Vizard 4
Teacher in a Book

Cade McCall

WorldViz

February 2012 printing

Table Of Contents

PREFACE

This book introduces the basics of creating virtual worlds using Vizard. The demos and examples used throughout are essential to its content and can be downloaded from the WorldViz web site at www.worldviz.com/learn. The book is divided up into four sections. The Basic Scripting section provides a brief introduction to writing scripts with Python (the language used by Vizard). The Modeling section introduces users to the essentials of creating and animating dynamic 3D environments. The Hardware section deals with incorporating input from, and sending output to, hardware devices. Finally, the Planning your world section provides suggestions for gathering resources and setting up the sequence of events for large scale programs.

BASIC SCRIPTING

AN INTRODUCTION TO SCRIPTING

In this workbook we'll write scripts using Vizard, a software tool for creating virtual worlds. These scripts will be easier to understand once you've learned a few of the basic coding structures used in Vizard. If you're an experienced programmer and you know a little about the Python scripting language, you can probably skip this chapter. If you're unsure about your programming skills, take the time to go through it. It should give you enough information to start parsing through the example code and completing the exercises in this workbook.

The first thing to know about scripting in Vizard is that it's all done in the Python programming language. Vizard is just one of the many places where people use Python. As an open source language, Python has a large user community and a host of free resources. That fact comes in handy when you're looking for advice or scripting tools. Along those lines, many of the topics we'll cover in this chapter are explored in greater detail in the Python documentation which you can access through Vizard's "Help" drop-down menu ("Python help") or through the Python web site (www.python.org). Python's help documents include an excellent tutorial. Beginner programmers are strongly encouraged to complete sections 3, 4, and 5 of the Python documentation's Tutorial before attempting to write scripts in Vizard.

Vizard provides an interface for writing Python scripts and includes a vast library of VR-related code for you to use when you're creating virtual worlds. Let's start the tutorial by getting acquainted with it. Open up Vizard 4.0 on your computer (probably under "Worldviz" in your Start menu). There's a lot going on in this interface, but for the time-being we're going to focus on the basics. To get a better feel for all that's available in the Vizard interface, check out Vizard's documentation.

Let's get rolling by opening up a new script. To do so, go to the "File" drop-down and select "New Vizard File". When the dialog box pops up, click "Empty File" and then "OK". This will bring a blank Python script up in the middle of your interface. Go to "File" again and save this file. Then start the next section.

CONSTANTS, VARIABLES AND THEIR TYPES

Scripts have a handful of basic building blocks that we'll go over in this chapter. The first, most basic of these basic building blocks are **constants** and **variables**. Constants and variables are names that refer to something in your script (e.g. a number). Once you've defined a constant or a variable in your code, that name is a memory handle that you can use whenever you need it. The difference between constants and

variables is that you might change the value of your variables over the course of the program (a given variable might equal 1 at the beginning of the program and 100 later) while a constant will stay the same.

This following chunk of code defines a constant (MY_CONSTANT) and a variable (my_variable) and then prints their values. All the lines that begin with the # sign are just **comments**. Python recognizes these as comments and will not run them as code. You can use comments for notes or for preventing specific lines of code from being executed. Add this code to your script and then run it by hitting the "run" button at the top of the interface.

```
#Define a constant and a variable.
MY_CONSTANT = 'whatever'
my_variable = 1

#Print their values.
print MY_CONSTANT
print my_variable

#Change the value of the variable.
my_variable = my_variable + 1

#Print the value of the constant
#and the variable.
print MY_CONSTANT
print my_variable
```

The print statements here tell Vizard to print the values of the constant and variable in the input/output window in the interface. After you've hit run, check the input/output window for the four values. If you didn't type the code in correctly, you may get an error in that window. Hopefully the error will give you enough information to help you figure out what you did wrong. Whenever you run a script and it isn't working, the first thing to do is to check this window for error messages. (Note: the line at the bottom of the input/output window is a handy place to try individual lines of code to see what they do.)

You probably noticed that the variable above was in lower case while the constant was in upper case. That convention is used sometimes, but it's not necessary. However, Python *is* case sensitive. So make sure you're consistent with each constant or variable's name.

You can't refer to a variable or a constant until you've defined it. If you put an undefined variable or constant name in a script, you will get an error that will prevent your script from running correctly. Add a print statement to your code to add an undefined

variable (we'll call it "new_variable") so that you script looks something like the following:

```
#Define a constant and a variable.
MY_CONSTANT = 'whatever'
my_variable = 1

#Print their values.
print MY_CONSTANT
print my_variable
print new_variable

#Change the value of the variable.
my_variable = my_variable + 1

#Print the value of the constant
#and the variable.
print MY_CONSTANT
print my_variable
```

Now run your script again and check out the input/output window for the error message. Let's fix the problem we created by defining our new_variable before we refer to it in the script.

```
#Define a constant and a variable.
MY_CONSTANT = 'whatever'
my_variable = 1
new_variable = 'another variable'
#Print their values.
print MY_CONSTANT
print my_variable
print new_variable
#Change the value of the variable.
my_variable = my_variable + 1
#Print the value of the constant
#and the variable.
print MY_CONSTANT
print my_variable
```

As you can tell from the script that we just wrote, variables (and constants) can be of a variety of different types. The types you will most often see in this workbook are numbers, strings, lists and dictionaries. **Strings** are simply text values. In the example we went over above, MY_CONSTANT was a string type. **Lists** or **arrays** are ordered sets of values. Once you've defined a list, you can refer to its elements by

referring to its index, a number which identifies where the element is in the list. The first element in a list is index 0, the second index 1 and etc. Start a new script and add the following code.

```
#Define an array.
my_array = ['a','b',1,2]

#Print the value of the second
#element (index 1).
print my_array[1]
```

A **dictionary** is like a list, but it has no order to it. Instead, values in a dictionary are identified by keys. In that sense, they are analogous to a physical dictionary where you look up a word (the key) to find its definition (the value). The following code defines a dictionary and then refers to an element in it.

```
#Define a dictionary with two keys
#(and two values
#associated with those keys).
my_dictionary = {'Idaho':'Boise', 'Michigan': 'Lansing' }

#Call find the value for the key
#'Idaho'.
print my_dictionary[ 'Idaho' ]
```

There are a variety of ways that lists and strings can be spliced, appended, and otherwise edited. Before you go on, it's a good idea to check out the Python documentation on different types to see how this can be done. The best place to start is by going to "Python Help" under Vizard's "Help" drop-down menu. Go to "Tutorial" on the Python Documentation page. Within the Python Tutorial, go through section 3: "An Informal Introduction to Python".

SIMPLE LOGIC

The basic flow logic of the scripts we write in this workbook relies on a handful of different statement types. We'll briefly go over these types in this section, but to get a strong foundation in writing the flow logic of a script, go back to the Python Tutorial ("Python Help" under Vizard's "Help" drop-down menu) and go through sections 3.2 ("First Steps Towards Programming") and sections 4.1 through 4.5 (in "More Control Flow Tools"). These tutorials provide an easy way to get quickly acquainted with some of the most basic components of the code you will use when making virtual worlds with Vizard.

Throughout the examples in this text, we use **"while"**, **"for"** and **"if"** statements. Both "while" and "for" are **loop statements**. The contents of loop statements are repeated. The indented area under a "while" statement repeats as long as the given condition is true (i.e. "while this condition is true, run this loop"). Start a new script and add the following code, making sure that the lines under the "while" statement are indented:

```
my_variable = 0

while my_variable < 5:
   print my_variable
   my_variable = my_variable + 1
```

Run the script. This code should have produced a list of numbers in the input/output window. Notice that the statement looped through the indented code until the condition (my_variable < 5) was no longer true.

IMPORTANT: Indentation is critical to scripting in Python. The snippet of code you just added will only work if the lines inside the loop are indented (try removing the indentation from the last lines and you will get an error). Indentation is just as critical for the other statements we'll talk about in this section and in all the example code in this workbook.

A "for" statement is like a "while" statement except that it cycles through every element of a list or an array (i.e. "for every element in this array, do the following"). Add the following code to your script:

```
#Define an array.
my_array = ['a','b','c']

#Loop through every element in
#my_array.
for element in my_array:
   print 'current element is', element
```

This snippet of code went through the loop once for every member of the list "my_array". For each member's loop, "element" was set to the value of that member. Go back to that snippet of code and replace "my_array" with "range(5)" so that it looks like this:

```
#Define an array.
my_array = range( 5 )

#Loop through every element in my_array.
for element in my_array:
   print 'current element is', element
```

The **range** function that we're using here creates an array of numbers of the requested size (in this case, 5). Unless otherwise specified, range will create an array that starts at 0 and counts upwards until it's full. If this description isn't clear, try typing "range(5)" into the command line at the bottom of the input/output window. The input/output window is a handy place to check individual lines of code.

An "if" statement tests for a given condition. If the condition is true, the code underneath the statement is run. If not, it isn't. If you couple an "if" statement with an "else" statement, then when the "if" statement is false, the code under the "else" statement will be run. Try running the following code to see how this works:

```
for element in range(5):
   print 'current element is', element
   if element == 4:
      print 'all done'
   else:
      print 'wait,there is more'
```

FUNCTIONS

Imagine you had an equation that you wanted to use repeatedly in your script. You could write out the entire equation every time you need to use it. That, however, would be quite tedious if you used the equation 100 times throughout your script. Instead, you could write a **function** to handle that equation every time you need to use it. If you pretend your script is like an office, a function is like an employee within that office who's an expert in a specific area. When you need help in that area, you call that expert, tell him your question, and he responds with an answer. Similarly, when you need a function, you **call** it. When you call it, you can **pass** the function **arguments** and they'll **return** values.

To create a function, you **define** it. Once you've defined a function, you can use it as many times as you want. Let's start by defining a very simple function. This function will be one that multiplies whatever number we've passed it (the argument) by 20, and then return that value. We define the function using a def statement:

```
#Define a function named
#"multiply_by_twenty".
#This function requires
#an argument, "number".
def multiply_by_twenty( number ):
   answer = number*20
   return answer
```

A function doesn't do anything until you call it. So, we'll add some code that calls our function for each member in an array. Run the script once you've added this last snippet.

```
for i in [1,5,23]:
   #Call the function.
   print multiply_by_twenty( i )
```

One important concept to keep in mind when you're working with functions is the difference between **global** and **local** variables. Conceptually, global variables are known throughout the script while local variables are only known within a given function. In other words, a function has access to both global variables and its own local variables but it does not have access to other functions' local variables. In terms of the office metaphor, local variables are things that only the expert knows while global variables are things that everyone in the office knows. In other words, a function's local variables stay within that function. Try running the following block of code. The "worker_bee" function uses a global variable and returns a string.

```
#Define a global variable.
season = 'spring'

#Define a function
def worker_bee():
   if season == 'spring':
      return 'honey'
   else:
      return 'I got nothing'

#Call the function and store
#the returned value as "bee_response".
bee_response = worker_bee()
print bee_response
```

The global/local distinction is important because if you try to call local variables within the main body of the script (or within another function), you will get an error. Try running this snippet of code:

```
def fruitless():
   apple = 'hello'

fruitless()
print apple
```

Because "apple" was defined inside the function it's local to that function and when we refer to it outside of the function, the main script doesn't know what we're talking

about. If we want this variable defined globally, we'll need to specify that it's a global variable at the beginning of our function:

```
def fruitless():
   global apple
   apple = 'hello'

fruitless()
print apple
```

Along these lines, it's important to remember that although you can access global variables within a function, any changes you make to that variable will stay within the function unless you specify that the variable is global within the function. So, try running the block of code below. Notice that although the function can access the global variable "value", the changes that it makes to that variable do not affect the global variable.

```
value = 'hello'

def change_variable():
   value = 'goodbye'
   print 'local ', value

change_variable()
print 'global ',value
```

If, however, we use the global statement at the beginning of our function, the changes we make to the variable within our function will be made global:

```
value = 'hello'

def change_variable():
   global value
   value = 'goodbye'
   print 'local ', value

change_variable()

print 'global ',value
```

Note: the local variables within a function do not change if the function is run repeatedly. Each call of the function is independent.

