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

Topic Notes: Basic Modeling

Getting Started with Ambrosia

As you have heard by now, much of our work for the course is done using the Ambrosia Modeling System, which will allow you to develop models in the Python programming language, and have those models rendered into images, and later, animations, using the Persistence of Vision raytracing program (POVRay).

We will get into Ambrosia and our first graphical models very quickly, but first we need to learn how to start Python.

Note: these notes assume you are using Ambrosia on the college's Macintosh systems. It should work on all Macs in public labs on campus, and a very similar process can be used your own Mac, if you have one, but requires some software to be installed.

Those interested in running Ambrosia from a Windows system will need to use our remote rendering server, `ascg.strose.edu`. This involves setting up an account for you there and setting up some software on your PC to help you access it.

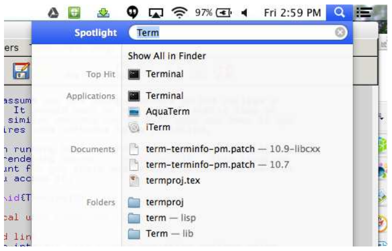
MacOS X `Terminal`

Most computer users interact with modern operating systems using *graphical user interfaces (GUIs)*. Icons represent applications, files, and folders. Running programs present one or more graphical windows though which the user interacts with the keyboard, mouse, touch screen, or other devides. GUIs are nice, and have made computing accessible to a wide audience of users, but they are not always the most efficient way to interact with the computer. This might seem old fashioned, but many tasks can be accomplished much more efficiently by interacting with the system using *shell*, or *command line*, where commands are issued directly to the system by typing them at the keyboard.

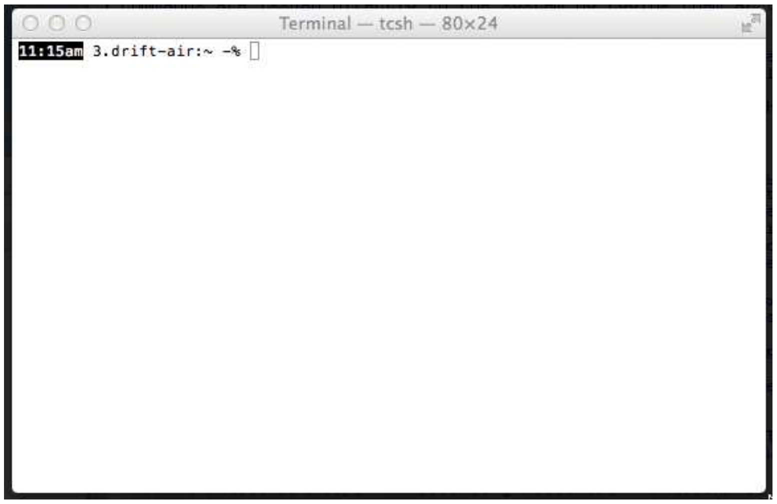
When working at the command line, you will be presented with a prompt. This is your direct interface to issue commands to the operating system. When you type a command here, the shell will execute the command on your behalf, print out any results, then reissue the prompt.

On our Mac systems, you will run Python from the command line. Once logged into a Science Center 469A Mac system, you can access the command line in a few ways – we'll start with the `Terminal` application.

The easiest way to launch the `Terminal` application is to use the Spotlight (the magnifying glass in the upper right corner of the desktop) and type "`Terminal`" and click its icon to launch.



You should see a new window pop up that looks something like this:



```
11:15am 3.drift-air:~ -%
```

Your prompt might look different (mine's customized it with some extra information) but there should be some sort of prompt.

Give your shell a command to make sure it works. Try typing "`who am i`" and make sure it produces some output.

Along the way, you will learn some of the basics of working with the command shell. The learning curve can be steep, but it is a very valuable skill. This gives you a more direct and powerful way to tell the computer what you want it to do.

Interacting with Python

Ultimately, we will almost always run Python programs by creating a program in a document and then sending that to Python to run. But to get us started, we will use Python in an interactive mode.

To enter the Python programming environment, we simply type its name at our command prompt:

```
python3
```

The "3" at the end is necessary because we will be working with the third major revision of the Python programming language, and this ensures that we will get the correct version. Ambrosia works only with Python version 3.

