



Computer Science 112

Art & Science of Computer Graphics

The College of Saint Rose
Fall 2015

Topic Notes: Model Building

We have seen many of the basics of Ambrosia, and now we will look at how to apply these basic ideas (and some new ones) to build more interesting models.

Good Model Building

First, here are some quick guidelines to building a good Ambrosia model. Some of these we have seen, others will become more clear as we progress through our current topics.

1. Include comments. Comments help you or others understand your model later.
2. We will soon be assigning names to parts of our models. When you name something, give it a descriptive name. This will also help you or others understand your intent.
3. When building more complex models, start by defining the fundamental building blocks at the start of the model. These will then be used and combined and augmented as you proceed to produce the final result.

A typical model will follow this general outline:

- A comment at the top of your files describing your model. This should always include your name.
 - Definition of constants that specify sizes and positions and other parameters of your model. These should also be commented.
 - Construction of the actual model. Again, do not be stingy with comments.
 - One or more camera operations to generate your image(s).
-

Names, Classes, Instances

One of the fundamental ideas of computer programming, including our work building Ambrosia models with the Python programming language, is the ability to define *names* for values and objects.

Many names have been defined by Ambrosia, some of which we have already seen: *e.g.*, `cube`, `scene`, `bulb`, and `redPlaster`. Ambrosia also defines names for a few numbers, such as `pi` and `golden`.

We can ask Python the value assigned to its name by typing it.

```
>>> pi
3.141592653589793
```

We can also use it anywhere we'd normally use its value:

```
>>> pi * 3 * 3
28.274333882308138
```

for the computation of the area of a circle whose radius is 3.

We can define our own names using Python's *assignment statement*. For example, if we wanted to define a name that represents the size of our ice cream scoop in the ice cream cone example.

```
scoopDiameter = 20
```

We can then use the name anywhere we'd otherwise use the number 20.

The names can be used for the result of arithmetic expressions as well:

```
scoopScale = scoopDiameter / 100
```

And once we have these names, we could use them to add a $20 \times 20 \times 20$ sphere (apparently representing a scoop of ice cream) to our `scene` so it sits atop the origin (rather than being centered there):

```
scene.add(sphere, scale(scoopScale, scoopScale, \
scoopScale)*translate(0, scoopDiameter/2, 0), redPlaster)
```

(Note: the `\` at the end of the first line indicates a continuation of the line on the next.)

There are many advantages to introducing names such as these.

One is the improved readability of the program. Rather than having numbers like 20 and 0.2 in the program with no obvious indication of what those numbers mean.

A second is that we can change the assignment of `scoopDiameter` in one place at the top of the program and have the `scoopScale` and (most importantly) the size and position of the scoop in the `scene` be adjusted accordingly.

Some of the names we have been using are actually *instances* of *classes* of objects.

For example, the name `cube` is an instance of the class `Cube`. The class is like a factory or cookie cutter that is used to form new instances. Somewhere deep in Ambrosia, there is a statement:

```
cube = Cube()
```

This says using the class `Cube` as a template, create a new instance of that type of object (yes, a `cube`), and name it `cube`.

We can create our own instances of these classes and give them our own names if we want (and we will want):

```
box = Cube()  
scene.add(box, redPlaster)
```

We can then use `box` instances the same way we use `cube`. Here, we have gained little if anything, but we will see many places where these names will come in handy.

A few notes about these:

- Names of classes begin with upper case letters (*e.g.*, `Cone`, `Camera`).
- Objects are instances of classes. They begin with lower case letters (*e.g.*, `camera`, `yRot`, `scene`).
- A *message* or *method* is a command to **do** something: add an object to the scene, shoot an image with the camera, or change the background color of the image to be generated:

```
scene.add(cube, magentaPlaster)  
camera.shoot()  
image.background(purple)
```

The names can refer not only to a “default” version of one of our objects, but to one with some transformations already applied. Consider this model:

```
tiltCone = Cone().zRot(45)  
scene.add(tiltCone, translate(100,0,0), redPlaster)  
scene.add(tiltCone, translate(-100,0,0), bluePlaster)
```

Here, we give create an instance of the class `Cone`, apply the `zRot` transformation to it, and name that resulting object as “`tiltCone`”. So when we then add our `tiltCones` to the scene, they’re already rotated by 45 degrees about the z-axis. Any transformation we apply after that is effectively composed with the `zRot` that was applied when we created our `tiltCone`.

We can also apply a material to an object and give that a name:

```
redCone = Cone().material(redPlaster)
```

And when we add any `redCone` to our scene, it's already given the `redPlaster` material property, and we do not need to apply it again.

Several properties can be applied at once, but the syntax is a bit different here:

```
crazyCone = Cone().material(greenPlastic).zRot(90).scale(.5, .5, .5)
```

Here, we construct a $100 \times 100 \times 100$ Cone, apply the `greenPlastic` material to it, apply the `zRot(90)` transformation to that green cone, then apply the `scale(.5, .5, .5)` transformation to that rotated green cone, and call the resulting shrunken, rotated, green cone `crazyCone`.