Computer Science 110
The Art & Science of Computer Graphics
Mount Holyoke College
Spring 2008

Topic Notes: Programming

So far, we have used a very limited subset of the power of DrScheme. Next, we will examine some Scheme techniques that will help us develop more complex models and, later, animations.

Mathematical Functions

We have defined objects and values, and used some built-in mathematical functions.

We can also define *custom functions*.

We can think of these like mathematical functions. Suppose I want a mathematical function that adds two:

$$f(x) = x + 2$$

After defining the function f as above, we would know if someone were to write $f(7)$, that we should take 7, substitute it for x in the definition of f, and evaluate to 9.

We can make a similar definition in Scheme:

```
(define (add2 x) (+ x 2))
```

Then, if we want to add 2 to some number, we can also write

```
(add2 7)
```

In the definition, `add2` is the name of the function, `x` is its *parameter*, and `(+ x 2)` is the *body* or *definition* of the function. We are defining the bevavior of the *function call* `(add2 x)` to be `(+ x 2)`.

When it encounters the function `add2`, Scheme will recognize it as a function that takes one parameter. It will then do whatever is defined by the body of the function, replacing instances of the parameter `x` with the provided value 7.

In this case, this is somewhat silly. It's not very hard to type the original addition function. But functions can be significantly more complicated. A slightly more complicated function we might define:

```
(define (hypotenuse a b)
  (sqrt (+ (* a a)
           (* b b)
           )
        )
  )
```

Here, we provide two parameters, and have a more complex function definition.

After defining `hypotenuse`, we can use it in our programs, providing it the two numbers for `a` and `b` that will be plugged in for `a` and `b` in the definition of the function.

```
(hypotenuse 5 12)
```

We can also build functions to add capabilities to existing functions.

Recall the function `(random)`, which returns a number in the range [0,1).

Suppose we want a function that returns a value in a different range.

We can define such a function. The following computes a random value between low and high (low is possible, high is not).

```
(define (rand low high)
  (+ low (* (random) (- high low))))
```

Keep this one in mind – we'll use it later.

List Functions

Some of the things we've been dealing with, in particular coordinates and colors, are pairs or triples of values that are treated by Scheme as a *list*.

When we've specified lists, we've had to put a quote mark in front so Scheme doesn't try to "evaluate" the list as a function.

We can manipulate these lists in a variety of ways. First, we consider how to break down the list into components. Scheme provides some very cryptic list-manipulation functions with names that come from ancient history. Instead of using those, we'll define some functions with more meaningful names:

```
(define (first l) (car l))
(define (rest l) (cdr l))
(define (second l) (cadr l))
(define (third l) (caddr l))
```

Note that these are not defined automatically by Scheme or Mead, so you need to include the above lines in your program if you want to use them.

For example, if we want to get the first (the red) component of the Mead-defined color `magenta`:

```
(first magenta)
```

Should give us `1`.

We can define a list of anything:

```
(define aleast '('yankees 'redsox 'orioles 'bluejays 'rays))
```

And then use the above operations to manipulate this list.

We might also want to build a list out of the results of some other operations. If we want to construct such a list, we need to use the `(list)` function.

For example, we might want to write a function that takes an RGB color (a list of three numbers) as its parameter, and returns another color that is a darker or a lighter shade of the given color:

```
; RGB colors are lists.  The following function
; generates a darker RGB color:
(define (darker c)
  (list
   (/ (first c) 2.0)
   (/ (second c) 2.0)
   (/ (third c) 2.0)))

; Lighter could be defined as follows:
(define (lighter c)
  (list
   (- 1 (* 0.5 (- 1 (first c))))
   (- 1 (* 0.5 (- 1 (second c))))
   (- 1 (* 0.5 (- 1 (third c))))))
```

See Example:
`/home/jteresco/shared/cs110/examples/DarkerLighter`

The Conditional Function: `if`

We can make decisions in our Scheme programs. One way to do this is with `if`.

```
(if (cond) (domeiftrue) (domeiffalse))
```

See Example:
`/home/jteresco/shared/cs110/examples/Countdown`

We can also use the conditional to generate functions that create or manipulate Mead objects.

See Example:
`/home/jteresco/shared/cs110/examples/SaltShaker`

The key idea here is the `addSalt` function that uses a conditional to determine if items still need to be added, and if so, adds one then **calls itself** to add the rest.

Note that we can `tell` objects like `Group`s just like we can `tell` the `scene` or the `camera`.

We can use the function we developed in the previous example for multiple purposes:

See Example:
`/home/jteresco/shared/cs110/examples/SaltAndPepper`

In this case, we have two different types of objects and two different groups, but as use the same `addSalt` function to add the objects to the group.

Next, we make an even more general-purpose function – one that takes a list of objects that we wish to add, selecting each randomly.