CLASSES

Classes are another handy programming structure. Once you've defined a class within your script, you can call it like a function. When you call a class in your script, you create an **instance** of that class called an **object**. As an analogy, you can imagine the class as a car factory and the objects instantiated by the class as cars coming from that factory. Once the car is made, it has all the capabilities that the factory gave it, but it's an entirely different thing than the factory. In the same sense, you can create multiple objects from one class but once the objects are made, they're their own entity.

In the code below, we first define a class and then call it twice. After that we change one of the variables within one of the objects and then print out the variables from both objects to show that they are different (you can refer to variables within an object using *[the object's name].[variable name]*).

```
#Define a class.
class most_basic:
   whatever = 'original value'

#Use the class to instantiate
#two objects.
one_object = most_basic()
another_object = most_basic()

#Change the variable
#in one of the objects.
one_object.whatever = 'changed value'

#Now print out the variable
#from each object. Notice that
#they have different values
#(we changed "whatever" in
#one but not the other).
print one_object.whatever
print another_object.whatever
```

Classes can have functions within them, sometimes called **methods** (technically, any function within a class is a method). Once you've instantiated an object, you can call any of its methods, referring to the method in the same way that we referred to an object's variable above, *[the object's name].[the method name]*. In the code below, we define a class that has a method, then we create an object with the class. Finally, we call one of the object's method.

```
#Define the class.
class make_car:
   def peel_out( self ):
   #This function can
   #be called from within
   #or outside of the
   #class. It will print
   #somthing and change the
   #object's moving variable.
   print 'vroom'
   print 'vroooom'
   print 'VROOOOOOOOOM'

#Instantiate an object using
#the make_car class.
my_car = make_car()

#Call a function in the object.
my_car.peel_out()
```

One specific kind of method that you can use in a class is an **initialization** method for that class. This method is executed whenever you instantiate an object with this class. So, you can use this method to define variables for an object when it's first created. This method also accepts any arguments you provide when you first call the class. You distinguish this method from other methods in its name: "def __init__(self, *[any arguments you pass the class]*):".

The global/local distinction is important when you're writing a class. Within a class, you need to use the prefix "self" to refer to variables and methods within that same class (technically, "self" refers to the specific object created with that class). In terms of variables, you need to use "self." at the beginning of a variable name if you want to refer to that variable elsewhere in the class. By putting "self." in front of the variable name, you make that variable global to the class, otherwise that variable will be local and only accessible to the method in which it's created.

Along similar lines, we also use "self" as the first argument of all the methods within our class. When you call a method within a class and that method belongs to that same class, use the "self." prefix.

Once you've used a class to instantiate an object, you can refer to the "self" variables and methods of a class by replacing "self" with the name of that specific object.

If this sounds confusing, walk slowly through the next example. The code below defines a class. The first method of this class is an instantiation method that defines

the class's "moving" variable. We put "self." in front of that variable to make it global to that class so that we can refer to it later within the "peel_out" method of this class. After defining the class, we use it to instantiate an object and then call the object's "peel_out" method and refer to its "moving" variable. Note that when we call the "peel_out" method from outside the class, we don't pass it a "self" argument.

```
#Define the class.
class make_car:
   def __init__( self ):
      #Initialize the class.
      #This method will be called
      #when you first instantiate
      #an object.
      self.moving = False

   def peel_out( self ):
      #This method can be called
      #from within or outside
      #of the class. It will print
      #somthing and change the
      #object's moving variable.
      print 'vroom'
      print 'vroooom'
      print 'VROOOOOOOOOM'
      self.moving = True

#Instantiate an object using
#the make_car class.
my_car = make_car()

#Call a method in the object.
my_car.peel_out()
print my_car.moving
```

One final thing to understand about classes is **inheritance**. A class can inherit the variables and methods of other classes, gaining all the abilities of those other classes. Codewise, you include these base classes in parentheses after the class name when you're defining it. In the example below, we define the new class "driver", using the "make_car" class as a base class. Then we use one of the inherited methods ("peel_out") within the new class.

```
#Define the class.
class make_car:
   def __init__( self ):
      #Initialize the class.
```

```
      #This method will be called
      #when you first instantiate
      #an object.
      self.moving = False

   def peel_out( self ):
      #This method can be called
      #from within or outside
      #of the class. It will
      #print somthing and change
      #the object's moving variable.
      print 'vroom'
      print 'vroooom'
      print 'VROOOOOOOOOM'
      self.moving = True

class driver( make_car ):
   def leave_town( self ):
      print 'I am out of here'
      #Call an inherited method.
      self.peel_out()

#Instantiate an object using the
#driver class.
my_driver = driver()

#Call a method in the object.
my_driver.leave_town()
#Refer to one of my_driver's
#inherited variables.
print my_driver.moving
```

Classes can inherit from multiple classes, although you need to be mindful that these base classes may sometimes contain methods or variables of the same name.

For more information about classes, check out section 9 of the Python documentation's Tutorial section (go to Python Help under the Vizard Help menu).

MODULES

Modules are accessory scripts that we use in Python to import functionality into our scripts. So, for example, Python has a module called "math" that provides a host of functions that handle mathematical issues. When you import math into your script,

your script **inherits** all the functions within that math module. Try adding and running the following code:

```
#Import the math module.
import math

#Define a function that uses a
#method in the math module.
def get_square_root( number ):
   return math.sqrt( number )

#Call that function and print
#the returned value.
print get_square_root( 16 )
```

The main reason we use software like Vizard is because it provides modules of commands that help render virtual worlds. In this workbook we'll use Vizard's **viz module** in almost all of our scripts. This module contains a library of functions and classes that are useful in creating virtual worlds. The code below uses the go function of the viz module to open up a graphics window and render a virtual world with some objects in it. This example also uses the module's add function to render a 3D model and to return a **node3d object**. This node3D object provides a slew of methods that you can use to manipulate the 3D model. Here we'll use the "setPosition" node3d object to set the position of the objects we add.

```
import viz

#Use the viz module's go
#function to render a 3D
#world in a graphics
#window.
viz.go()

#Use a for loop to . . .
for i in range( 5 ):
   #Use the viz module
   #to add a 3D model.
   #The viz.add method
   #will return a node3d object.
   ball = viz.add( 'white_ball.wrl' )
   #Use a node3D method
   #to place the object
   #in the world.
   ball.setPosition(i*.2,1.8,3)
```

The fastest way to see the contents of Vizard's modules is to check out the **Vizard Command Index** (under the Help drop-down menu). This list groups methods together based on the objects or modules that they're a part of. For a description of this structure, check out "Vizard Library Structure" in the Vizard help documentation.

To check out the code for Vizard's modules, look in the python folder in Vizard's directory. To check out the contents of other more generic Python modules, go to Python Help under the Help drop-down and then go to the "Global Module Index".

EVENTS

Interactivity is essential to virtual worlds. The output of a system changes continually as users navigate around environments and as the components of that environment must both respond to user input and to their own simulated interactions. Programming interactivity presents a unique set of challenges. A particularly handy tool in dealing with these challenges are **callbacks** and **events**.

A callback is like a watchman and an event is like an alarm. When you issue a callback in your script, you're telling the program to watch for a specific event to happen. When that event happens, the watchman sets off the alarm and calls a specific function associated with that event. You can issue callbacks for many different kinds of events-- anything from timers ("watch for this amount of time to pass") to keyboard presses ("watch for this specific key to be pressed") to intersections between models in the virtual world ("watch for model A to intersect with model B"). When the given event occurs, the callback will call a specified function and may feed that function specific data about the event.

Let's say, for example, that you are writing a program in which you don't want a world to appear until the user hits the 'a' key. To do so, you could make the world invisible and then issue a callback that watches and waits for the user to hit the right key. When that happens, you can call a function to make the world appear. Open a new Vizard script and add the following lines:

```
#Add a model.
court = viz.add('court.ive')
#Make it invisible.
court.visible( viz.OFF )
```

These lines will add the model of a room and make that model invisible. Now we will add a callback that watches for a keypress and a function for that callback to call when that event happens. In the following code we write the function first and then the callback second:

```
#Write the function that will be called.
def onKeyDown(key):
   #If the key that was pressed is the
   #a key, then make the court visible.
   if key == 'a':
       court.visible( viz.ON )
#Issue a callback for keyboard events.
#When those events occur, call the
#onKeyDown function.
viz.callback(viz.KEYDOWN_EVENT,onKeyDown)
```

Run the script and trying pressing the 'a' key. Notice that the world only appears if you hit the right key. In this particular snippet of code, we used a global callback. You can also use callbacks from the vizact module. The following code does the same thing we just did above, but with a somewhat simpler format:

```
#This command comes from the vizact library.
#We give the key to wait for, the function to
#call, and the argument to pass that function.
vizact.onkeydown( 'a', court.visible, viz.ON )
```

Instead of using a keypress now, let's use a timer event to trigger the appearance of the world. Timer events go off when a specified amount of time has passed. Once again we'll use the timer event callback from the vizact module.

```
#This command will wait for 1 second
#to pass and then it will call the
#court.visible function with the viz.ON
#argument. The 0 here means that the
#timer will only go off once.
vizact.ontimer2( 1, 0, court.visible, viz.ON )
```

For more information on events in Vizard, check out the "Event Basics" and "Event Reference" sections in the Vizard Help Documentation. There you'll find the full range of different events that you can use in your code. You can also find information about how to make your own events.

3D NAVIGATION MODES

See "3D NAVIGATION MODES" on page 17There are two main ways to naviSee "3D NAVIGATION MODES" on page 17gate around the 3D worlds that we will construct in this workbook. Vizard’s default navigation mode uses the mouse.

Open the script named "using actions example.py" in the Vizard interface and hit "run". Now, try playing with the mouse. Mouse navigation works like an automobile. You set the mouse pointer to the middle of the render screen and then press and hold the LEFT mouse button. To move forward, ease the cursor upward - the farther up the faster you'll go. To move backward, ease the pointer downward - again adjust speed by the distance you move the pointer from the center of the screen. To steer, push the pointer to the left or right. Holding down the RIGHT mouse button instead can be used to translate the viewpoint directly sideways or up and down. Holding down both the LEFT and RIGHT mouse buttons simultaneously as you move the mouse up and down let's you pitch forward or backward. Holding down both the LEFT and RIGHT mouse buttons simultaneously as you move the mouse to the left or right let's you roll the viewpoint to either side (pitch is rotation about the local X axis; roll is rotation about the local Z axis). At first it may be a little difficult to move around, but with practice you'll get used to it.

The other navigation mode we use in this workbook is pivot navigation. Pivot navigation is used in many of the demos. Run "transformations demo.py" now to get a feel for it. This style of navigation pivots around a center point. To pivot, hold down the LEFT mouse button and move the mouse around. To move up, down, or sideways, hold the RIGHT mouse button down and move in the desired direction. To zoom in on the center, use the mouse's scrolling wheel.

AN EXAMPLE

See "AN EXAMPLE" on page 18See "AN EXAMPLE" on page 18See "AN EXAMPLE" on page 18Now let's look at an example of some of the basic scripting tactics we've gone over See "AN EXAMPLE" on page 18in this chapter. This example will use a lot of code that you're not familiar with yet. Don't worry about the details-- just try and get a feel for the overall logic of the script.

The first thing we'll do in this script is import the viz module. Most every script in this workbook is going to import the viz module because it holds many of the functions and classes that we'll use to create virtual worlds. On the next line, we'll use the function "go" from the viz module to open up a **graphics window,** the window in which you'll see the world.

```
import viz
viz.go()
```

After adding these lines, try running the script to see the graphics window. It will just be black for the time-being since we haven't added anything to the world.

