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

Topic Notes: Introduction to Programming

All of our Ambrosia model building this semester has been *computer programming*: we are teaching the computer how to do something. In our case, we are doing so in a *programming language* called Python.

Let's look a little more carefully at the kinds of things we have been doing in our programs so far.

- The commands we include are called *statements*
- All statements in Python are typically one line long
  - there have been some exceptions in a few examples
- We have seen two types of statements so far
  1. *assignment statements*, which assign some value to a name, *e.g.*:

     ```
     ballHeight = 100
     pacMan = Difference()
     ```

  2. *object manipulation*, which send a message to some existing object, *e.g.*:

     ```
     scene.add(sphere, translate(100, 0, 0), redPlaster)
     camera.shoot()
     ```

- The statements of our program are executed *sequentially* – in the order in which they are encountered in the program.

  A few people have made the mistake of forgetting this from time to time: placing statements out of order.

Repetition

What we have done so far only scratches the surface of what is possible with programming. The first concept we will examine is that of repetition.

People find *repetition* boring. Fortunately, computers don't feel this way. This is fortunate because repetition is the only way we can exploit the full power of a computer. As we discussed in the first class, part of the computer's power comes from the fact that it can follow the instructions stored within its memory rapidly without waiting for a human being to press a button or flip a switch.

In all of the examples we have considered so far, the sequences of instructions are quite short. The computer works for a fraction of a second. We could get the computer to do more work by writing programs with thousands or millions of instructions, but this would be painful.

But we can get the computer to execute thousands or millions of instructions without writing thousands or millions of instructions ourselves: we can have the computer execute the same instructions over and over and over again.

At first, this may seem like a boring and inefficient use of the computer. In fact, when in comes to following instructions, doing the same thing over and over again can be very interesting. Suppose we want to add 50 spheres to our scene. The following does this:

On the Wiki: FiftySpheresPainful

Sure enough, this adds 50 spheres to our scene by executing 50 object manipulation statements, each of which adds a `littleRedSphere` to the `scene`, but with a slightly different translation, so they all end up lined up in a row.

If we were adding 5 or 10 spheres to our scene, this might be a reasonable way to do this. But no one wants to type that same line 50 times with just the one small change from line to line: how far to translate in the x direction.

By using a *loop construct*, in this case Python's `for` loop, we can greatly simplify this program:

On the Wiki: FiftySpheresNotBad

The sequence of 50 statements has been replaced by two parts:

1. an assignment of a name to a *list* of numbers, and
2. a special kind of statement that executes part of its contents multiple times.

The assignment defines the name `xVals` not to be a single number or object as we have seen many times before, but to an entire list of values. These lists are a fundamental construct in the Python language, and have analogs in every other modern programming language. In Python, a list's contents are given in square brackets, and values within the list are separated by commas.

The `for` statement is a brand new construct for us. Here, `xTrans` is a name that will be given, in turn, once to each value in the list `xVals`. The value of `xTrans` is then used inside the indented statement on the line following the `for`. The indented statement, the `scene.add`, will happen 50 times, once for each value in the list `xVals`. The first time through, Python will assign the value -245 to `xTrans`, then -235, -225, and so on, until it gives 245 the last time through. Note that we use that name as the amount to translate in the x direction in the statement.

This construct is one of the most common in programming. We can use it in Python to do many things. For example, consider these Python statements if entered at the interactive Python prompt:

```
>>> nums = [ 12, 23, 8, 17 ]
>>> sum = 0
>>> for n in nums:
...     sum = sum + n
...
>>> sum
60
```

There is still a tedious aspect of the previous example, however. Who wants to type in the 50 numbers that make up the list `xVals`?

Fortunately, Python provides us with ways to generate ranges of values automatically that can be used as the set of values for a `for` loop. Again at the interactive Python prompt,

```
>>> sum = 0
>>> for i in range(10):
...     sum = sum + i
...
>>> sum
45
```

The `range(10)` results in all of the values from 0 to 9 (all integer values starting at 0 up to but not including 10) being used for `i`, once each and in order, in the statement `sum = sum + i`.

If we wanted to add up the first 100 positive integers, we could do this by specifying `range(1,101)`, which means the range of all integers starting at 1, up to but not including 101.

```
>>> sum = 0
>>> for i in range(1,101):
...     print(i)
...     sum = sum + i
...
1
2
3
4
5
(lots of numbers omitted here)
98
99
100
>>> sum
5050
```

Note here that we have two statements that get executed repeatedly inside the loop. In Python, we specify which statements are to be repeated by the loop by indenting them 4 spaces.

But for our example of the 50 red spheres, we want a more general range:

On the Wiki: FiftySpheresGood

The three-parameter form of `range` here says that we want the range of numbers from -245, up to but not including 255, counting up by 10's.

There is also nothing stopping us from using more than one loop in a program. Consider this one:

On the Wiki: TwentyFiveHundredSpheres

The first loop fills up a `Group` with 50 spheres lined up in a row. The second loop adds 50 copies of that `Group` at a range of y positions!

Our loops can be used to control many things, not just positions of objects. Consider this one, where the loop determines how much we are rotating an object:

On the Wiki: CircleOfCones

Now consider this one, where we add 36 cones in a line with a loop that counts from 0 to 35, and use that "cone number" to compute the amount to translate and rotate each cone:

On the Wiki: ManyConesRotated

And there is nothing stopping us from using loops to do things other than add items to a `Group`. In this example, we use a loop to "add" parts to a `Difference` to get poker chips with a series of grooves carved into them.

On the Wiki: PokerChip