A 3D-Modeling Language for MC28

Max Hailperin

November 27, 1995

Two years ago, the MC28 class and I designed a small, special-purpose programming language intended for producing three-dimensional scenes for viewing. The following semester, the MC28 class and I revised the language design somewhat. Now I again have the urge to revise the language design some more. This handout is my best shot at the new, re-revised language. However, you are welcome to point out gaffes I've made in the re-revising, or to suggest improvements to the language design.

Here is a very simple program in this language:

```
((compose sphere
          (cone where (point base-center is (0 -1 0))))
 where
 (material stuff is blue-plastic)
 (scalar radius is .5))
```

This program would put together a cone and a sphere, both made out of blue plastic. The radius of the sphere and of the base of the cone are both .5 units. The base center of the cone is at the point $(0, -1, 0)$. Other, unspecified, properties take on their default values. In particular, the sphere's center is at $(0, 0, 0)$ and the cone's apex is at an offset of $(0, 1, 0)$ from it's base center, i.e. in this case it is at $(0, 0, 0)$.

This defaulting occurs not through some special-purpose mechanism, but rather through general-purpose properties of the language. In particular, the language has no procedures and hence no parameters. Instead, it just has naming, where names are bound using definitions after `where`. However, names can be bound not only to values, but also to un-evaluated expressions. For example, the global environment of pre-defined names includes bindings for `center` and `apex` that could instead have been produced by the following definitions:

```
(point center is (0 0 0))
(calculate point apex as (offset (0 1 0) from base-center))
```

The definition with `calculate` and `as` in it means that the name should be bound to an unevaluated expression, rather than to the value produced by evaluating the expression. Later, when `apex` is actually used, the expression is evaluated in the current environment. Where is `apex` used? Conceptually, `cone` is bound to an unevaluated expression that contains `apex`, `base-center`, and `radius`. So, when `cone` is used, that expression is evaluated, which causes `apex` to be used, which causes the `offset` expression to be evaluated.

This treatment of naming can be extended to add new, composite structures that also act as though they had named parameters with default values, just like the built-in `cone`. Also, we can set things up so that two names are by default calculated in terms of one another. That way, the programmer can override either, and the other will be calculated from it. This is used in the following program to allow mannequins to be positioned either by specifying their base or by specifying their top. This program shows off lots of other language features too:

```
((compose
  (mannequin where (point base is (0 0 0))) ;this one stands on
  (mannequin where (point top is (0 0 0))) ;this one's head

  (mannequin where (point base is (.5 0 0)) (scalar height is .4)) ;short

  (mannequin where (point base is (-.5 0 0)) ;conehead with miniskirt
             (scalar skirt-longness is .25)
             (calculate scene head as
                        (cone where (point base-center is neck-top)
                              (point apex is top)
                              (scalar radius is (distance between neck-top and
                                                          head-center)))))

  (mannequin where (point base is (0 0 -.5)) ;hand off waist and blue
             (calculate point right-hand as
                        (offset (shoulder-width 0 0) from waist))
             (material stuff is blue-plastic)))

 where

 ;; One or the other of top and base must be overridden to prevent an
 ;; infinite loop.  They are both given as calculations in terms of
 ;; each other so that some mannequins can be positioned by specifying
 ;; the base and others by specifying the top.
 (calculate point top as (offset (0 height 0) from base))
 (calculate point base as
           (offset height towards (offset (0 -1 0) from top) from top))

 ;; The following are the (default) numerical parameters for mannequins.
 (scalar height is .5)                  ; most common parameter to vary
 (scalar skirt-fullness is .5)          ; this could vary for fashion reasons
 (scalar skirt-longness is .5)          ; ditto for this
 (scalar head-size is .04)              ; the rest of these are more constant
 (scalar neck-length is .02)
 (scalar neck-radius is .015)
```

```
(scalar shoulder-width is .06)
(scalar golden-ratio is .618) ; controls relative height of waist
(scalar penetration is .04)   ; controls how non-point-like the waist is
(scalar arm-radius is .01)
(scalar leg-radius is .012)

;; Next come various reference points computed using the above
;; numbers; normally only the numbers are varied, but these points could
;; also be overridden to pose the mannequin differently, e.g. with hand
;; other than on waist.
(calculate point head-center as
  (offset head-size towards base from top))
(calculate point neck-top as
  (offset head-size towards base from head-center))
(calculate point neck-base as
  (offset neck-length towards base from neck-top))
(calculate point right-shoulder as
  (offset (shoulder-width 0 0) from neck-base))
(calculate point left-shoulder as
  (-1 from neck-base to right-shoulder))
(calculate point mid-torso as
  (.5 from neck-base to waist))
(calculate point right-elbow as
  (offset ((distance between neck-base and mid-torso) 0 0)
          from (offset (shoulder-width 0 0) from mid-torso)))
(calculate point left-elbow as
  (-1 from mid-torso to right-elbow))
(calculate point right-hand as waist)
(calculate point left-hand as waist)
(calculate point waist as
  (golden-ratio from base to top))
(calculate point skirt-base as
  (skirt-longness from waist to base))
(calculate point hem as
  (skirt-fullness from skirt-base to
                  (offset ((distance between waist and skirt-base) 0 0)
                          from skirt-base)))
(calculate point right-leg-highest-visible as
  (offset (leg-radius 0 0) from skirt-base))
(calculate point left-leg-highest-visible as
  (-1 from skirt-base to right-leg-highest-visible))
(calculate point right-foot as
  (offset (leg-radius 0 0) from base))
(calculate point left-foot as
  (-1 from base to right-foot))

;; Next come the various component parts, defined in terms of the reference
;; points.  These would only be overridden to make a mannequin different in
```

```
;; the shape of its parts, for example a conehead.
(calculate scene head as
  (sphere where (point center is head-center) (scalar radius is head-size)))
(calculate scene neck as
  (cylinder where (point top-center is neck-top)
            (point base-center is neck-base)
            (scalar radius is neck-radius)))
(calculate scene torso as
  (cone where (point base-center is neck-base)
        (point apex is (offset penetration towards base from waist))
        (scalar radius is shoulder-width)))
(calculate scene arm as                ; generic template for both arms
  ((compose (cylinder where (point base-center is elbow)
                      (point top-center is shoulder))
            (cylinder where (point base-center is hand)
                      (point top-center is elbow)))
   where (scalar radius is arm-radius)))
(calculate scene left-arm as
  (arm where (point shoulder is left-shoulder)
       (point elbow is left-elbow) (point hand is left-hand)))
(calculate scene right-arm as
  (arm where (point shoulder is right-shoulder)
       (point elbow is right-elbow)
       (point hand is right-hand)))
(calculate scene skirt as
  (cone where (point base-center is skirt-base)
        (point apex is (offset penetration towards top from waist))
        (scalar radius is (distance between skirt-base and hem))))
(calculate scene left-leg as
  (cylinder where (point base-center is left-foot)
            (point top-center is left-leg-highest-visible)
            (scalar radius is leg-radius)))
(calculate scene right-leg as
  (cylinder where (point base-center is right-foot)
            (point top-center is right-leg-highest-visible)
            (scalar radius is leg-radius)))

;; Now we're finally ready for mannequin itself.
(calculate scene mannequin as
  (compose head neck torso right-arm left-arm skirt right-leg left-leg)))
```