Next, we'll use "add" from the viz module to add a 3D model of a balloon to the world. We pass the "add" function the name and location of the 3D file we want to use: "balloon.ive" from the "art" folder in this script's directory. Note that it's critical for you to save this script in your "Teacher in a Book Demos and Examples" directory because that directory has the "art" folder which contains the balloon model. If you don't save your script in that directory, it's not going to find the balloon and you'll get an error when you try running the script.

```
#Add a model.
balloon = viz.add( 'art/balloon.ive' )
```

Calling this function will create a **node3d object.** That's a kind of Vizard object that has a slew of methods we can use to manipulate the 3D model we've added. We've given the object the variable name "balloon" so we can use that to refer to the object when we call its functions in the future. One of the first functions we'll call is the "setPosition" function to set the position of the 3D model in the world. Add these lines to your script and then run it:

```
#Set the position of the balloon.
balloon.setPosition( 0, 1.8, 1  )
```

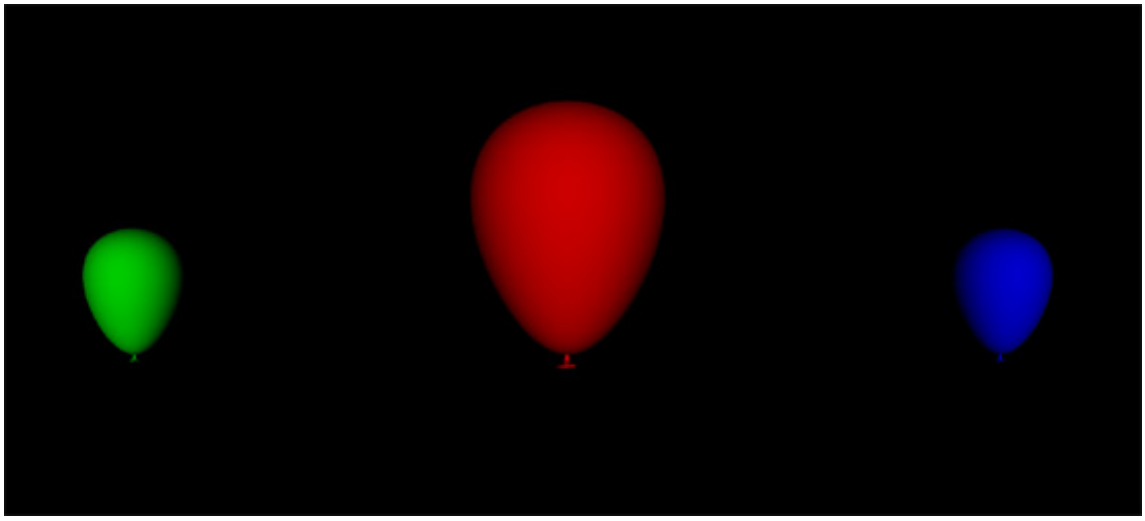
Now we'll add two more balloons. We could do so by repeating the lines that we just added. It'll be more efficient, though, to use a loop since we want to repeat the same lines. Each time we go through the loop we'll we add a balloon and we'll append it to an array called "balloons". So, replace the lines you just added with the following lines and run the script:

```
#Add an empty array for the balloons.
balloons = []

#Use a loop to add 3 balloons to the world.
for i in [-1,0,1]:
   #Add a model.
   balloon = viz.add( 'art/balloon.ive' )
   #Set the position of the balloon.
   balloon.setPosition( i, 1.8, 3  )

   #Append the balloon to our array.
   balloons.append( balloon )
```

Notice that the balls appear across the screen in different places. That's because we used "i" when we set the position of each balloon. Since each balloon had a different "i" (either -1,0, or1), they each ended up in different places.

Next we'll change each balloon's color separately. We'll do this by referring to the different indices of the "balloons" array (from 0 to 2). We use "color", which is one of the node3d object's methods. We pass that function an argument to set the color. That argument is in the form of a **flag**. Flags are just the names of standard constants that Vizard uses. The basic colors all have a flag in Vizard.

```
#Change each balloon's color.
balloons[ 0 ].color( viz.GREEN )
balloons[ 1 ].color( viz.RED )
balloons[ 2 ].color( viz.BLUE )
```



Finally, we'll define our own function to inflate the balloons. We'll make this function so that it accepts a balloon object (referred to within the function as "who"). Inside the function we'll create an **action**-- an animation strategy that you'll learn more about in the *Modeling action* section of this workbook. After we've created that action, we'll add it to whatever balloon gets passed to the function.

```
#Define a function.
def inflate( who ):
   #Define the inflating action.
   inflate_animation = vizact.size(2,2,2)
   #Add the action to the node.
   who.addAction( inflate_animation )
```

Now we've got to call this function to make it work. Here we'll call it for the middle balloon.

```
#Call our function and pass it a balloon.
inflate( balloons[1] )
```

Run the script and see if it works and then add these last few lines. These lines will issue a callback for a keyboard event so that one of the other balloons will get inflated when you hit the 'b' key.

```
#Use a keyboard event to blow one of the
#other balloons.
vizact.onkeydown( 'b', inflate, balloons[0] )
```

EXERCISES

1. Change the script that we just created so that you add 6 balloons to the world. NOTE: you might have to use your mouse to navigate backwards to see all the balloons.

2. Add a line that changes the balloon to purple when it inflates (use the flag viz.PURPLE) and use that function to inflate all of the balloons.

VIZARD TIPS

When programming with Vizard, you'll frequently need to search through the contents of the Vizard library. The Vizard Command Index (under the Help menu in the Vizard interface) lays out all the commands available to you. These commands are grouped by kind such that commands pertaining to viewpoints or lights or avatars bones are grouped together. The entries in this index tell you what kind of arguments these commands require and what they will return.

For more in depth content, head to the Vizard Manual (under the Help menu as "Vizard Help"). If you look here under the Contents tab, you'll find a section called "Reference". This section is split up by topic areas and each topic has a "Basics" page which should help get you started on that subject. These pages also contain links that will route you to other parts of the manual that might answer your question. Finally, the Vizard Manual has a section on Tutorials and Examples that can be a helpful source of example code.

MODELING

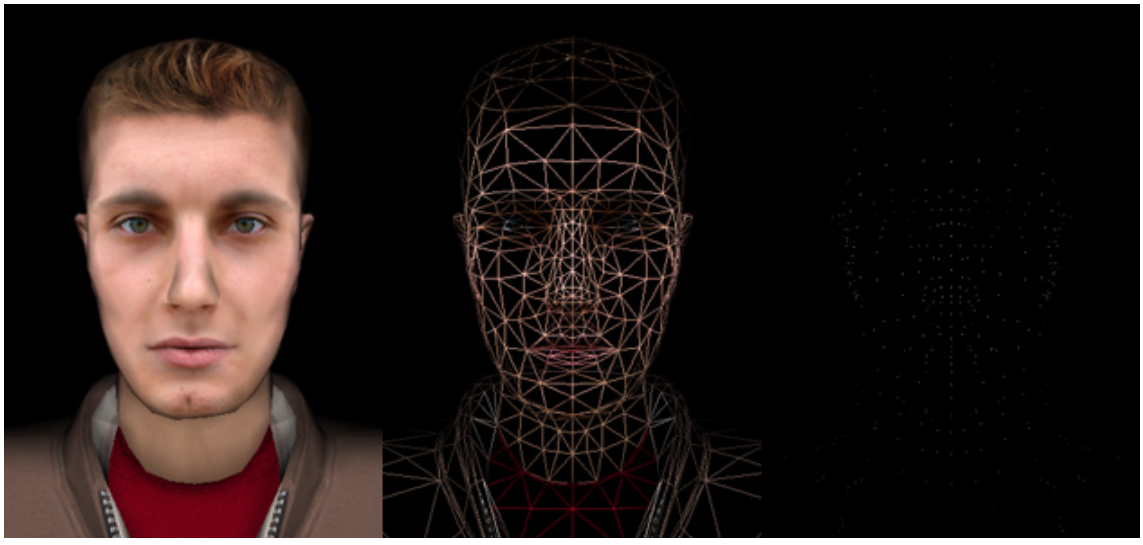
GEOMETRIC MODELING

Virtual worlds are filled with 3D models simulating everything from the solar system to human beings. In this section we'll do a quick overview of the basic components of See "MODELING" on page 23these virtual objects. See "MODELING" on page 23

VERTICES, LINES AND POLYGONS

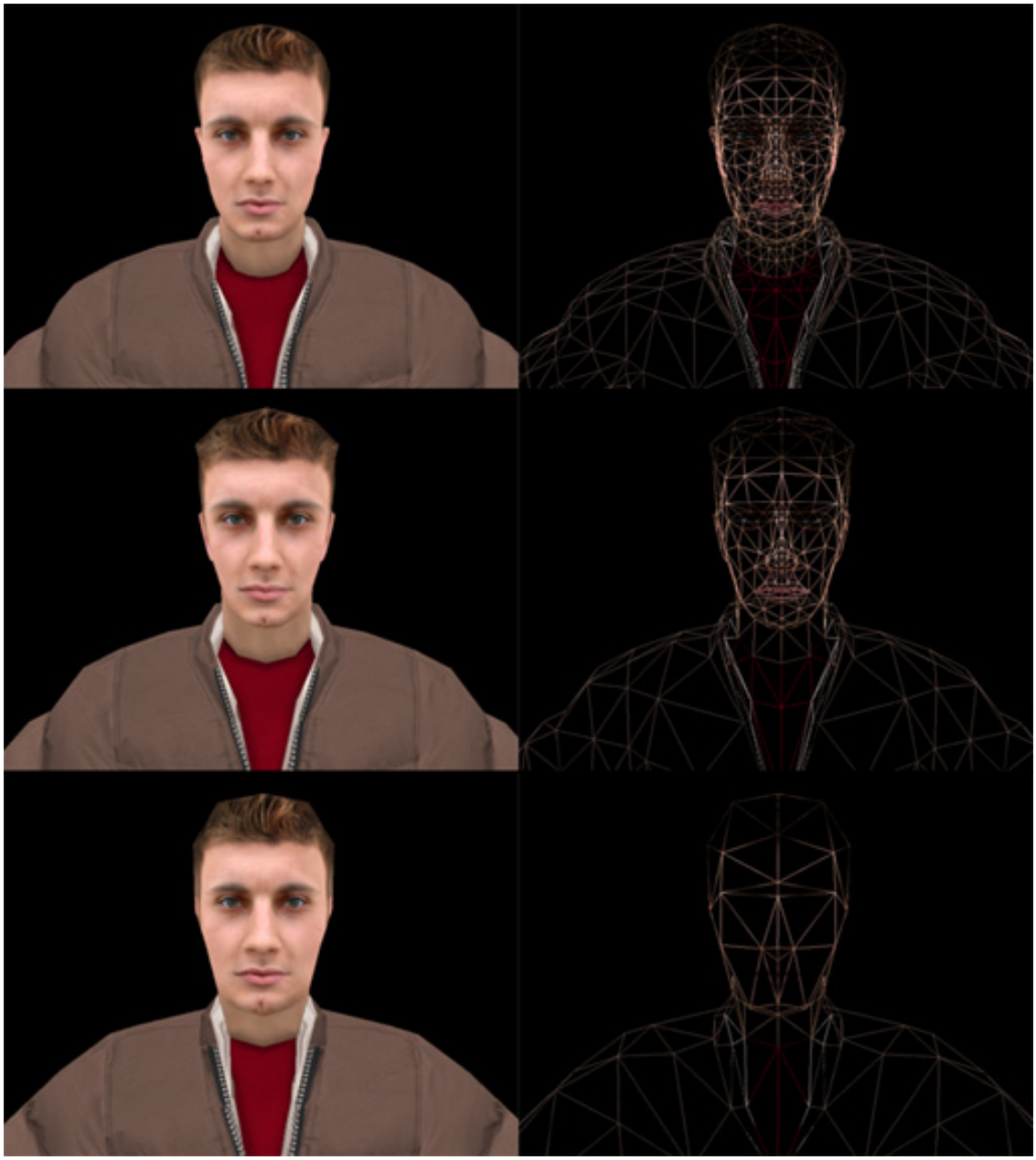
You can think of virtual objects as a collection of 3D surfaces. Although they may appear and behave like solid objects in a virtual world, it’s more accurate to think of them as hollow, wire-framed structures with materials wrapped around them. The 3D surfaces that compose an object are defined by a series of **vertices** or points in 3D space. These vertices are used to define shapes, called **polygons**, that make up an object's surface. The polygons of a model come together in a mesh that creates the contours of the object. A **texture** is then applied to that surface to give the surface a more realistic appearance.

