CMPSC 390
Visual Computing
Spring 2014
Bob Roos
`http://cs.allegheny.edu/~rroos/cs390s2014`

Review Notes
Visual Python

Major Concepts

working in 3 dimensions, Python basics,

General Python Programming Notes

Special packages can be loaded using an `import` statement, e.g., "`from visual import *`" or "`from random import *`". (There are several variations of this.)

The boolean constants are `True` and `False` (note capitalization).

Indentation is used to indicate the body of loops, "`if`" statements, etc. An improperly indented program might produce an error. Here are some examples of proper indentation. However, you should type in the code to see what each example does!

`from visual import *`
`scene.range=(25,25,25)`
`s = sphere()`
`for i in range(100):`
`  s.x = s.x+.05`
`  s.y = s.y-.2`
`  s.z = s.z+.075`
`  rate(30)`
	`from visual import *`
	`scene.range=(10,10,10)`
	`b = box()`
	`while True:`
	`  if b.x >= 5:`
	`  b.x = -5`
	`  b.x = b.x + .2`
	`  rate(25)`
		`from visual import *`
		`range(10,10,10)`
		`scene.autoscale=False`
		`for i in range(10):`
		`  for j in range(10):`
		`  p = points(pos=[(i,0,j)])`
		`  rate(10)`

A sphere makes 100 steps to the right, down, and forward	A box moves from $(-5, 0, 0)$ to $(5, 0, 0)$, over and over again	Creates a 10-by-10 grid of points in the xz-plane

Other Useful Information:

- Type conversions are done with functions such as "`int(···)`", "`float(···)`", "`str(···)`", etc. As in C and Java, an integer divided by an integer produces an integer.
- The "mod" operator is the same as in C or Java: "`%`"—thus, "`18 % 5`" equals 3.
- Python has a data type called a "tuple" that consists of a sequence of values in parentheses. Thus, we can write things like "`c = (.8,0,.3)`" to create a color. Tuples are fixed in size and can be indexed—e.g., "`c[0]`" is .8 in the example.

- Visual Python includes a `vector` type that is similar to the tuple, except it permits mathematical operations on the entire vector. Things like `pos`, `axis`, etc. are stored internally as `vector`s, not tuples.
- Python includes a "list" data type. of values. Lists are enclosed in square brackets, e.g., "`pts = [p1,p2,p3,p4]`".
- Colors in Visual Python are `float` values between 0.0 and 1.0, not integers between 0 and 255. Thus, $(0, 1, 0)$ means `color.green`.

3D Shapes

Every 3D shape has some properties (the documentation calls them "attributes") that are unique to that shape (so, the `box` shape has a `length`, `width`, and `height`; the `cylinder` shape has a `radius`; an `arrow` has a `shaftwidth`; etc.) However, some attributes are common to all shapes. The `pos` property is a vector indicating the (x, y, z) position of the object (exception: for some objects, like the `curve`, the `pos` property is a *list* of vectors). The `x`, `y`, and `z` properties are the three values in the `pos` vector (again, this is different for some objects, like `curve`).

Most attributes have default values (e.g., `pos` usually has a default value of $(0, 0, 0)$, `color` usually has a default value of $(1, 1, 1)$, or `color.white`, `axis` usually has a default value of $(1, 0, 0)$).

We can set an object's attributes when it is initially created, or add or change them later. For example, the following two segments of Python code do exactly the same thing:

```
s = sphere()
s.pos = (10,15,-3)
s.color = color.red
s.radius = 4
```

```
s = sphere(pos = (10,15,-3), color=color.red,
           radius = 4)
```

An object's attributes can change over time (this is the basic idea behind animation—repeatedly change the `pos` property of an object). The following program creates a box whose color changes constantly (try it yourself!):

```
from visual import *
from random import *

scene.background = (1,1,1)
scene.range = (3,3,3)
b = box(axis=(1,-1,1))

while True:
    b.red = b.red + uniform(-.01, .01)
    b.red = max(0, min(b.red,1)) # must be between 0 and 1

    b.blue = b.blue + uniform(-.01, .01)
    b.blue = max(0, min(b.blue,1)) # must be between 0 and 1
```

```
b.green = b.green + uniform(-.01, .01)
b.green = max(0, min(b.green,1)) # must be between 0 and 1
rate(30)
```

The location of the `pos` value relative to the object depends on the object. For the `sphere` and `box` object, `pos` denotes the geometric center of the object. For objects like `cone`, `cylinder`, `arrow`, and `pyramid`, the `pos` location is at the center of the base of the object.

A number of objects have an `axis` attribute, a vector indicating the principal orientation of the object. For instance, the `axis` of a `box` indicates the direction along which `length` is measured.

Objects that have an `axis` also usually have an `up` attribute, a vector perpendicular to the `axis` that indicates "which way is up."