If you successfully issued the command, you should see something similar to this:

```
Python 3.4.3 (default, Feb 26 2015, 09:25:49)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.56)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The first few lines are providing some information about the Python version, and the ">>>" is a new prompt, this time one where you can interact directly with Python.

Each time we issue a command to Python, it will either tell is a result of that command, or give an error message. For example, if you give Python a number, it gives you that same number back:

```
>>> 8
8
```

Important Note: the ">>>" in the interaction examples is the prompt that Python prints to your screen. You will type only whatever is on the line after the prompt. In the above case, just the "8".

But if we give it a word, it is not quite so happy:

```
>>> python
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'python' is not defined
```

But it turns out that if we put that same word in quotes, it will echo the word back to us:

```
>>> 'Python'
'Python'
>>> "Python"
'Python'
```

As you can see, Python was not picky about whether we used a single or a double quote, but when it gives us back our "answer", it was in single quotes in both cases.

So far, pretty useless, right? No one needs a programming language to have it echo back things we type. This is a **computer** programming language after all, so we should be able to compute some things. And we can. Python can do math:

```
>>> 2+2
4
>>> 7*3
```

```
21
>>> 42-10
32
>>> 9/3
3.0
```

Even in these first few simple examples, there are some possibly surprising things. Our addition and subtraction commands look a lot like those from math, but note that Python (and many programming languages) use "`*`" to denote multiplication, as the "`x`" character is already used for the letter x, and there is no separate key on your keyboard that makes a real "×" symbol. Similarly, there is no "÷" key on your keyboard, so Python (and many other languages) uses "`/`" to represent division.

More complicated mathematical expressions are also supported, with the same *precedence rules* that you are familiar with from math, which are used to determine the *order of operations*. Specifically, if we are evaluating a complex expression with `+`, `-`, `*` and `/`, the `*` and `/` operations are evaluated first, left to right, then the `+` and `-` operations are evaluated, left to right. Again as with standard mathematical notation, the order of operations can be overridden using parentheses. Examples:

```
>>> 4*6+5
29
>>> 4+6*5
34
>>> (4+6)*5
50
>>> 4+9/3-9*2+30
19.0
>>> (4+9)/(3-9)*(2+30)
-69.33333333333333
>>> 1+2+3+4+5+6+7+8+9+10
55
```

Like any computer programming language, Python requires that we follow a rigid set of rules (the *syntax*) when forming commands. While the syntax of human languages is quite flexible, and we are capable of understanding someone who forms a sentence with an unusual (looking at you, Yoda) or even incorrect syntax. Programming languages require that we follow their syntax precisely. For example, you could tell a person "Compute 4+5" or "Add 4 and 5" or ask "What's 4 plus 5?" and they'll respond with 9. With Python, we have to specify it as "`4+5`". If we try to specify it slightly differently, Python will be unable to understand:

```
>>> add 4 and 5
  File "<stdin>", line 1
    add 4 and 5
```

```
        ^
SyntaxError: invalid syntax
```

We will do more arithmetic later, but we can also ask Python to do more complicated math as well:

```
>>> import math
>>> math.sin(0)
0.0
>>> math.cos(0)
1.0
>>> math.sqrt(49)
7.0
>>> math.pow(4,3)
64.0
>>> math.sin(math.pi)
1.2246467991473532e-16
>>> math.cos(math.pi)
-1.0
```

We will discuss more of Python's mathematical capabilities when we need them later on.

Ambrosia Basics

We will also learn more of what Python can do for us beyond calculator capabilities as we go on, but for now we turn our attention to Ambrosia, which will allow us to start generating some graphics.

Ambrosia is built in to the Python programming language. It is rather a *library*, so we have to do some extra set up to be able to use it. This involves a few steps, some of which were already done by placing Ambrosia's files in the right place on the lab computers, and configuring Python to know where to find them. However, we still need to tell Python, every time we use it and want to use Ambrosia, that we will be using it.

This involves two steps. The first is done at our Unix prompt, **before** we launch Python. Each time you open a Terminal, you will need to enter the command

```
. 112
```

(note the space between the `.` and `112`)

Then, launch Python, as above, with the `python3` command.

The second step is within Python, and you know you are "in Python" if you see the >>> prompt, we issue this statement:

```
>>> from ambrosia import *
```

If you did the "`. 112`" step before launching Python, this will not generate any output – you will just get your prompt back. We have not asked it to compute anything, so it doesn't give us any value back. We should just get a new prompt.

To see if Python and Ambrosia are set up correctly at this point, we can ask it to take a picture of the model we have constructed so far:

```
>>> camera.shoot()
```

If all goes well, you should get a short informational message, something like

```
[terescoj-23068.png: 23:52:23...23:52:24]
```

and then a blank, white window should appear. Not exciting, but correct! We have just asked our Ambrosia "camera" to "shoot" a picture of the model we have described. Since we did not yet describe a model, the picture it takes is a picture of an empty model: and we see nothing.

Note that we are required to type the command exactly as above. Any variation like "`shoot camera`" or "`camera().shoot`" will not work. When we imported Ambrosia, it defined for Python a thing named `camera`, and when we want to tell that camera to do something, in this case to `shoot`, we have to use the appropriate syntax: the "`.`", followed by the name of the thing we want to tell the camera to do, "`shoot`", followed by "`()`". We'll refer to this as *sending a message* to the `camera` to `shoot`. This is sometimes also called *calling a method* or *calling a function* of the camera object.

Over the course of the semester, we will see enough Python that what's happening here will become more clear.

Now let's add something to our model, so when we take a picture we see something.

```
>>> scene.add(sphere,redPlaster)
```

This should produce a cryptic message, something like:

```
<ambrosia.objects.Group object at 0x10255f108>
```

which is Python's way of telling you that the command was successful.

What we're seeing here is that Ambrosia also provides something called a "`scene`", and we can tell the `scene` that we want to add something to it. Unlike the `camera`'s `shoot`, which had just an open and close parenthesis pair, we have some information inside the parentheses. These are called *parameters*, and provide some information that go with the message we are sending. In this case, we are sending a message to the `scene` to `add` something. So it needs to know exactly what it is we would like to add. Here, we are going to add a `sphere` (anothing thing provided for us by Ambrosia), and that we'd like to build that sphere out of a material called `redPlaster` (also provided for us by Ambrosia). Again, the syntax is important, and every bit of what we typed must be typed precisely: the order, spelling, case of the characters in the names, and all of the punctuation.

OK, so we have placed into our scene a sphere made of red plaster. In order to see it, we have to take a picture. We know how to do that:

```
>>> camera.shoot()
```

This time, you should see a lovely red circle. Congratulations, you have constructed and photographed your first Ambrosia model!

Hopefully you can see some shading on the circle that implies that it's really supposed to be a sphere. We have a two-dimensional computer display trying to display a three-dimensional scene, so we will rely on visual cues such as lighting effects to indicate depth.

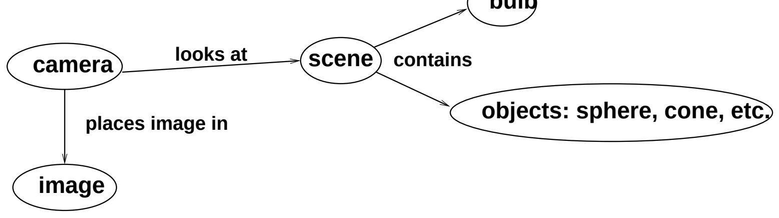
Specifically in this case, Ambrosia has generated a model in a scene with a single sphere made of a material that is supposed to look like a red plaster. It has placed that sphere in space in a position below a light source (hence the brighter red on the top and darker red, almost black, along the bottom), and the camera taking a picture from off to one side. We'll consider all of this more carefully soon.

The Ambrosia Players

What we have just begun doing is placing objects into a three-dimensional space to construct a *virtual model* using a "python-based" *modeling language*.

- our *x*-, *y*-, and *z-axes* (the *coordinate axes*) meet at the *origin* at location `(0,0,0)`
- the *left hand rule* will help us find the positive directions of the axes
- we will usually think of positive x going to the right, positive y going up, and positive z going into the screen (but it's all a matter of perspective – where our camera is located and where it is pointing matters, of course)

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.

This image is created by *projection* of the model onto a virtual *image plane*. We will not necessarily see the entire model – it will be *clipped* so we can only see the area in the *view volume*.

When we built our first model with the red sphere, we never specified anything about the camera (where is it? in what direction is it facing? what kind of lens does it have?) nor did we place any lights in the scene.

Unless we specify otherwise, Ambrosia will always provide us with:

- 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)`.

If we place our objects near the origin, as we did when we added the red sphere (objects are placed at the origin unless we say otherwise), there will be some light on them and the camera will be "looking at" them when we ask it to shoot.

Storing our Models/Programs in Files

Even with the small "red sphere" model we looked at above, we had to type a few lines at the Python prompt to get the image we wanted. As our models, and correspondingly, our Python programs, get more complicated, we will not want to be typing all of those statements over and over. So we will write the programs in a file, and have Python operate on that file. This way, we can incrementally create (and fix, when we inevitably make mistakes) our programs without retyping the things that already work. Think of this as the same thing as writing a paper in a word processor. You don't always write the entire paper and print it in one sitting. You will save your document as a file on some persistent storage device, so you can continue to work on it, print it, or send it somewhere later on.

To get started, we first need to learn a bit about the way to access your network storage on the College's Mac systems. By using the network storage, your files will be available when you log into any public computer on campus, both Mac and Windows (though they won't do you much good in Windows).

When you log in, you should see a "network hard drive" icon on your desktop, named with your userid. If you click on that, you should see the same folders you have seen when using the "H Drive" on the College's Windows systems (if you have used them). Open that folder and create a folder there for your work for this course. Something like "cs112" would be a good choice. Do not use spaces or punctuaction in the name.

To use Python with Ambrosia, we need to access this same folder from the Terminal. To do so, when you open a new Terminal, you will type