To check out the different parts of a model, run the geometric modeling demo. Once the scene loads, you'll see three textured models. Hit F3 and you'll see the mesh of polygons that make up the models. Hit F3 again and you'll see the vertices that define those polygons.



As you can imagine, the more polygons a model has, the more intricate its surface can be. If you check out the demo again, notice that as you move from right to left across the three models, they increase in **polygon count,** the number of polygons in the model. This increase in polygons improves the surface detail such that the models

have a more organic appearance. Given that, it might seem obvious that you'd want to use models with high polygon counts in your virtual worlds. The problem with using high polygon models, however, is that it's costly for processing. Each polygon that gets rendered during a given frame uses up some of the processing power of your system. As a consequence, a scene with an extremely high polygon count will have a slow **frame rate** (the number of frames rendered per second). But if we want our world to be interactive in real time, we need it to render quickly. As such, we've got to balance the benefits of higher polygon counts with processing costs when choosing 3D models. Fortunately, we have a few tricks to tip the balance in our favor (e.g. elaborate textures, level of detail modifiers). Moreover, graphics cards continue to improve in efficiency, allowing us larger and larger polygon counts.



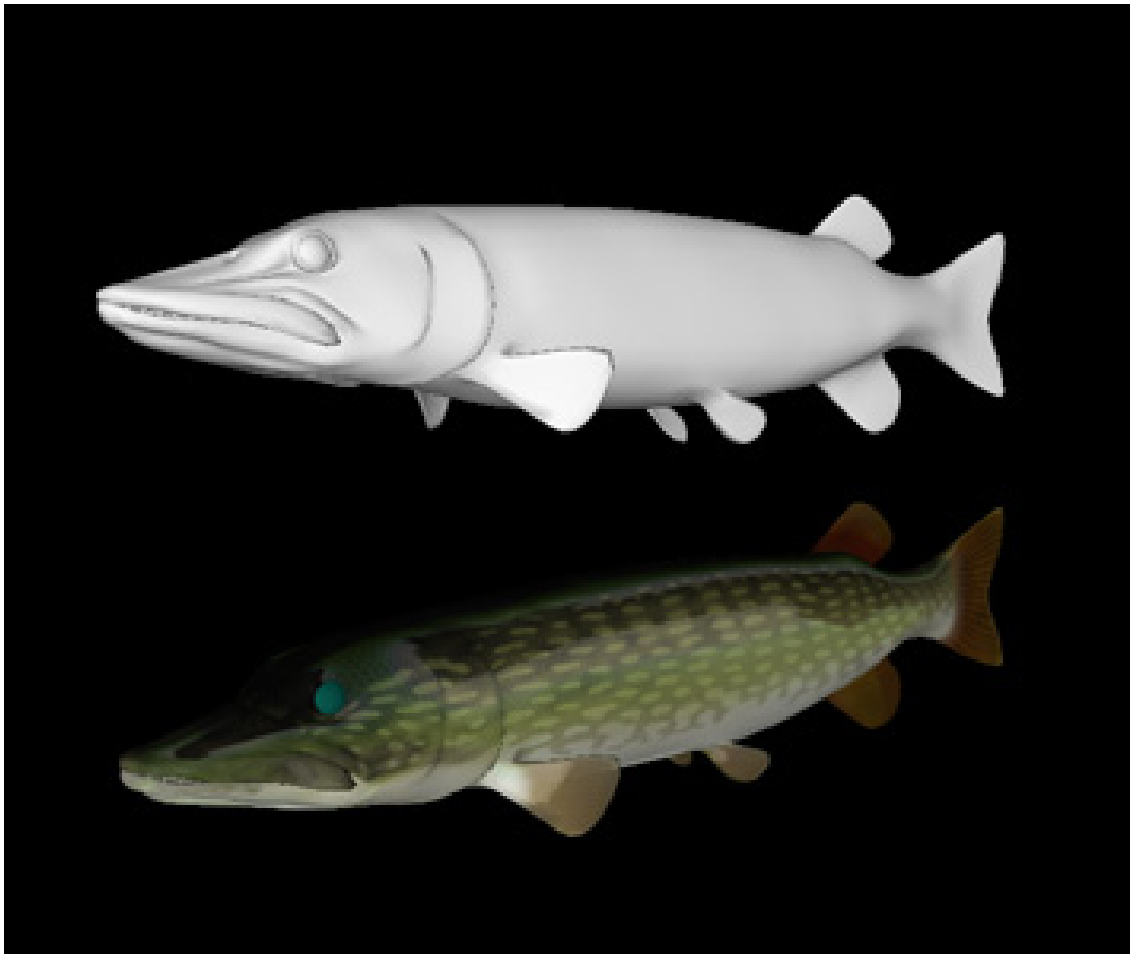
Most of the time when you’re creating virtual environments you’ll use models that are built in software specially designed for creating 3D objects (3DS Max, Maya, etc.). When it comes time to add objects to your worlds, you’ll just import them. Note: you can, however, create models within scenes "on the fly". This can be handy if you're creating a model that's going to change shape over the course of an interaction (e.g. a

fishing line that's going to be cast and then reeled in). For more on creating objects on-the-fly, check out the Vizard documentation.

TEXTURES

As we covered in the Geometric modeling section, 3D models are just groups of polygons defined by a series of vertices. Imagine you wanted to create a model of a brick. If you wanted to illustrate the variegated colors and depths on a brick's surface, you'd need to use a tremendous number of polygons. That many polygons, in addition to being a whopping pain to model in the first place, would gobble up your system's processing power when you tried to use them in an interactive world in real-time.

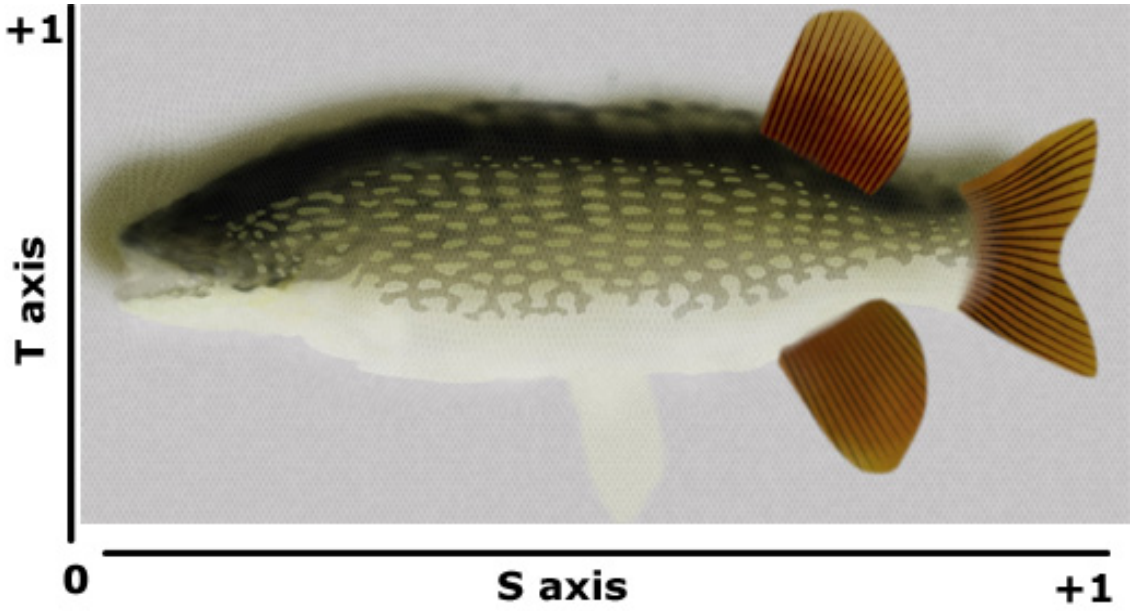
See "TEXTURES" on page 26To get around this predicament, we use textures. Textures are images that are "wrapped" on 3D model geometry, adding color and pattern to those surfaces. Textures add a dramatic amount of detail to models without additional polygons, so they're an integral part of realistic looking models. Also, textures are used to display 2D media, such as paintings and video, within the virtual world.



TEXTURE COORDINATES

Textures are usually 2D image files (e.g. jpegs or tif files). In order for these 2D files to appear in 3D space, they need a surface to **wrap** onto. (In the examples in this section we'll be wrapping textures on texture quads which are simply a plane in 3D space.)

To determine how 2D files should be wrapped on 3D surfaces, we use **texture coordinates**. The texture coordinates of a 3D surface define where any texture should be applied to that surface. Texture coordinates have two axes: the S axis which refers to the width of an image, and the T axis which refers to the height. Regardless of texture or model scale, texture coordinates range from 0 to 1.0 in both dimensions with (0, 0) at the bottom left.



Each point on the surface of a 3D model is assigned an S,T - coordinate so that when a texture is wrapped onto that model, the color of the pixel at point S,T on the texture will be used to color that point on the model. For example, let's say I have a 3D model of a head and the S,T - coordinate assigned to the tip of the nose is (.5,.5). When I wrap a texture on that model, the model will take the color at point S,T on the texture and use that color to paint the tip of the nose.

In order to apply a texture to a model, the model needs S,T coordinates. Modeling software (e.g. 3DSMax) calls the process of wrapping geometry in textures S-T mapping or U-V mapping.