Figure 1 shows two apparently identical boxes (apart from position and color), but they have different lengths, widths, and heights! Changing the `axis` and `up` vectors changes the definitions of `length`, `width`, and `height`.

```
from visual import *
scene.background=(1,1,1)
scene.range=(30,30,30)

b1 = box(pos = (0,7.5,0), axis=(0,1,0),
       length=15, width=10, height=5,
       up=(0,0,1), color=(0,0,1))

b2 = box(pos = (15,7.5,0), axis=(1,0,0),
       length=10, width=5, height=15,
       up=(0,1,0), color=(0,1,0))
```

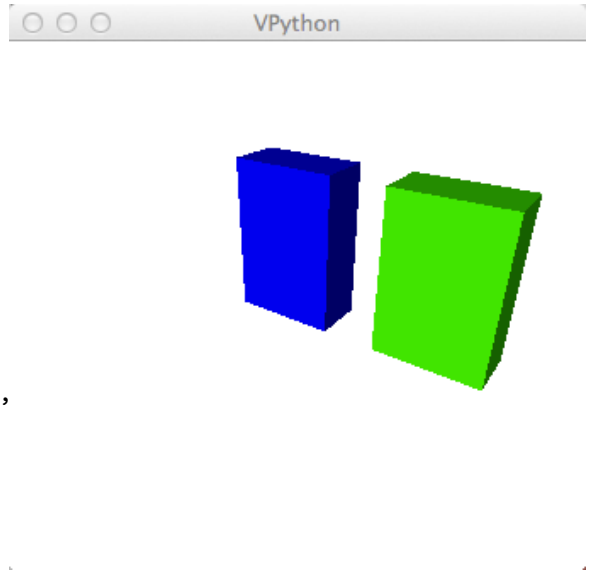

Figure 1: Two Boxes: `axis` and `up` Vectors

The following shapes are the ones you should be the most familiar with:

- `sphere`: know how to set and modify the `pos`, `radius`, `color`, `x`, `y`, and `z` properties.
- `box`: know how to set and modify the `pos`, `axis`, `color`, `length`, `width`, and `height` properties. Know the intuitive meaning of the `up` vector.
- `arrow`: know how to set and modify the `pos`, `axis`, `shaftwidth`, `color`, and `fixedwidth` properties.

For the first exam, you do not need to know any other shapes; you do not need to know anything about the `frame` or `faces` or `curve` objects. You do not need to know about normal vectors or light or "smoothing."

Basic Animation

Animation is achieved by repeatedly changing the attributes of one or more objects inside a loop. We have looked at two main loop structures in Python:

- **The `for`-loop:** The simplest version of this is a counting loop. For example, the loop in the following code moves an object named `s` from position $(0, 2, 0)$ to position $(19.8, 2, 0)$ in 100 moves, shifting the `x`-value by .2 each time:

```
for i in range(100):
    s.pos = (i*.2, 2, 0)
    rate(30)
```

  Note that `range(100)` consists of the set of values $\{0, 1, \ldots, 99\}$. (If you know C or Java, this is the same as `for (i=0; i < 100; i++)`)

- **The `while`-loop:** Here are two examples. The one on the left moves an object `s` until its `y` value exceeds 10. The one on the right continuously jumps object "`s`" back and forth between $(10, 0, 0)$ and $(-10, 0, 0)$.

```
while s.y <= 10:
    s.y = s.y + .1
    rate(30)
```

```
s.pos=(10,0,0)
while True:
    s.x = -s.x
    rate(1)
```

The animation effect is produced by the repeated redrawing of the VPython canvas; we require the `rate` function to tell us how often that should happen. The command "`rate(30)`" means "update the screen 30 times per second," while "`rate(1)`" means "update the screen once per second." This is also sometimes called the *frame rate*. A common frame rate is 30 frames per second—fast enough to keep the animation from looking jerky, but not so fast that we can't process the moving objects. In the second example above, a frame rate of 1 gave a very slow, 1-update-per-second, animation.

The frame rate is one way to control the speed of an animation, but we can also control it by the amount of change per loop iteration. In the first example above, the value of `s.y` changes by .1 each time the loop repeats. Without changing the `rate`, we can speed up the animation by using a value larger than .1, e.g., "`s.y + .2`". Or we can slow it down by using a smaller value, e.g., "`s.y + .01`". This allows us to move different objects at different speeds within the same animation. Try typing in this program and running it:

```
from visual import *
scene.background=(1,1,1)
scene.range=(40,40,40)
s1 = sphere(radius=2,color=color.green)
s2 = sphere(radius=1,color=color.blue,pos=(0,0,2))

# s1 moves at a steady .1 unit per frame
# s2's speed is proportional to s1's, but in the opposite direction.
```

```
# Thus, s2 "speeds up" as the animation progresses.
while s1.x < 50 and s2.x > -30:
    s1.x += .1
    s2.x -= .05 * s1.x
    rate(30)
```

Sample Exam Questions (continued)

9. In Visual Python, write the statements needed to create arrows representing a red x-axis and a blue y-axis. Each axis should start at $(0, 0, 0)$ and be 10 units long. Don't worry about the shaftwidth or other properties.

10. Which of the following program segments causes a `box` named `b` to move "out from the screen" towards the viewer, assuming that the scene has not been rotated and the axes are in standard position?

```
while True:
    b.axis = (0,1,0)
    rate(30)
```
(a)

```
while True:
    b.pos = (0,0,1)
    rate(30)
```
(b)

```
while True:
    b.z += .1
    rate(30)
```
(c)

11. Write the statements needed to create a $1 \times 2 \times 3$ box centered at the origin, with the `axis` pointed along the positive z-axis. (The orientation of the box doesn't matter, i.e., it doesn't matter which side has dimension 1, which side has dimension 2, etc.)

12. **NOTE: You do not need to understand the code for circular motion in order to answer this question!** The following Python code revolves a sphere `s` around the y-axis in a circle:

```
angle = 0
while True:
    s.pos = (3*cos(angle), 0, 3*sin(angle))
    angle = angle + .2
    rate(30)
```

Suppose we have a second sphere, `s2`, and we also want it to revolve around the y-axis, but at half the speed of `s`. What additions or changes must you make to the code above? (Don't worry about sphere radius, distance from the origin, etc.—I'm interested only in the *speed* at which the spheres travel in a circle.)