See Example:
`/home/jteresco/shared/cs110/examples/SpiceMix`

Some things to note about this example:

- we define several types of objects and then package them up in a list to send them to the `addSpices` function
- when it's time to add one of the objects to our group, we select an entry from the list at random, using some new techniques:
  1. the `length` function to get the number of entries in a list
  2. the new form of the `random` function that takes an integer parameter `n` that results in a random integer (whole number) value in the range 0 to `n-1`
  3. the function (defined in this program) `nth` to retrieve the n^{th} entry of a list

---

Adding Multiple Objects

We sometimes want to add multiple objects in precise positions rather than randomly. As an example of this, we'll create a bunch of doughnuts.

First, to make the doughnut shape, we'll use a new Mead object type: the `Torus`.

When creating a `Torus`, we need to specify the dimensions. In this case, it's the radius of the cross section of the torus, and the radius of the circle traced out by the center of the cross section.

We set these by sending messages when we create our `Torus` object:

```
(object t Torus
        (major 50)
        (minor 20)
)
```

See Example:
`/home/jteresco/shared/cs110/examples/Doughnuts`

With our doughnut object defined, we now see how to add several of them to the scene in a fixed pattern.

The key function we will define is one that has a lot of flexibility in adding multiple objects:

```
(define (multiAdd n obj group initialXform deltaXform)
  (if (<= n 0) group ; return the group, if nothing to be added
      (begin  ; if something, add one, then the rest
        (tell group
              (add obj initialXform)
              )
        (multiAdd (- n 1) obj group
                  (compose initialXform deltaXform)
                  deltaXform)
        )
      )
  )
```

We see a typical usage of this in the `Doughnuts` example:

```
(multiAdd numDoughnuts doughnut scene
          (translate (- (* (/ numDoughnuts 2) doughnutSpacing))
                     0 0)
          (translate doughnutSpacing 0 0))
```

The parameters to the `multiAdd` function are defined as follows:

- `n` – the number of copies of the object to add
- `obj` – the object being added multiple times
- `group` – the group to which the items are being added (often a `Group` object, like the `scene`)

- `initialXform` – a transformation to be applied to `obj` to put the first instance that is being added into its correct location, orientation, and size
- `deltaXform` – a transformation that is applied to each successive object before it is added

In this example, we add a "row" of doughnuts to the scene.

We can extend this idea, using the `multiAdd` function twice, to add lots of rows of doughnuts.

See Example:
`/home/jteresco/shared/cs110/examples/MoreDoughnuts`

Combining Randomization with the `mulitAdd` Idea

Suppose we want to create a randomly-colored `Material` for each item in a group being added.

See Example:
`/home/jteresco/shared/cs110/examples/RandomColorMarbles`

Here, we see a modified version of the `multiAdd` function, which we call `randomMatAdd`. In addition to adding a group of objects as we saw in `multiAdd`, it creates a random color value and uses that to construct a new `Material` for use with each object.

```
(define (randomMatAdd n obj group initialXform deltaXform)
  (if (<= n 0) group
      (let* ([m (new Material)]
             ; to avoid too many dark and light colors
             ; and to avoid the grays completely, we'll
             ; generate a color as an HSV with a random
             ; hue, but saturation and value both of 1.
             [c (hsv2rgb (list (rand 0 360) 1 1))]
             )
        ; now we can give our material the properties we want
        (tell m
              (color c)
              (type 'plastic)
              )
        ; add one instance of our object to the group, but
        ; also specify our brand new material
        (tell group (add obj initialXform m))
        ; now add the other n-1 copies
        (randomMatAdd (- n 1) obj group
                      (compose initialXform deltaXform)
                      deltaXform)
        )
      )
  )
```

The basic structure of our function is very similar to `multiAdd`, and it takes the same parameters.

The difference comes in when it's time to add the object. Rather than wrapping our statements for the "$n > 0$" case in a `begin`, we wrap them in a new function, `let*`. The `let*` construct works like begin, but it takes an extra parameter where we can specify a list of "local" names that we can use throughout the remainder of the `let*`. This is a lot like using `define`, except that we don't need to worry about overwriting names defined outside of our function.

In this case, we'll define two names: `m` which will be our `Material`, and `c`, which will be our RGB color to use in the construction of the `Material`.

When we *declare* the name `m`, we use the name to refer to a new `Material`, but one to which we have not yet assigned any properties. We do that below, when we `tell m` those properties: we give it a color of the randomly-generated color `c`, and use the type `'plastic`.

Then, when we add it to our `group`, we specify the material `m` as part of the `add` message.

We can take this a little further and randomize more properties of the `Materals`.

See Example:
`/home/jteresco/shared/cs110/examples/RandomMaterialMarbles`

More Repetition Functions

The next two examples are building stacks of poker chips. Like a stack of poker chips in real life, the chips are not perfectly aligned on top of each other.

We first use a very basic poker chip, a squashed cylinder, and focus on the slightly randomized stacking.

See Example:
`/home/jteresco/shared/cs110/examples/PokerChipStack`

The key here is the `imperfectStack` function, described in detail in the comment near its definition in the example.

With the imperfect stack of very boring poker chips taken care of, we can think about how to make a more realistic poker chip. To do this, we'll use the `multiAdd` function from before in a different way – we'll use it to carve some little grooves in our chips.

See Example:
`/home/jteresco/shared/cs110/examples/FancyPokerChipStack`