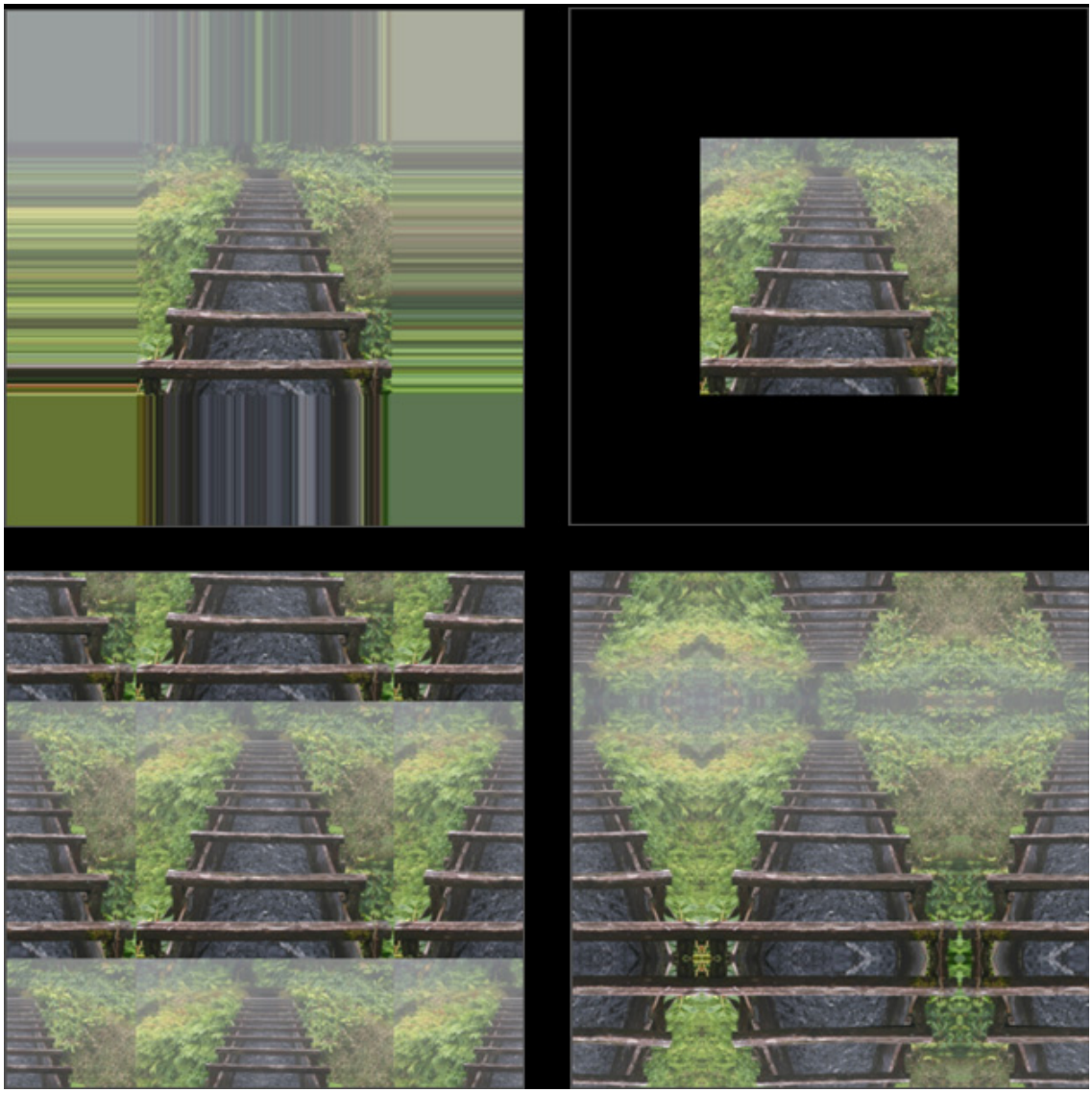
To get a feel for how texture coordinates work, run the texture coordinates demo. In this program, a texture has been applied to a texture quad (a 3D model that's simply a 3D plane). When the program first loads, the texture quad's texture coordinates are assigned such that a square texture maps perfectly onto the quad. The 0,0 point on an image falls on the bottom left of the quad, the 1,1 point on the image falls on the upper right of the quad, and etc. In this demo, however, you can alter the quad's texture coordinates by changing their scale or by translating their position across the surface. Try going to the "texture coordinates" menu and changing the scale of S and T coordinates to "2". This change will double the texture coordinates in both dimensions and, as a consequence, the image will only fill half of the surface in each dimension. (At first it might seem counterintuitive that doubling texture coordinates would shrink the image, but remember that the surface's texture coordinates are what we're

doubling, not the image size. So, the point that was originally assigned to (.5,.5) in the image is now assigned to (1,1) in the image.)

Along the same lines, you can also change the position of the model's texture coordinates by translating them. Try changing translating the texture coordinates of the model by setting the translation of both S and T coordinates to "-.5. This translation should move the image up and to the right. (If this again seems counterintuitive, consider that the point on the surface assigned to (1,1) will be assigned to (.5,.5) after the translation of the surface's S-T coordinates

As you can see in this demo, texture coordinates can be assigned in such a way that the texture does not cover the entire object. **Texture wrap modes** specify how the rest of the object (where the texture coordinates fall outside of the 0 to 1 range) should be textured. One option for wrapping is to repeat the same texture over the surface of the object. Mirroring also repeats the texture, although with mirroring the texture is reflected for each iteration across the given dimension. If you don't want to repeat a texture, you can **clamp** the texture and use either the border color or the pixels at the very edge of the texture to color the rest of the object.

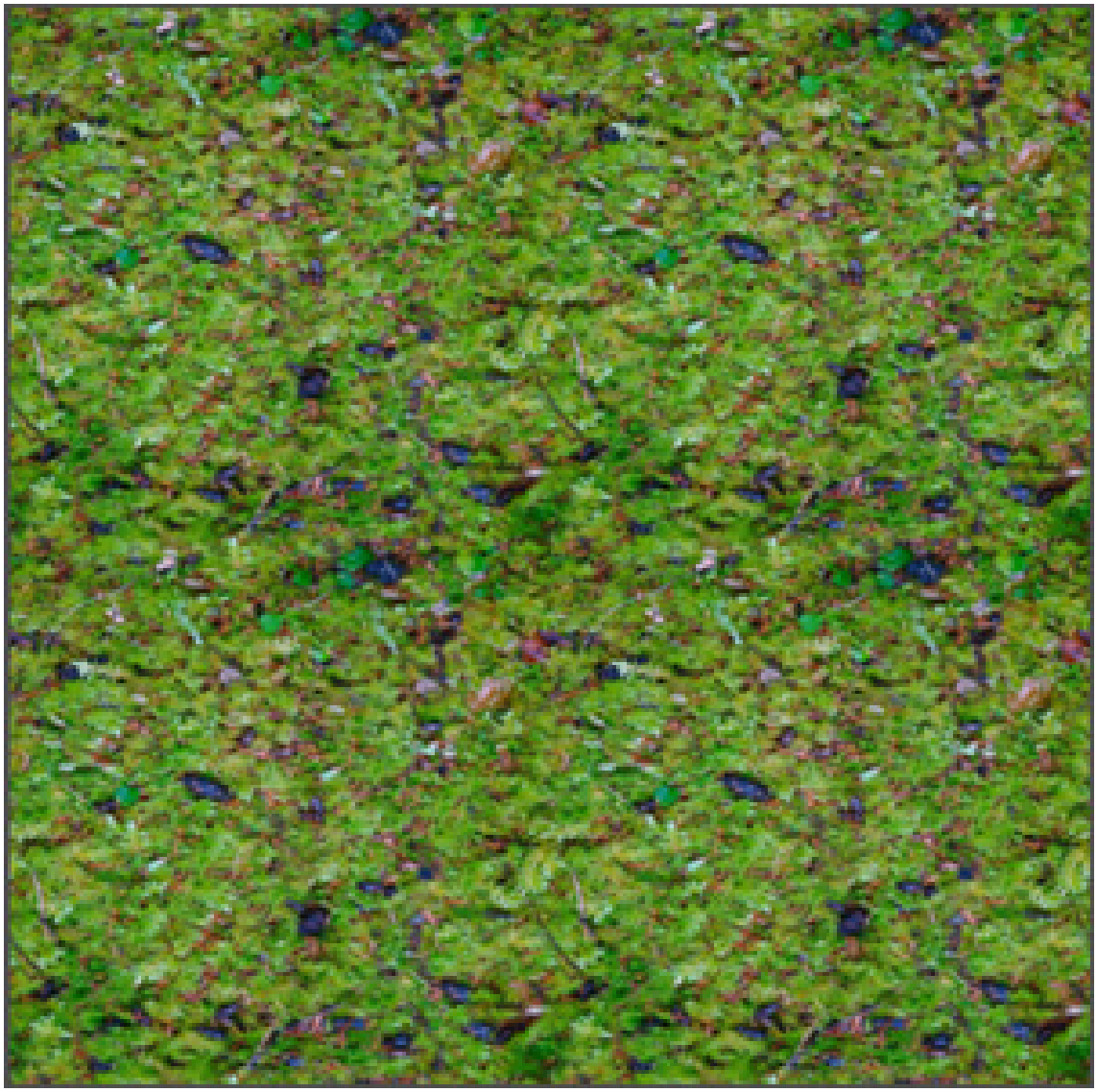
Run the texture coordinates demo and after scaling the texture coordinates so that the texture does not fill the entire quad, go under "wrap modes" and play with the different options. If you clamp to the border, try changing the border color with the "border color" menu.



When it comes to repeating textures across the surface of a model, it's important to think about whether or not the texture you're using is **tileable**. A texture is tileable when it's similar enough on it's edges that when you place a number of copies of that texture next to each other, it's difficult to tell where one ends and the other one begins.

Run the texture coordinates demo and again scale the texture coordinates to that the texture does not fit the quad. Set the wrap mode to "REPEAT" and go to the "blend" menu and move the bar all the way to the right. Notice how it's hard to distinguish

where this texture begins and ends. A texture like this one would work well on a large model with a lot of tiling, such as a model for the ground.



MULTITEXTURING

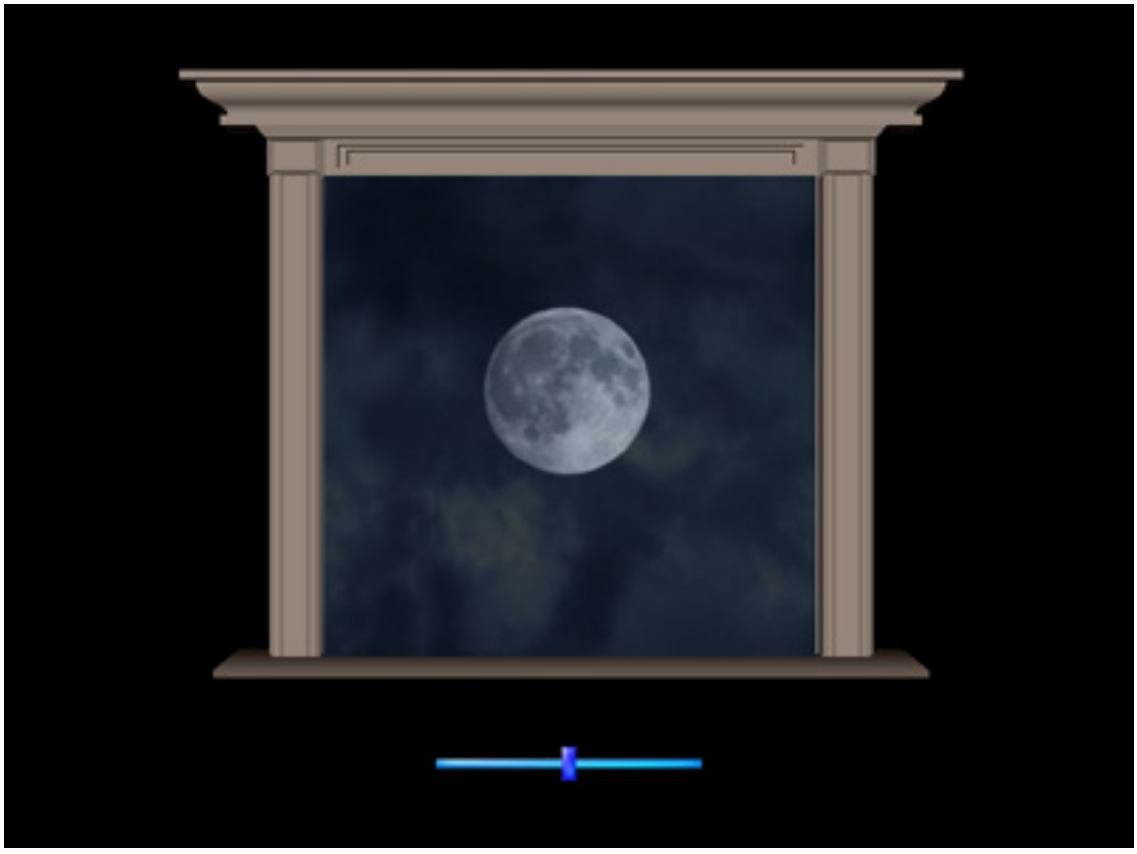
With **multitexturing**, surfaces can have more than one texture. Multitexturing can give a surface a more complex appearance by adding light patterns or the simulation of surface dynamics. You can also blend between textures to simulate a changing

appearance (such as a leaf turning color). 3D modeling software products have a host of options for applying multiple textures and defining the relationships between them (bump-maps, alpha maps, etc.).

