Computer Science 110
The Art & Science of Computer Graphics
Mount Holyoke College
Spring 2010

Topic Notes: Basic Modeling

DrScheme and Mead Basics

We will be working in a development environment called DrScheme, using the Mead Modeling System.

DrScheme is a general purpose development environment for the Scheme programming language.

We will see in lab how you can run DrScheme and use Mead yourself.

Some DrScheme/Mead basics:

- The DrScheme window has two main areas where we can type.
  1. The top window is where we develop our model descriptions.
  2. The bottom window is used to issue interactive commands to DrScheme.
  3. The same commands work in both places, but the commands in the top window are executed only when we click the "Run" button, while the ones in the bottom are executed immediately.
- Like any computer programming environment, DrScheme and Mead require that we follow a set of rules (the *syntax*) when forming commands. This will become more clear through examples.
- To be able to use Mead, we need to issue a command to tell DrScheme that we want to use it:

  ```
  (require (lib "Defs.ss" "Mead"))
  ```

  Nearly all of our model files will begin with this incantation.

  If Mead is working properly, it will print a message "Cheers!" and tell you the version number of Mead that you are using.

  This is also our first look at the syntax required – we'll be typing lots of parentheses!
- Here are some very simple commands to try out in the interactive window:
  1. `3`
  2. `3.4`

3. `1/2`
4. `6/8`
5. `pi`
6. `golden`

- Those don't compute anything, but these *function calls* do:

1. `(+ 1 2)`
2. `(+ 1 2 3)`
3. `(/ 8 13)`
4. `(/ 8.0 13.0)`
5. `(- 3)`
6. `(+ 1)`
7. `(* 0.232321 0.223112)`
8. `(sqrt 5)`
9. `(/ (+ 1 2 3) 3`
10. `(/ 2 (+ 1 (sqrt 5)))`
11. `(sin pi)`
12. `(random)`
13. `(* 100 (random))`
14. `"Hey there!"`

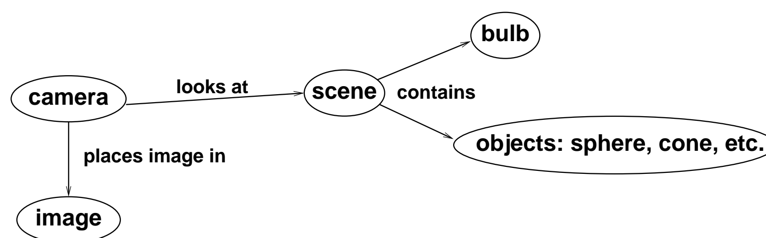
Modeling Basics

So Mead can clearly do some math. But we want to do some graphics.

We will be placing objects into a three-dimensional space

- our *x*-, *y*-, and *z-axes* meet at the *origin* at location `(0,0,0)`
- the *left hand rule* will help us find the positive directions of the axes

There are several interacting components that we need to be concerned about in our image construction:



As you might guess, the scene is our universe and it contains objects and light sources. Then a camera takes a picture of the scene from a specific vantage point and produces an image that we can view on our screen.

Building a Our First Model

So far, we have been primarily using the bottom window in DrScheme, since we just want to have DrScheme compute some simple value and print it out.

As we develop complex models, however, we will want to combine these simple statements into a *program*. We will develop our programs in the top window in DrScheme.

All of our programs start with the magic incantation:

```
(require (lib "Defs.ss" "Mead"))
```

Unless we specify otherwise, Mead always provides us:

- A `scene` where we place the objects we wish to model.
- A `camera` that can take a picture of the scene, which sits at position `(0, 0, -500)`, pointing at the origin.
- A `bulb` that illuminates the scene, which sits at position `(0, 300 -300)`.

We need to put something in the scene. To do this we send it a "message" (using the `tell` command), saying that we want to add a cube:

```
; build a default cube (100x100x100, gray, centered at the origin)
(tell scene
      (add cube)
)
```

This puts a cube in the scene. As we see in the *comment*, it is by default a gray color, sitting at the origin, and with dimensions 100x100x100. Normally, this isn't exactly what we want and we would specify a different position, size, and color for our cube.

The idea of a comment is very important – anything after a semicolon on a line in a Scheme program is ignored – it's just there for our benefit. Other than the semicolon, the syntax is not important for a comment, so we should make it human-readable. Here, we use the comment to remind ourselves that even though all we're saying is to add a cube, that cube is 100x100x100, centered at the origin, and is gray in color.

Use lots of comments in your models. It's helpful to you and to me. And it will get you better grades on your assignments!

So now we have an object in the scene. We can now take a picture of it:

```
(tell camera
      (shoot)
)
```

When we click the “Run” button in DrScheme, it executes the program we’ve developed in the top window. In this case, the `scene` is told to add a default cube and the `camera` is told to shoot to create an image of the scene.

If all is well, we should see some messages, and then an image of our gray cube should show up.

We have built a model!

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

Colors and Transformations

Now that we have a simple model, we can experiment with ways to make it a bit more interesting.

Instead of the default gray cube, we can construct it out of a red “plaster-like” material:

```
(tell scene
      (add cube redPlaster)
)
```

Mead knows about several colors, each of which has a “Plaster” material with that color:

```
black, dkGray, gray, ltGray, white, red,
green, blue, magenta, yellow, cyan
```

We can move our cube:

```
(tell scene
      (add cube redPlaster (translate -100 0 0))
)
```

Finally, some evidence of a third dimension!