```
cd /Volumes/youruserid/cs112
```

where you replace "youruserid" with your user ID and "cs112" with whatever you named the folder you created in the previous step.

If you haven't already done so in the Terminal window in which you're working, set up for Ambrosia with

```
. 112
```

But now, instead of starting Python and typing in our statements at its >>> prompt, we will create a file that contains our entire Python program describing an Ambrosia model.

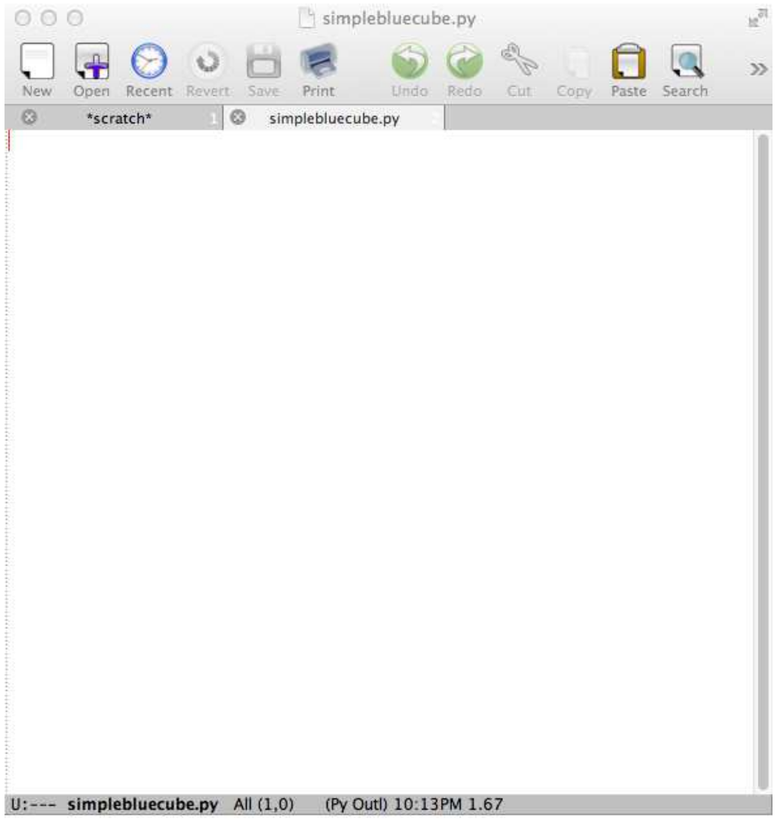
To do this, we first need to create the file. At the Terminal's command line, if we wanted to create an empty file called "SimpleBlueCube.py", the command to use is