The "blend" menu in the texture coordinates demo made use of multitexturing. As you move from the original texture to the more tileable texture with the sliding bar, you're changing the degree to which each texture is being used to color the texture quad.

EXAMPLE SCRIPT: APPLYING TEXTURES

In this example script, we will take a piece of a model and then apply a couple textures to it, and manipulate the texture coordinates for those textures. Open your own script for this example, and as we go through pieces of code, add them to your script to see their effects.



First we import the viz library and begin rendering a scene with the go command. After that, we take the main viewpoint and place it such that we'll be able to look at the scene when it loads. Next we add a model to the scene and grab one of it's chil-

dren, a **sub-node** called "glass". Both of these commands will return node3d objects (that we've labelled "model" and "window") that we can use to control these models.

```
import viz
viz.go()

#Put the viewpoint in a good place to
#see the texture quad.
viz.MainView.setPosition( 0, .68,-1.3 )
#Add a model and grab one of its
#children to put the textures on.
model = viz.add('art/window.ive')
window = model.getChild( 'glass' )
```

Next we'll add the texture files with the viz library's "addTexture" command. These two files are jpegs in our art directory.

```
#Add the texture files.
clouds = viz.addTexture('art/tileclouds.jpg')
moon = viz.addTexture('art/full moon.jpg')
```

Now we'll apply these textures to our window with the node3D "texture" command. This command allows you to add a given texture to the model and allows you to specify which unit the texture will go in. Specifying the unit is important when you're applying multiple textures to a model.

```
#Apply the textures to the window.
#The moon will go in the window's
#first unit(0) and the clouds will
#go in the second (1).
window.texture( clouds, '', 1 )
window.texture( moon, '', 0 )
```

To control the ways in which the cloud texture wraps, we'll specify a wrap mode with the wrap command which accepts two flags-- one for the dimension (S or T) and the other for the wrap mode itself. Here we'll use the repeating texture mode.

```
#Set the wrapping mode for the clouds
#to be REPEAT so that as the surface's
#texture matrix translates, there will
#still be texture to show.
clouds.wrap( viz.WRAP_S, viz.REPEAT )
clouds.wrap( viz.WRAP_T, viz.REPEAT )
```

Next we're going add a slider device to the scene so that the user can interact with it. Specifically, the user will be able to push the slider back and forth to control the blend

of the two textures on the model. First we add a slider object (a node3D gui object) and set its position on the screen. Then we add a function that will be called every time the user changes the slider. Codewise, we're setting up a slide event here with the onslider command from the vizact library. When the slider event occurs, it will call our swap_textures function and tell that function where the slider has been moved to. Our swap_textures function will take the slider's position and blend in the clouds (the texture in unit 1 of the model) anywhere from 0 (none) to 1 (the object is 100% textured by the clouds texture).

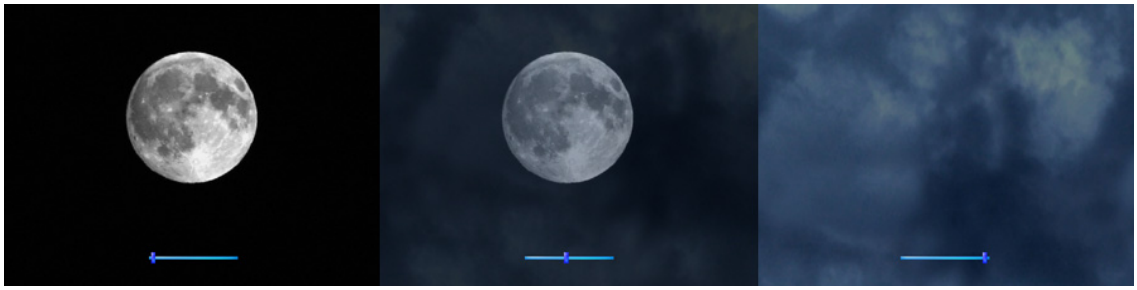
```
#Add a slider and put it on
#the bottom of the screen.
slider = viz.addSlider()
slider.setPosition(.5,.1)
#This function will be called
#every time the slider
#is moved and will swap the
#textures according to the
#slider's position.
def swap_textures( slider_position ):
   #Use the slider's position to get
   #the amount of cloud blend.
   cloud_amt =  slider_position
   #Blend the clouds (unit #1) in that amount.
   window.texblend( cloud_amt, '', 1 )
#Set up the slider event to call our function.
vizact.onslider( slider, swap_textures )
#Set the initial blend to match the slider.
swap_textures(0)
```

Run the script now to see how that looks. Try playing with the slider and backing up so that you can see the entire model.

Next we're going to manipulate the window's texture coordinates to move the clouds across the surface. To do this, we first create a transform matrix using the vizmat library. This matrix will include all the information about our transform of the texture coordinates including the scale and translation. Then we'll use a timer to call the move_clouds command (timers are discussed in more detail in the Modeling action section). The move_clouds command applies a translation to our transform matrix with the "postTrans" command. It then applies that matrix to unit 1 (the cloud texture) of our window's texture coordinates with the node3d texmat command.

```
#Create a matrix that we'll use to
#the surface's texture matrix.
matrix = vizmat.Transform()
#This function will move the clouds incrementally.
```

```
def move_clouds():
   #Post translate the matrix (move it in the s and t dimensions).
   matrix.postTrans(.0005,.0005,0)
   #Apply it to the surface.
   window.texmat( matrix, '', 1 )
#Run the timer every hundredth of a second.
vizact.ontimer( .01, move_clouds )
```



Now run the script again and you should see the clouds moving when you slide the slider to the right.

EXERCISES

1. Add the "art/wall.ive" model to a world, add the "art/stone wall.jpg" texture and apply it to that model.

2. Change the scale of box's texture coordinates for the unit that the stone wall texture is in. Then repeatedly tile the texture across the box.

3. Add the "art/pine needles.jpg" to another unit of the wall model from the previous exercise and blend the two textures so that the wall looks like the picture below.

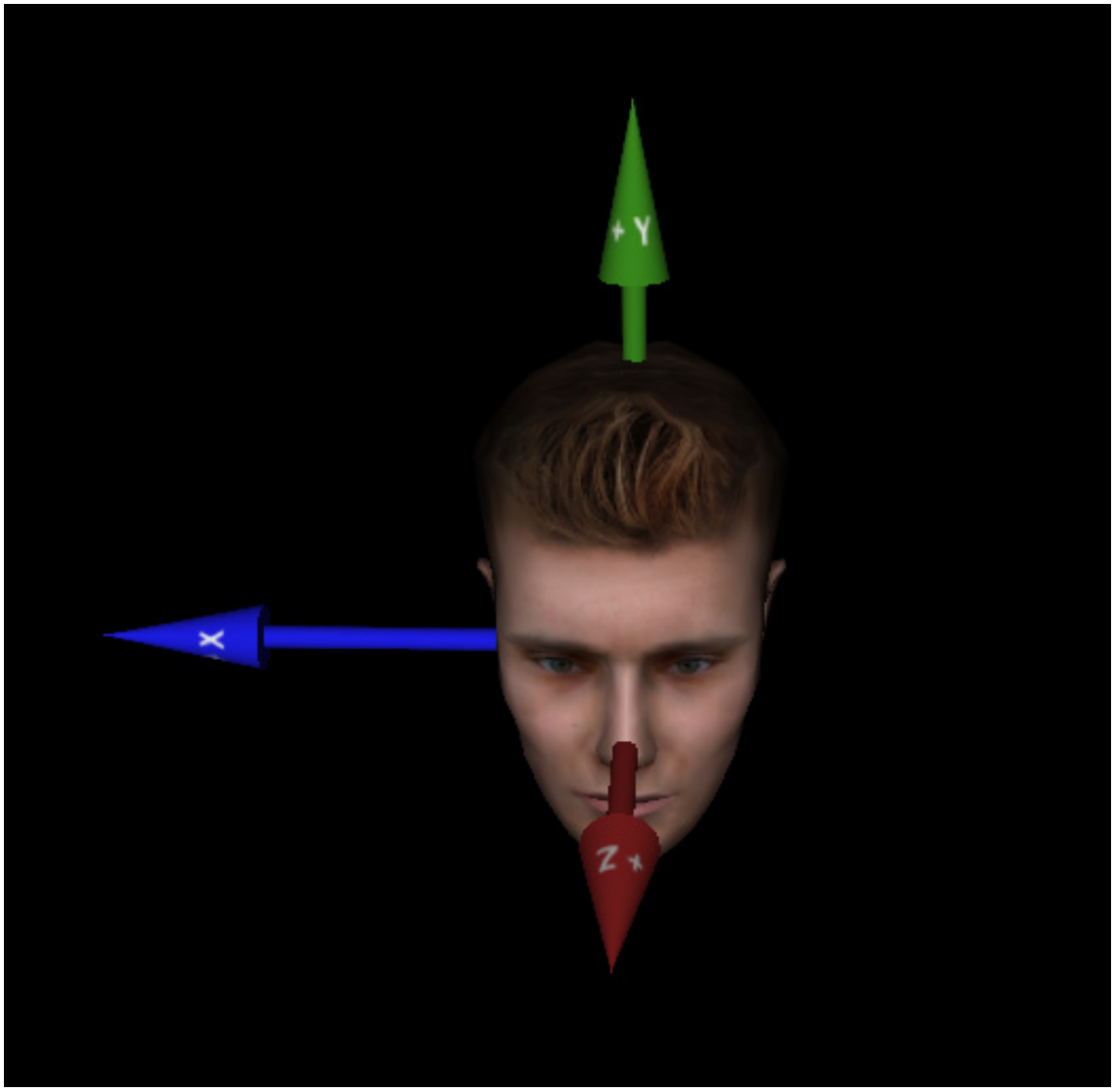
THE SCENE GRAPH

The **scene graph** defines the basic structure of a virtual world. A scene graph is a hierarchy of **nodes** that include all the objects in a world along with the viewpoints from which the world is rendered. Nodes in the scene graph can be changed using 3D **See "THE SCENE GRAPH" on page 36transformations** (or **transforms**). These transformations affect the position, orientation, and scale of a node. In this section we'll go over the basics of how 3D See "THE SCENE GRAPH" on page 36trans-formations are applied to objects and the scene graph as a whole.

TRANSFORMATIONS

When you're working with 3D spaces, it's critical to have a basic understanding of 3D geometry. If you're already clear about 3D geometry, you can breeze through this section. Otherwise, take a moment to get oriented.

3D space is defined by [x, y, z] coordinate systems. The x, y, and z axes of a coordinate system come together at the **origin**. In the Vizard coordinate system, the x and z axes define the horizontal plane and the y axis defines the vertical. If you were to put your head at the origin of these axes, the +z axis would be coming out of your nose, the +x axis would be coming out of your right ear and the +y axis would be coming out the top of your head. (Note: not all 3D modeling software uses these definitions-- some define horizontal plane with the x and y axes and the vertical with the z axis).



We use [x,y,z] coordinates to define transformations of position, orientation and scale. If you want to move an object up one meter, you position it at [0,1,0]. If you want to move the object an additional meter to the right, we position it at [1,1,0]. Scale factors are also defined in [x,y,z] dimensions such that scaling an object in a given dimension will increase the size of the object along that dimension. To make an object twice its height, we would apply a scale of [1,2,1].

Rotational definitions also use the [x, y, z] axes. If you imagine, again, that your head is at the center of a coordinate system, a positive rotation around the x axis would rotate you forward. This kind of rotation is called **pitch**. A positive rotation about the z

axis would tilt you over onto your right side. This kind of rotation is called **roll**. A positive rotation about the y axis would turn you around to your right. This kind of rotation is called **yaw**. There are a few different ways in which these rotations are specified mathematically. One way is to give the axis around which an object is rotated (x, y,or z) and then the degree to rotate. Another way is to use **eulers**. A rotation using eulers takes a specified yaw, pitch and roll and then applies them in *that* order to the object. The important thing to remember about using eulers is that order matters: [yaw, pitch, roll]. The third way to specify rotations is with **quaternions**. A quaternion is a 4-Dimensional vector that can be thought of as a radius vector to a point on a 4-Dimensional unit sphere.