Finally, here is a description of the entire syntax of the language, one concept at a time:

Scene expressions

The top level of any program in our language is a scene expression, i.e., an expression describing a three dimensional scene. Scene expressions can also occur other than at the top level. In our language, each kind of expression has its own grammar, unlike in Scheme where there is a single unified grammar for all expressions. Here is the grammar for scene expressions:

⟨scene expression⟩ $\longrightarrow$ ⟨name⟩
 | `(`⟨scene expression⟩ `where` ⟨definition⟩$^+$`)`
 | `(compose` ⟨scene expression⟩$^+$`)`

If the ⟨scene expression⟩ is a ⟨name⟩, the ⟨name⟩ must be a name for a scene or for a deferred ⟨scene expression⟩. (A deferred ⟨scene expression⟩ would have been set up with a `calculate ... as` definition.) The value of the ⟨name⟩ is the scene it is a name for or the value of the deferred ⟨scene expression⟩.

If the ⟨scene expression⟩ is the form with `where` in it, then the constituent ⟨scene expression⟩ is evaluated in the environment resulting from the ⟨definition⟩s. That environment then disappears—the ⟨definition⟩s create a new environment only for this one ⟨scene expression⟩. The first ⟨definition⟩ is evaluated in the environment the overall ⟨scene expression⟩ is being evaluated in. The second ⟨definition⟩ (if any) is evaluated in the environment resulting from the first ⟨definition⟩, and so on, with the environment produced by the last ⟨definition⟩ being the one that the constituent ⟨scene expression⟩ is evaluated in.

If the ⟨scene expression⟩ is of the `compose` variety, the constituent ⟨scene expression⟩s are evaluated and the resulting scenes combined together.

Definitions

A definition can be either immediate or deferred; both kinds can be for any of the language's four types, namely scene, point, scalar, or material:

⟨definition⟩ $\longrightarrow$ ⟨immediate definition⟩ | ⟨deferred definition⟩

⟨immediate definition⟩ $\longrightarrow$ `(scene` ⟨name⟩ `is` ⟨scene expression⟩`)`
 | `(point` ⟨name⟩ `is` ⟨point expression⟩`)`
 | `(scalar` ⟨name⟩ `is` ⟨scalar expression⟩`)`
 | `(material` ⟨name⟩ `is` ⟨material expression⟩`)`

⟨deferred definition⟩ $\longrightarrow$ `(calculate scene` ⟨name⟩ `as` ⟨scene expression⟩`)`
 | `(calculate point` ⟨name⟩ `as` ⟨point expression⟩`)`
 | `(calculate scalar` ⟨name⟩ `as` ⟨scalar expression⟩`)`
 | `(calculate material` ⟨name⟩ `as` ⟨material expression⟩`)`

The evaluation of an ⟨immediate definition⟩ consists of the evaluation of its constituent expression and the extension of the environment with a new binding between the ⟨name⟩ and the resulting value. The evaluation of a ⟨deferred definition⟩ consists only of the extension of the environment with a binding between the ⟨name⟩ and a special "value" which contains the deferred expression.

