wish list entries for those handlers. So, at an early stage a world program might look like this:

```
(require 2htdp/universe)
(require 2htdp/image)

;; A cat that walks across the screen.

;; Constants:

(define WIDTH  200)
(define HEIGHT 200)

(define CAT-IMG (circle 10 "solid" "red")) ; a not very attractive cat

(define MTS (empty-scene WIDTH HEIGHT))
```

```
;; Data definitions:

;; Cat is Number
;; interp. x coordinate of cat (in screen coordinates)
(define C1 1)
(define C2 30)

#;
(define (fn-for-cat c)
  (... c))



;; Functions:

;; Cat -> Cat
;; start the world with initial state c, for example: (main 0)
(define (main c)
  (big-bang c                         ; Cat
            (on-tick   tock)          ; Cat -> Cat
            (to-draw   render)))      ; Cat -> Image

;; Cat -> Cat
;; Produce cat at next position
;!!!
(define (tock c) 1)  ;stub

;; Cat -> Image
;; produce image with CAT-IMG placed on MTS at proper x, y position
; !!!
(define (render c) MTS)
```

Note that we are maintaining a wish listof functions that need to be designed. The way to maintain the wish list is to just write a signature, purpose and stub for each wished-for function, also label the wish list entrywith `!!!` or some other marker that is easy to search for. That will help you find your unfilled wishes later.

Forming wish list entries this way is enough for `main` (or other functions that call a wished for function) to be defined without error. But of course `main` (and other such functions) will not run properly until the wished for functions are actually completely designed.

As you design the program remember to run early and run often. The sooner you can run the program after writing anything the sooner you can find any small mistakes that might be in it. Fixing the small mistakes earlier makes it easier to find any harder mistakes later.

Key and Mouse Handlers

The `on-key` and `on-mouse` handler function templates are handled specially. The `on-key` function is templated according to its second argument, a `KeyEvent`, using the large enumeration rule. The `on-mouse` function is templated according to its `MouseEvent` argument, also using the large enumeration rule. So, for example, for a key handler function that has a special behaviour when the space key is pressed but does nothing for any other key event the following would be the template:

```
(define (handle-key ws ke)
  (cond [(key=? ke " ") (... ws)]
        [else
         (... ws)]))
```

Similarly the template for a mouse handler function that has special behavior for mouse clicks but ignores all other mouse events would be:

```
(define (handle-mouse ws x y me)
  (cond [(mouse=? me "button-down") (... ws x y)]
        [else
         (... ws x y)]))
```

For more information on the `KeyEvent` and `MouseEvent` large enumerations see the DrRacket help desk.

Created Thu 21 Feb 2013 2:08 PM EST (UTC -0500)

Last Modified Fri 20 Sep 2013 12:06 AM EDT (UTC -0400)