Together the position, scale, and orientation of a node are included in a matrix of numbers called the **transformation matrix**. When you apply a transform to a node, you are changing the transformation matrix of that node. Oftentimes, you will not deal with the transformation matrices of your nodes directly, but it's important to know that these matrices are behind the tranforms you're applying.

Run the transformations demo to get a feel for how transforms effect a model. Try hitting each of the buttons to see an animation of a transformation along that dimension.

THE SCENE GRAPH AND COORDINATE SYSTEMS

Virtual objects are nodes in the hierarchical structure of a scene graph. The scene graph has a **scene root** from which all of the nodes branch off. Each of the branches in the scene graph can have multiple nodes on it. As you travel down a branch, the node below another node in the hierarchy is a **child** of the **parent** node that's above it. If a node doesn't have any node above it besides the scene root, then the scene root is that node's parent.

Each node in a scene graph has its own **local coordinate system**. The coordinate system of the node's parent is the **parent coordinate system** and the coordinate system of the root node is the **global coordinate system**. If a node is directly under the scene root, then it's parent coordinate system *is* the global coordinate system.

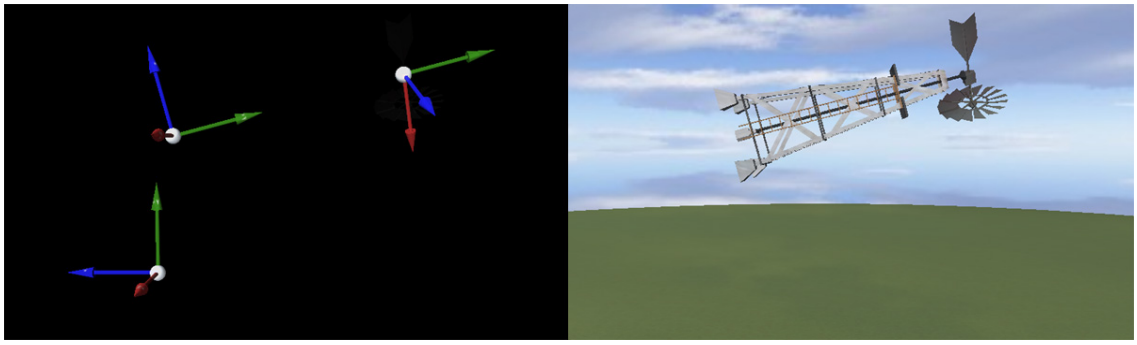
There are two important things to remember about transforming a node: 1) transforms are applied within a specific coordinate system, and 2) nodes inherit the transforms of their parents. For example, let's say you're a node and I translate you 1 meter up [0,1,0] in your parent's coordinate system. Now let's say I translate your parent 2 meters up [0,2,0] in the global coordinate system. Because you inherit your parent's transforms you will now be 3 meters up in the global coordinate system [0,3,0], although you're still just 1 meter up in your parent's coordinate system. If that's too abstract, think of it this way: your parents move 50 miles to get away from

New York (the global coordinate system), then you move an additional 50 miles in the same direction to get away from your parents (the parent coordinate system). Now you're 100 miles from New York (the global coordinate system).

In terms of transformation matrices, a node has its local transformation matrix which is combined with the parent's transformation matrix to determine the position, scale and orientation of the node in the scene.

Note: a node's local coordinate system follows wherever it goes. A node will always be located (or oriented) at [0,0,0] within its local coordinate system. You can apply a transform with reference to that coordinate system, but once the transform is applied, the coordinate system assumes the new position and orientation as its origin.

One more thing to know about transformations is that they can either can be applied **relative** to the object's current position/orientation/scale or in **absolute** terms. So, let's say you're a node and I apply a 10 degree rotation to you about the y-axis of the global coordinate system. If this rotation is in absolute terms, then you will rotate to 10 degrees in that system. If I apply the rotation again, you'll stay there because you've already reached 10 degrees in absolute terms for that system. If, however, I apply the rotation relatively, then it'll rotate you relative to wherever you are when I apply the transform. So if I apply the transform again, you will continue to rotate about that axis another 10 degrees.



To get a feel for how transforms are applied at different levels of the scene graph, run the coordinate systems demo. In this demo, the parent is the tower and the child is the turbine placed atop the tower. Use the menu at the top of the screen to select which object you want to transform, the coordinate system you want to use in your transformations, whether you want the transform to be relative or absolute, and the axis you want to perform the transform on. To see a schematic of the coordinate systems, go to "show coordinate systems". When you're ready to apply the transform, hit

one of the buttons on the bottom of the screen ("rotate", "translate", or "scale") to see how your transform affects the scene.

EXAMPLE: TRANSFORMATIONS AND THE SCENE GRAPH

Now we'll work through an example script where we add several models to the scene graph and then apply transforms to them.

Open a Vizard script and add some models with viz library's "add" command. This command will return a "node3D" object that we will use to control the model.

```
#First add a few models to the world.
ground = viz.add('art/sphere_ground3.ive')
rod = viz.add('art/rod.ive')
fish = viz.add('art/pike.ive')
barrel = viz.add('art/barrel.ive')
avatar = viz.add('vcc_male.cfg')
```

Next, we'll make the background blue and put the viewpoint in a convenient place.

```
#Give the background a blue hue.
viz.clearcolor( [ .5, .6, 1] )
#Place the viewpoint so we can see everything.
viz.MainView.setPosition(-7,1.5,.33)
viz.MainView.setEuler(90,0,0)
```

If you run the script now you'll see a clump of models, all at the origin of the global coordinate system. Let's get things arranged, first by making the fish a child of the fishing rod and then adjusting its position and orientation in the rod's coordinate system. The node3d "parent" command will place a node3d object (in this case our fish) in the scene graph under the specified node (the rod). To get the fish in the right place with regards to the rod, we set its position and euler rotation within the rod's coordinate system (the parent coordinate system) with the node3d setPosition and setEuler commands. As mentioned above, eulers are specified as [yaw, pitch, roll].

```
#Make the fish a child node of the rod.
fish.parent( rod )
#Position and orient the fish on the rod.
fish.setPosition( [.06,.04,2.28], viz.ABS_PARENT )
fish.setEuler( [0,-45,180], viz.ABS_PARENT )
```

Now let's scale the fish in it's own coordinate system to make it a little larger.

```
fish.setScale( [1,1,1.2], viz.ABS_LOCAL )
```

Next, we'll move the rod in the global coordinate system so that it appears to be leaning against the barrel. While we're at it, let's move the avatar out of the way. Notice that we didn't put any flag in the setPosition function for the avatar. That's because viz.ABS_PARENT is the default and that'll work for our purposes.

```
#Now move the rod up so that it's leaning against the barrel.
rod.setEuler( [0.0,-50.0,0.0], viz.ABS_GLOBAL )
rod.setPosition([-0.04,0.13,-1.14],, viz.ABS_GLOBAL)
#Move the fisherman.
avatar.setPosition(2,0,0)
```

Now let's add some turmoil to our bucolic scene. First we'll add some bees as a child of the avatar within the scene hierarchy. Then we'll position those bees so they're at head height.

```
#Add bees as a child node of the fisherman.
bees = avatar.add('art/bees.ive')
#Place the bees at head level for the avatar.
bees.setPosition( [0,1.8,0], viz.ABS_PARENT)
```

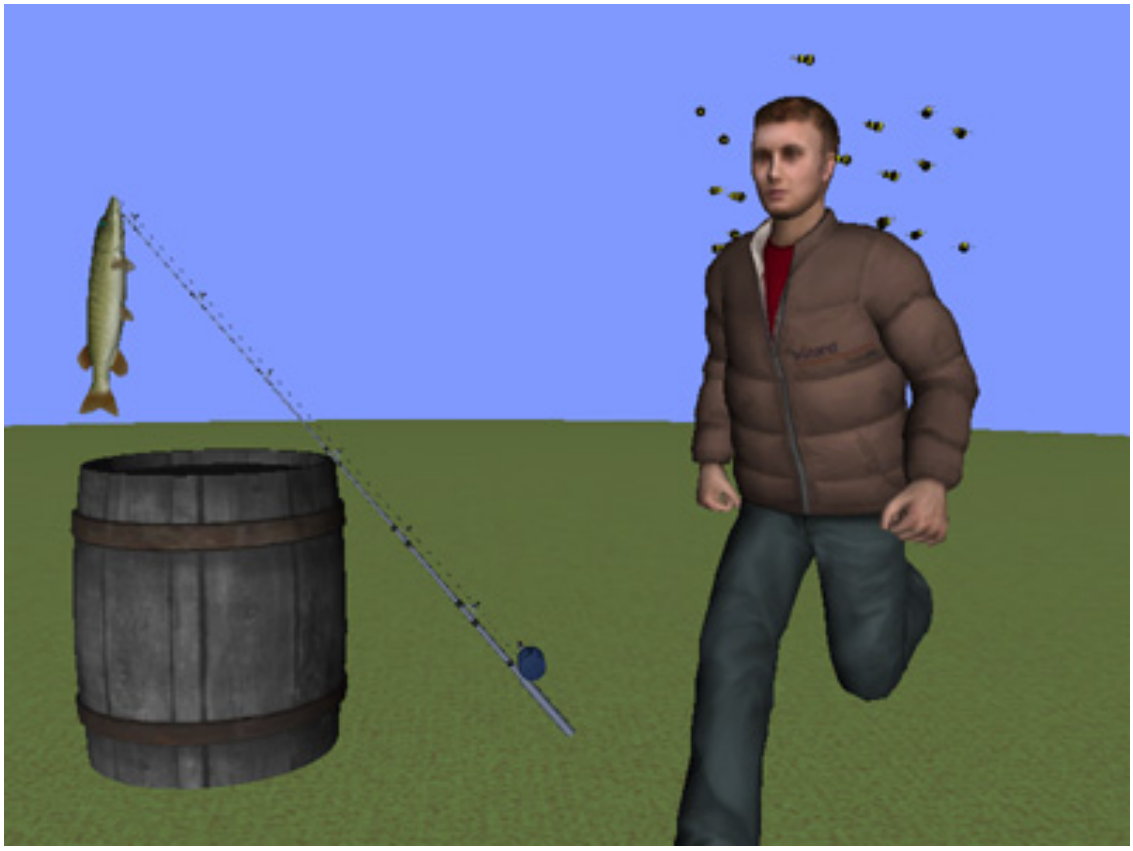
This scene's a little dull, so let's add some action by adding a timer that makes the bees spin around the parent coordinate system. The timer will go off every 100th of a second, rotating the bees at a relative yaw of 5 degrees. In the code snippet we're using here, we start the timer with the vizact library's "ontimer" command, giving it the expiration time of .01 (one 100th of a second) and calling the swarm function that we've defined here.

```
#Now add a function that will make the bees spin.
def swarm():
     bees.setEuler([5,0,0], viz.REL_PARENT)
vizact.ontimer( .01, swarm )
```

Now let's get our avatar to respond to the bees by having him run in a circle. First we put the avatar in a state of running (the avatar's 11th animation is running). After that we use commands from the Python math and time libraries to put together a function that uses trigonometry to place the avatar in some point along a circle and orients him so he's always facing forward. Finally, we'll call our function with the ontimer function again.

```
#And add a function that will
#make the avatar run.
import math
import time
avatar.state(11)
def run_around():
   newX = -math.cos(time.clock()) * 2.1
```

```
   newZ = math.sin(time.clock()) * 2.2
   avatar.setPosition([newX, 0, newZ], viz.ABS_PARENT )
   avatar.setEuler( [time.clock()/math.pi*180,0,0], viz.ABS_PARENT )
vizact.ontimer(.01, run_around )
```



LINKING

Links are related but distinct from parent-child relationships. Links have a **source** and a **destination** and the transformation matrix of the source is applied to the destination. So, if the source is spinning or moving upwards, then the destination will be spinning or moving upwards. A variety of things can act as sources including 3D nodes, tracking sensors, and viewpoints while destinations are usually 3D nodes or the viewpoint. Links can be extremely handy because they allow you to use the same transformation data to control any number of nodes or viewpoints. For example, you can link 3D tracking data from input devices to the viewpoint in order to navigate through a world naturalistically. You might also use links to create the appearance of grabbing objects so that when a hand reaches out for a model, that model can stick to and assume the subsequent transformations of the hand.