Point expressions

Rather than presenting all the productions at once, we'll present the grammar bit by bit. The first two productions are much as for scene expressions:

⟨point expression⟩ ⟶ ⟨name⟩
  | `(`⟨point expression⟩ `where` ⟨definition⟩$^+$`)`

A point can also be constructed from the values of three ⟨scalar expression⟩s, which provide the x, y, and z coordinates:

⟨point expression⟩ ⟶ `(`⟨scalar expression⟩ ⟨scalar expression⟩ ⟨scalar expression⟩`)`

Another possibility is to compute the point by adding the corresponding coordinates of two points:

⟨point expression⟩ ⟶ `(offset` ⟨point expression⟩ `from` ⟨point expression⟩`)`

Rather than offsetting by an absolute (x, y, z) amount, it is also possible to go a specified distance in the direction of another point:

⟨point expression⟩ ⟶ `(offset` ⟨scalar expression⟩ `towards` ⟨point expression⟩ `from` ⟨point expression⟩`)`

Finally, the point can be calculated as some fraction of the way from one point to another, for example one-quarter of the way from p1 to p2 would be `(.25 from p1 to p2)`:

⟨point expression⟩ ⟶ `(`⟨scalar expression⟩ `from` ⟨point expression⟩ `to` ⟨point expression⟩`)`

Any point produced by an expression of this kind will lie on the line through the two given points. If the ⟨scalar expression⟩'s value is 0, then the first point will be returned, while if the value is 1, the second point will. Values between 0 and 1 indicate a fraction of the way in between these extremes, as in the example of one-quarter of the way. Values greater than 1 or less than 0 can be used for points beyond the two given points on the same line. For example, `(-1 from p1 to p2)` would evaluate to a point symmetrically on the other side of p1 from p2.

Scalar expressions

The grammar for these expressions is as follows:

⟨scalar expression⟩ ⟶ ⟨name⟩
  | `(`⟨scalar expression⟩ `where` ⟨definition⟩$^+$`)`
  | ⟨number⟩
  | `(distance between` ⟨point expression⟩ `and` ⟨point expression⟩`)`

The meaning of ⟨name⟩s and expressions with `where` in them is analogous to those cases above. The other two kinds of ⟨scalar expression⟩s, namely ⟨number⟩s and `distance` expressions, have relatively obvious meanings. Any legal Scheme real number can be used as a ⟨number⟩.

Material expressions

As usual, a ⟨material expression⟩ can be a ⟨name⟩ or a `where` expression; there is only one other more specialized form it can have:

⟨material expression⟩ $\longrightarrow$ ⟨name⟩
    | (⟨material expression⟩ `where` ⟨definition⟩$^+$)
    | (`ambient` ⟨point expression⟩ `diffuse` ⟨point expression⟩
      `specular` ⟨point expression⟩ `emissive` ⟨point expression⟩
      `shininess` ⟨scalar expression⟩ `transparency` ⟨scalar expression⟩)

The ⟨point expression⟩s in a material specification must evaluate to points with all three coordinates in the range zero to one; these represent the red, green, and blue components of the respective colors. Cooking up suitable material descriptions is painful enough that I'd recommend sticking with the pre-defined ones.

Names

Any legal scheme symbol can be used as a ⟨name⟩, with the exception of the following reserved words: `point`, `scene`, `scalar`, `material`, `calculate`, `as`, `where`, `is`, `offset`, `from`, `towards`, `to`, `distance`, `ambient`, `diffuse`, `specular`, `emissive`, `shininess`, `transparency`, `compose`, `between`, and `and`.

Also, there are the following predefined names:

- Scenes

Name	Expected bindings
cone	base-center(Point), apex(Point), radius(Scalar), stuff(Material)
cylinder	base-center(Point), top-center(Point), radius(Scalar), stuff(Material)
sphere	center(Point), radius(Scalar), stuff(Material)

- Points

```
(point center is (0 0 0))
(point base-center is (0 0 0))
(calculate point apex as (offset (0 1 0) from base-center))
(calculate point top-center as (offset (0 1 0) from base-center))
```

- Scalars

```
(scalar radius is 1)
```

- Materials

```
(calculate material stuff as red-plastic)
```

where the following have been appropriately defined: brass, bronze, polished-bronze, chrome, copper, polished-copper, gold, polished-gold, pewter, silver, polished-silver, black-plastic, blue-plastic, cyan-plastic, green-plastic, grey-plastic, pink-plastic, red-plastic, white-plastic, yellow-plastic, black-rubber, blue-rubber, cyan-rubber, green-rubber, magenta-rubber, orange-rubber, red-rubber, white-rubber, yellow-rubber, emerald, jade, obsidian, pearl, ruby, turquoise.