Computer Science 112
Art & Science of Computer Graphics
The College of Saint Rose
Spring 2016

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

Before we introduce any real new Ambrosia constructs, we take a quick look at some guidelines to building a good Ambrosia model. Some of these we have seen, others will become more clear as we progress through our upcoming 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).

Defining Names

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 sizes of our ice cream cone and scoop in the ice cream cone example.

```
coneHeight = 50
coneDiameter = 20
bottomScoopDiameter = 20
```

We can then use the names anywhere we'd otherwise use the corresponding numbers.

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

```
topScoopDiameter = .9 * bottomScoopDiameter
```

And once we have these names, we could use them to add a $20 \times 50 \times 20$ upside-down cone to our `scene` so it sits atop the origin (rather than being centered there):

```
scene.add(cone,
          scale(coneDiameter/100,coneHeight/100,coneDiameter/100)*
          xRot(180)*translate(0, coneHeight/2, 0),
          yellowPlaster)
```

There are many advantages to introducing names such as these.

One is the improved readability of the program. Rather than having numbers like 20, 50, 0.2, and 0.5 in the program, with no obvious indication of what those numbers mean, we know in this case we meant the diameter of the cone, height of the cone, and scaling factors to achieve these sizes.

A second advantage is that we can change the assignment of these names in one place at the top of the program and have the sizes and positions of the cone and scoops in the `scene` be adjusted accordingly.

On the Wiki: IceCreamConeWithNames

Note that in this example, we also see for the first time a second form of the `scale` transformation:

```
scene.add(sphere,
          scale(bottomScoopDiameter/100)*translate(0,coneHeight,0),
          redPlaster)
```

This one take just one number as a parameter instead of three. When we use it in this form, it applies the same scaling factor in all three dimensions. So,

```
scale(x)
```

is equivalent to

```
scale(x,x,x)
```

when applied anywhere a `scale` transformation can be applied.

Classes and Instances

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`. In Python (and hence in our Ambrosia models), the accepted convention is to give classes names that are capitalized and instances names that are not.

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 will think of the above statement as going to "`Cube` factory" to get a brand new `Cube` off the assembly line, and labeling it with the name `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)
```

This time we call the `Cube` from the `Cube` factory by a different name, `box`. We can then use the instance named `box` 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 ("factories") 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 `tiltCone`s 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 resuling shrunken, rotated, green cone `crazyCone`.

If you prefer, the same can be accomplished in steps:

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

Here is an updated version of our ice cream cone model that defines some named parts as it is being constructed:

On the Wiki: IceCreamConeNamedParts