```
touch SimpleBlueCube.py
```

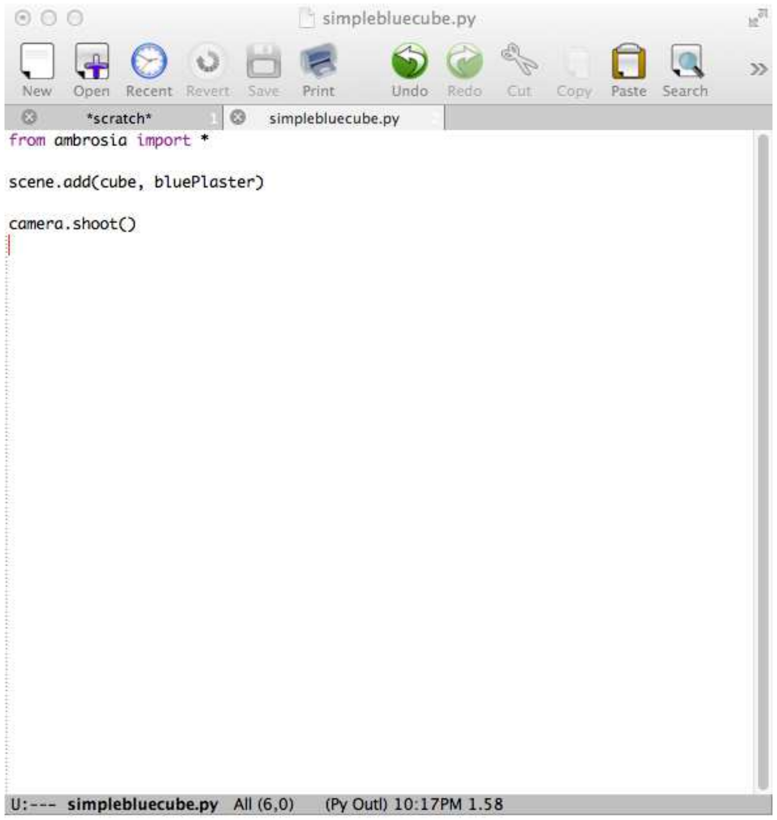
We now will open that file in an editor. There are many editors to choose from, and we will use one called "Aquamacs". Aquamacs is a Mac version of the very powerful Emacs editor that is used by many Unix/Linux users, but fortunately it also has a straightforward menu system so it's usable by less experienced users as well. To open the file we just created using Aquamacs so we can add our Python statements, we will use the command

```
open -a Aquamacs SimpleBlueCube.py
```

You should see a new window pop up that looks something like this:



In this window, we type the Python statements to create the model – just like we previously typed them at the >>> prompt in Python:



```
from ambrosia import *

scene.add(cube, bluePlaster)

camera.shoot()
```

```
U:--- simplebluecube.py All (6,0) (Py Outl) 10:17PM 1.58
```

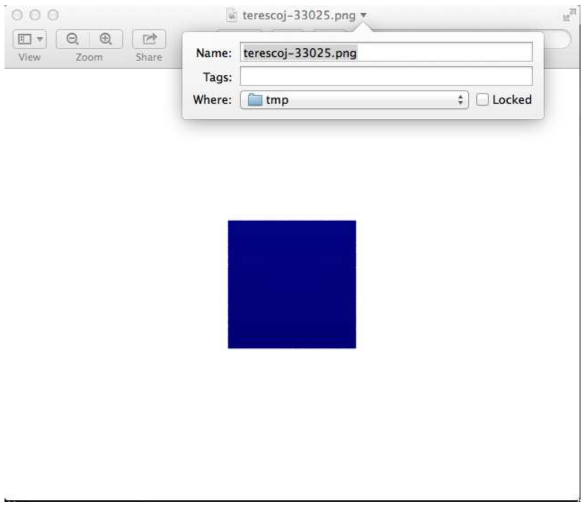
Hit “Save”, and then return to the Terminal window. We will again run Python, but instead of just typing “`python3`”, we will also type the name of the file we just created:

```
python3 SimpleBlueCube.py
```

You should get a short informational message from Python, then a window should pop up showing the image showing our scene, which will look like a blue square (actually a blue cube, but we’re looking directly at one of its sides).

At this point, we could go back to Aquamacs and make changes to the Python program, save it, and re-run the command above to try again. Hopefully you can see that this is a more efficient way than having to re-type everything each time you start Python!

It’s not especially important with this one, but let’s make sure we see how to save the image generated. Ambrosia automatically opens the generated image in the Mac’s “Preview” Program, but the image is stored in temporary space on the computer’s hard drive, and with a fairly cryptic name. In order to save it somewhere more permanent (like your network storage), select the Preview window that has the image you want to save and double click on the name of the file in the bar at the top of its window. That brings up a small window where you can type in a new name and choose a new location:



Name: terescoj-33025.png
Tags:
Where: tmp
Locked

Select an appropriate and descriptive name (but be sure to keep a `.png` extension), and choose your folder where you stored the Python program.

If all went well, you should be able to see both your Python program and your image file in the folder you created at the start of this exercise.

Our class examples will all be placed on the wiki. They will be referenced in these notes, as below, linked from the day's lecture page, and available in the "Spring 2016 Class Examples" page on the wiki.

On the Wiki: SimpleBlueCube

When you look at the version on the wiki, you will notice that I have added several additional lines of English text on lines that start with a "#". These are called *comments*. Python will ignore these completely, but they are very important for us. They allow us to include additional information in our Python programs, such as author information and explanations of what the actual Python statements are supposed to do. Commenting is essential to make all but the simplest programs understandable to a human. You will be expected to include comments in the Python programs that describe your Ambrosia models throughout the semester.

Ambrosia Primitive Objects

We have encountered two of the four primitive building blocks provided by Ambrosa: the sphere and the cube. The others are the cylinder and the cone.

To see these, try replacing the "`cube`" in `SimpleBlueCube.py` with each of the others and execute the program again.

Here's what you need to know about these primitive objects. Each measures $100 \times 100 \times 100$, and is **centered** about the origin. Note in particular that they do not sit atop the origin. We will see later why this is the case.

The `cube` and `sphere` are symmetric about the origin, so we do not need to concern ourselves with their orientation, but the others are not.

- The `cylinder` has its axis aligned along the y-axis and the ends are capped.
- The `cone` points upward along the y-axis and is capped.

For a short animation that might help you visualize the axes and camera position, check out this model:

On the Wiki: CameraAndAxes

Here, the x-axis is shown in red, the y-axis in yellow, and the z-axis in blue. The green cone is at its default size of $100 \times 100 \times 100$ and position centered about the origin. The purple cone represents the camera position, with the tip of the cone the direction of the camera's view.

Note: if you want to run this yourself, you will need to create a folder within the same folder where you run this model called `movieImages`.

A First Look at Colors and Materials

In our very simple models so far, we have created objects with just a few colors. To be accurate, it is not the colors we specify when adding objects, it is a *material*. We will study material properites, including specification of colors, very soon.

For now, we will look at some of the pre-defined materials provided by Ambrosia. There are 2 material types (`Plaster` and `Plastic`) that come in 12 colors (`black`, `ltGray`, `gray`, `dkGray`, `white`, `red`, `yellow`, `green`, `cyan`, `blue`, `magenta`, and `purple`) that can be used in any combination. As you might guess, the "plaster" materials are supposed to look like plaster (not "shiny" at all) and the "plastic" materials are supposed to look like plastic (a little "shiny"). The material names are a color followed by a material type, like `redPlaster` or `greenPlastic`.

If we do not specify any material when adding an object to the scene, it uses the *default material*, which will show up as a black or dark grey. We rarely will make use of the default material.

We can experiment with these by adding the primitive objects and specifying these materials and taking pictures to see how they look.

However, it will be easier to compare once we've seen some basic object transformations that will allow us to put the objects in different locations in our scene.

Fundamental Transformations

Recall that the default primitive objects are always $100 \times 100 \times 100$ and centered at the origin. We need to be able to create models with objects of different sizes, positions, and orientations. These are accomplished by applying *transformations* to our objects.

Translations

We begin with changing positions, which is done with a transformation called a *translation*. A translation is simply the amount we want to move the object in the x, y, and z directions. Let's look at a model that has 4 cubes, each translated in either x or y.

On the Wiki: FourCubes

There are a number of things to think about here. Let's look at the Python statements that add the cubes to the scene:

```
scene.add(cube, translate(100, 0, 0), bluePlaster)
scene.add(cube, translate(-100, 0, 0), redPlaster)
scene.add(cube, translate(0, 100, 0), greenPlaster)
scene.add(cube, translate(0, -100, 0), yellowPlaster)
```

Notice that the transformation to apply to an object when adding it to the scene is specified as an extra parameter to the `add` message. It is an extra piece of information that we can provide to have more control over what happens when the object is added.

The transformation to move an object is called `translate`, and itself takes three parameters: the amount to move in the x, y, and z directions.

When we render the model, we see the four cubes, and several things become apparent:

- a translation of a positive amount in x moved the blue cube to the right
- a translation of a negative amount in x moved the red cube to the left
- a translation of a positive amount in y moved the green cube up
- a translation of a negative amount in y moved the yellow cube down
- we can see the first more convincing evidence of the three dimensional nature of our models

To understand why we see what we see in the scene, we need to remember and think carefully about the position of the camera. Just like in our real three-dimensional world, objects look very different when we look at them from different distances and angles. Remember that unless we specify otherwise (which we will not to quite yet), our camera is positioned at (0, 0, -500), pointing toward the origin. When the camera is in this position, we can think of our computer screen displaying the image as the xy-plane, with negative z values coming out of the screen at us (we are looking from the camera's perspective at z=-500) and positive z values are behind the screen.

Think about the coordinates of each of the cubes. Before the translations, each is a $100 \times 100 \times 100$ cube centered at the origin. So each occupies the space that is between -50 and 50 along each axis. After the translations, each still sits between -50 and 50 in the z-axis. The blue and red cubes are still sitting from -50 to 50 along the y-axis, but now lie from 50 to 150, and -150 to -50, respectively, along the x-axis. The green and yellow cubes remain between -50 and 50 along the x-axis, but are from 50 to 150, and -150 to -50, respectively, along the y-axis.

There are two lines in the model file that are "commented out" (lines start with "#") that we can "uncomment" (remove the "#") to see what it would look like to include cubes translated along the z axis as well. Try it out.

For now, we will be leaving the camera where it is, so we can get used to our translations doing what we expect.

Scaling

Our second fundamental transformation is *scaling*. A scaling transformation changes the size (and we'll see, potentially the position) along each of the three coordinate axes.

We can add a thin, flat red object with a statment like this:

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

Keep in mind that we began with a $100 \times 100 \times 100$ cube, then scale it by a factor of .2 in the x dimension for a "width" of 20, by a factor of 1 in the y dimension so it remains with a "height" of 100, and a factor of .01 in the z dimension for a "depth" of 1.

From the default camera position, it is hard to see that the object we added is so very thin in the z dimension.

As another example, we can make a short, wide cone:

```
scene.add(cone, scale(2, .2, 2), greenPlaster)
```

But what if we want to apply both a scaling and a translation to an object? Ambrosia allows us to do this by performing a *composition* of the transformations. The following will scale a sphere, then translate it:

```
scene.add(sphere, scale(.25, 2, .5)*translate(100, 0, 0), bluePlaster)
```

This results in a $25 \times 200 \times 50$ sphere being translated by 100 along the positive x-axis.

The $\star$ operator between the two transformations specifies the composition. For our purposes, we just think of the composition as one transformation (here, the `scale`) which is performed first, followed by a second transformation (here, the `translate`). Behind the scenes, these transformations are represented by matrices, and the $\star$ operator is actually matrix-matrix multiplication.

We will not worry about the matrices behind the scenes, but one important thing about matrix-matrix multiplication will be important: it is not a commutative operation. That is, the order matters when we multiply matrices. $A \times B$ is not, in general, the equal to $B \times A$. Along those same lines, the order of transformations in a composition (usually) matters. We can see this by reversing the composition above:

```
scene.add(sphere, translate(100, 0, 0)*scale(.25, 2, .5), bluePlaster)
```

Here, we first translate our original $100 \times 100 \times 100$ sphere by 100 along the positive x-axis, then scale it by .25 in x, 2 in y, and .5 in z. When we try this, we see that the squashed and stretched sphere is now located further out along the x-axis!

The important piece of information about scaling that we have not yet considered is that all scaling is done about the origin (*i.e.*, the center of our scene's universe), not about the center of the object. The way to think of this is that for the `scale(.25, 2, .5)` transformation, every point in the object to which it is applied will have its x-coordinate multiplied by .25, its y-coordinate multiplied by 2, and its z-coordinate multiplied by .5. When an object is centered at the origin, this is simply expanding or shrinking the object by the given factor in each dimension. But when it is located elsewhere, it results in the object moving in addition to be scaled!

Thought question: where exactly are the spheres in the above two compositions?

An example showing more scaling and translations:

On the Wiki: ScaleAndTranslate

Rotations

Our next fundamental transformation is the *rotation*.

There are three separate rotation transformations: one for rotating about each of the coordinate axes.

For example, we can rotate a cone about the x-axis:

```
scene.add(cone, xRot(45), purplePlaster)
```

This says to take the default $100 \times 100 \times 100$ default cone centered about the origin, and rotate it positive 45 degrees about the x-axis. Which direction is positive? We'll use a left hand rule again. Using your left hand, put your thumb in the direction of the positive x-axis. Your fingers curl around in the direction of positive rotation. So here, the tip of the cone is rotated back into the screen, and the base comes out toward the camera. Using a negative rotation will point the tip toward us:

```
scene.add(cone, xRot(-30), purplePlaster)
```

Note that the above could also be accomplished with a postive 330 degree rotation.

Rotations about the y-axis of the default cone are pretty boring, so we'll use a cube instead for that one.

```
scene.add(cube, yRot(60), yellowPlaster)
```

To figure out which direction our positive y rotation spins the cube, we again bring out the left hand rule. Point your left thumb up, in the direction of the positive y axis, and your fingers curl in the direction of positive rotation about the y axis. So the face of the cube that was originally facing the camera has been rotated "to the left" and the face that was pointing toward positive x has been rotated so we can see it as well.

Finally, for rotation about the z axis, let's go back to our cone.

```
scene.add(cone, zRot(45), dkGrayPlaster)
```

Again, we can use the left hand rule by pointing your left thumb into the screen (the positive z direction) and your fingers curl in the direction of positive rotation, which from our camera viewpoint is counterclockwise.

We can compose rotations with each other and with our other transformations. Again, order matters. As with scaling, a rotation of an object is not performed about the center of the object, but about the axis around which we are rotating.

Consider the difference in position between these two objects:

```
scene.add(cube,zRot(45)*translate(100,0,0),redPlaster)

scene.add(cube,translate(100,0,0)*zRot(45),bluePlaster)
```

The red cube is first rotated about the z axis, changing its orientation, but not its position, then translated out by 100 units in the positive x direction. The blue cube is translated first, which puts a cube in its original orientation 100 units along the x axis, then rotated. This rotation is still about the z axis, however, so the entire cube rotates around that axis, resulting in some movement in the positive y and negative x directions.

So just like with scaling, we often will perform needed rotations on an object before we translate to avoid unexpected movement. That said, this feature is also sometimes very useful as we will see soon.

A couple of example models that make use of our transformations:

On the Wiki: IceCreamCone

(Next one will be developed in class.)

Basic Lighting and Cameras

Before we wrap up this initial discussion of the basics of Ambrosia modeling, we see how to add lights and move the camera.

Adding Light Sources

The default `scene` has just one light source: a standard white light at position `(0, 300 -300)`.

We can add more light sources, and the simplest way to do so is to add a `bulb` to our `scene` in the same was we added our primitive objects:

```
scene.add(bulb,translate(0,0,-300))
```

This adds a second bulb to the scene at position `(0, 0, -300)`. Note that adding a `bulb` with no transformations applied will place the `bulb` at the origin.

An example with 2 additional bulbs:

On the Wiki: MoreBulbs

This example does something else we have not seen before: generating multiple images by including more than one `camera.shoot()` statement. So yes, we can do that.

The first image lacks much light, as it contains only the default `bulb`, which does not illuminate some of the surfaces we see very effectively.

The second image has a light source between the camera and the origin, so the faces we see have much more light shining upon them.

For the third image, notice in particular that the `bulb` added near the side of the cube before the last image is very evident on that face of the cube.

Moving the Camera

As you remember, the default `camera` in Ambrosia is located at `(0, 0, -500)`, and is pointed at the origin.

We can move the camera to a new position with a statement like:

```
camera.pos([0, 250, -500])
```

This will move the camera to `(0, 250, -500)`, but it will remain "pointed at" the origin. We refer to the location at which the camera is pointing as its *center of interest*.

To change the center of interest, we can use the statement

```
camera.COI([0, 100, 0])
```

There is much more we can do with lighting and cameras, but for now this will be enough to build our initial models.