We can change the size of our cube by specifying a scaling factor in each dimension. Let’s make a thin, flat red object:

```
(tell scene
      (add cube redPlaster (scale .2 1 .01))
)
```

We can also rorate our cube about any of the coorinate axes:

```
(tell scene
      (add cube redPlaster (xRot -45))
)
```

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

Other Primitive Objects

In addition to the cube we've already used as building blocks, there are a number of primitive objects we can add to our scenes, subject to the same types of transformations we've seen.

- Spheres

  By default, a `sphere` is 100x100x100 and centered at the origin.

- Cylinders

  A `cylinder` is also, by default, 100x100x100 in size. It is aligned along the y-axis and the ends are capped.

- Cones

  The default `cone` is also 100x100x100. It points upward along the y-axis and is capped.

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

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

Multiple Transformations

So far, we've added objects of a particular material, but have only seen how to transform the object in a single way.

If we want to add multiple transformations to the same object, we group them into a single transformation with a `compose`:

```
      (add cube redPlaster
           (compose
            (scale 2 2 2)
            (translate 10 10 10)
            (yRot -45)
           )
       )
```

Order matters! Consider these:

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

We can also make some coordinate axes:

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

Defining Objects and Groups

While we're looking at very simple models so far, you might already be thinking that it's going to be very tedious if we always have to create each of our objects from the primitives, overriding default attributes every time. What if we want several small black spheres?

```
(object snowmanEye Sphere
        (material blackPlaster)
        (scale .1 .1 .1)
)

(tell scene
      (add snowmanEye
           (translate -25 0 0)
       )
)

(tell scene
      (add snowmanEye
           (translate 25 0 0)
       )
)
```

We can also define groups of objects that we can later manipulate as a single object.

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

In this example, we see how to define a new object that we name `iceCreamCone`. It is not defined as one of our primitive types, but as a `Group`. And then we specify the components to include in our group by `add`ing objects just like we'd add them to our `scene`.

Once we've created the group object, we can add it to the scene or other groups just like we can add instances of our primitive object types.

Notice here that in our `iceCreamCone` we have specified that the cone is always made of `yellowPlaster` (yum), but have not specified a material for our ice cream. This means that

the ice cream will use the default grey material unless we specify one. We do just that when we add the `iceCreamCone` to the scene. The ice cream, since we did not specify a material in the group definition, takes on the material specified when adding the group to the scene. The cone, on the other hand, was given a specific material in the group definition, and that one will take priority.

Another example, developed by the Spring ’08 class:

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

More Colors, Lights, and Cameras

As we prepare for the first Mead lab, it’s time to consider how to make more colors, lights, and to change the properties of the camera. We’ll come back to each of these in more detail soon, but this will help you make more interesting (and maybe even more realistic) models right away.

Defining Colors and Materials

To make new colors of our “plaster” material:

```
(define orange '(1 0.54 0))  ; 100% red, 54% green, 0% blue
(object orangePlaster Material
   (type 'plaster)
   (color orange)
)
```

As some of you may be aware (and those of you who aren’t, now you are), each color is defined by a blend of red, blue, and green light - the three primary colors of light and those used in computer displays to make all of the colors we enjoy. Again, we’ll say a lot more about this later, but for now you can construct colors by experimenting with combinations of red, green, and blue intensities (which, in our case, must be in the range 0-1) to create new colors.

Note: the apostrophes in the Mead code above are important! Normally, scheme attempts to interpret any list of items in parentheses as a function. But here, the list `'(1 0.54 0)` is not something to be interpreted, it’s just a list of numbers. For the term `'plaster`, we need the apostrophe to say that this is just a word, not a reference to a defined object (like `orange` or `camera`).

Adding Lights

We can also add more lights to the scene.

The bulb that is provided automatically by Mead produces white light, has an intensity of 0.5 (0 is no light, 1 is most intense light) and is located at `(0, 300 -300)`.

```
(object blueBulb Light
```

```
        (color blue)
        (intensity .75)
        (pos '(100 300 -250))
)
(tell scene
      (add blueBulb)
)
```

This will put a light source with .75 intensity at position (100,300,-250). This bulb emits blue light.

We can shine this light onto a white cube and see the effect:

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

Modifying the Camera and Image

Besides telling our default camera to shoot a picture, we can send it messages about where to locate itself and where to look. And we can tell the image to use a different background color.

```
(tell camera
      (pos '(0 500 -500)) ; move the camera up and toward us
      (coi '(0 0 0)) ; look at the origin (the default behavior)
)

(tell image
      (background white) ; if you hate cyan backgrounds
)
```

Additional Examples

For a little extra practice, and in honor of the upcoming Winter Olympics, we will develop in class a model of a ski race gate.

A few notes about this example:

- We will use named constants (like `orange` above) to define meaningful numbers. The advantage here is that we can define things like the diameter and height of the posts, the distance between them. We can then change these sizes by changing only the constant definitions and the changes will be applied throughout the model.
- We will use named components and groups as we did in previous examples.

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

And another example for your reference: Tinker Toys!

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