Once you've linked a destination to a source, you can manipulate the link so that you do not use all of the link's data or so that you perform transformations on the link's data. For example, you can apply an offset such that the position of the destination will always be a specific [x,y,z] away from the source. You can also apply **operators** on a link to transform the link's data before it is applied to the destination.

To get a feel for how links work, run the link demo. In this demo, the wheelbarrow and the soccer ball are linked. The wheelbarrow is the source and the soccer ball is the destination. First go to the menu and try rotating or changing the position of the wheelbarrow. The default for links is for orientation and position data to be applied to the destination but NOT scale data. Note that applying transformations to the destination (the ball) has no effect since its transformation matrix is defined by the source matrix. To change which data are used in the link, you can change the link's mask (under the "link" menu in this demo). If you use the "viz.LINK_POS" mask, only position data will be applied to the link. As such, the destination's position can be manipulated independently.

Also under the "link" menu in this demo are the options to put an offset or an operator on the link. Each of these functions require a three member array (e.g. [1, 0, 0])to work. Generally, pre-operators change the destination in the source's coordinate system while post-operators change the destination object in its parent coordinate system. When you reset a link, you remove all the operators from that link.

Play with the demo and see if you can get the ball to appear that it is in the wheelbarrow such that when you rotate or move the wheelbarrow, the ball stays.

EXAMPLE:LINKS

Now we'll start a new script to play with links. First we'll add some models to the world, setting their positions and orientations. We'll also do some animations here and manipulate the avatars with some code that you probably won't recognize. We'll go over what this code does in later sections of this book.

```
import viz
viz.go()


#Place the viewpoint where you want it.
viz.MainView.setPosition(0,1.8,-5)

#Add models.
tent = viz.add('art/tent.ive')
```

```
barrel = viz.add('art/barrel.ive' )
ball = viz.add('soccerball.ive')

#Shift the barrel to a different position.
barrel.setEuler( [0, 0, 90] )
barrel.setPosition( [ -1,.5,0 ] )

#Use actions to make some of the models
#spin
ball.addAction( vizact.spin(0,1,0,360 ) )
barrel.addAction( vizact.spin( 0,1,0,-90 ) )

#Add a female avatar.
female = viz.add('vcc_female.cfg')
female.setEuler( 180,0,0 )
female.state(2)

#Add a male avatar and pose him.
male = viz.add('vcc_male.cfg')
import debaser
debaser.pose_avatar( male, 'links_pose.txt' )
male.setPosition([1,0,0])
male.setEuler( -45,0,0 )
male.state(5)
```

Next we'll link one of the male avatar's bones to the ball.

```
#Get the bone of one of the avatar's fingers,
#and link the ball to it.
finger_bone = male.getBone( 'Bip01 R Finger1Nub' )
finger_link = viz.link( finger_bone, ball )
```

If you run the script now you'll see that the avatar's hand goes right through the ball. To fix that we'll add an offset to the link so that the ball (the destination of the link) is offset in the y dimension. We'll also apply a mask to this link so that it only uses position data. This way, our ball can spin (as it was spinning before we linked it).

```
#Now add an offset so that the ball appears to
#rest on the avatar's finger.
finger_link.setOffset( [0,.1,0 ] )
#Set the link's mask so that it only uses
#position data.
finger_link.setMask( viz.LINK_POS )
```

Now we'll link the female avatar to the barrel so that she appears to be walking on it. If you try running the first line here without the other lines, it'll look odd. So once again we'll apply a mask and an offset to the link to get the desired effect.

```
#Link the female to the barrel.
barrel_link = viz.link( barrel, female )
#Set the mask of that link.
barrel_link.setMask( viz.LINK_POS )
#Offset the link.
barrel_link.setOffset( [-.5,.4,0] )
```

Now run the script again and see how things look. Next we'll use some sensor data to move the barrel's position in th world. We'll simply use position data from our mouse as the source. Here we use the "swapPos" command to swap the position of the link data so that the x and y of the mouse data are applied to the x and z of the barrel. We fix the y-axis data of the link at .5 so that the mouse data do not influence the y position of the barrel.

```
#Link the barrel to the mouse.
mouse_link = viz.link(viz.Mouse, barrel )
#Use the position data from the x and y of
#the mouse for the x and z of the barrel.
mouse_link.swapPos( [1,3,2] )
#Use .5 instead of the mouse data for the barrel.
mouse_link.setPos( [None,.5,None] )
```

Finally, we'll use a link to get an avatar's-eye perspective of the world. To do this, we'll grab the head bone of the avatar and link the main viewpoint to it.

```
#Link the view to the male's head.
head_bone = male.getBone( 'Bip01 Head' )
view_link = viz.link( head_bone, viz.MainView )
#Set the eyeheight at 0 (so the default
#eyeheight is not added to the data).
viz.eyeheight( 0 )
```

EXERCISES

1. Run the coordinate systems demo and hit the "scramble" button. This will apply a random scale, translation, and rotation to the parent and child nodes. Now see if you can put the return the child and parent to their original states using the transforms within either coordinate system..

2. Add six copies of the "art/barrel.ive" model to a world and arrange them in a stack like the one in the picture below.

3. Take the "Transformations and the scene graph" script and see if you can tweak the code so that the bees fly around the fish instead of the avatar.

4. Link the viewpoint to the avatar's head in the "Transformations and the scene graph".

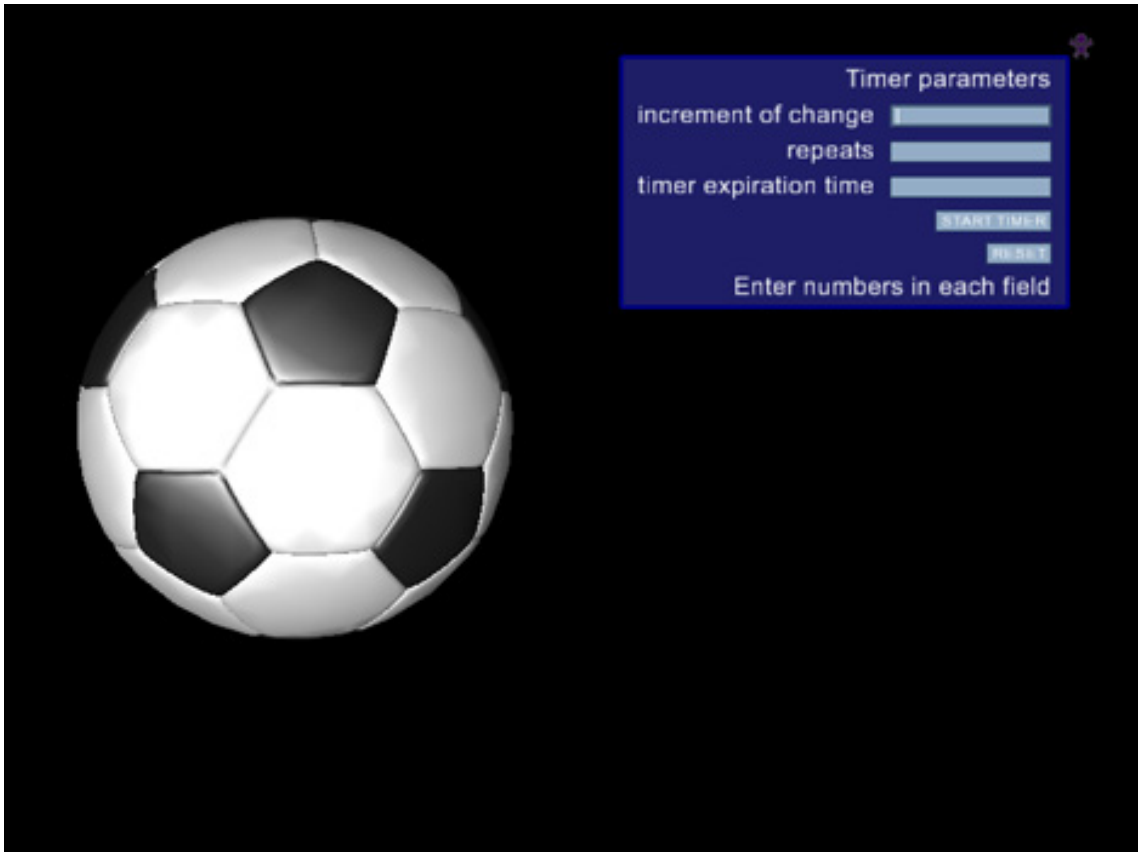
MODELING MOVEMENT ACTION

The 3D transformations we described in the previous section apply to instantaneous changes in an object's orientation, scale, position, or appearance. Of course in the real world, changes are seldom instantaneous. Cars move from one place to another, they don't just appear there. Plants grow larger, they don't just pop from one size to the next. In order to model gradual transitions, we need to apply changes incrementally over time. This section will go over some methods for coding changes. We'll

also go over ways to cue these kinds of animations so that the animations can be dynamic and interactive.

USING TIMERS

The trick to animating changes is using timers. If you went over the example in the scene graph section , you got a little exposure to timers already. Timers go off at scheduled intervals, calling a function every time they expire. If that function involves a transformation, then you can animate movement by repeatedly calling the function and applying the transformation incrementally each time. For example, if I want a basketball to appear to move one meter, then I could have it move one tenth of a meter every tenth of a second. The trick is to make the incremental changes small enough and the intervals of time short enough that the movement appears smooth and natural to a viewer.



Run the timer demo. The goal in this demo is to get the ball spinning in a fluid, realistic fashion using a timer. The menu will let you set the increment of relative change in rotation, the number of times you want the timer to go off, and the expiration time

of the timer (the interval before it goes off each time). Enter a few numbers and then hit 'START TIMER' and see how you did. Try spinning the object at different speeds, but getting it to move fluidly every time.

EXAMPLE SCRIPT: USING TIMERS

In this example, we'll animate a spider moving in a spiral, dropping a line of web as it goes. When you're going over this example, pay attention to the ways in which timers are used to animate a smooth path and to change the linear quality of the web.

First, let's get the program going by setting the position of the viewpoint, setting the background color, and adding and animating a spider avatar. We won't go over these commands in detail because they are covered in other sections. Note, though, that while the state command here animates the spider's leg movements, it doesn't actually move the spider-- that's what we'll use our timer for.

```
import viz
viz.go()

#Set the position of the view.
viz.MainView.setPosition( 0, 2, 0)
viz.MainView.setEuler( 0,90,0 )

#Set the background color.
viz.clearcolor( [.3,.3,.3] )

#Add the spider avatar.
spider = viz.add('art/spider/spider1.cfg')
spider_bone = spider.getBone( 'bone_root' )
spider.texture(viz.addTexture( 'art/spider/euro_cross.tif' ))
spider.setScale(.04,.04,.04)
#Animate the spider's legs.
spider.state(2)
```

Now we'll set up a timer for animating the spider's path. We do that by defining a function and then using the ontimer command from the vizact library to call that function repeatedly at an expiration time that we define (here it's one 100th of a second). Within our "move_spider" function, we'll use the variable "increment", adding to it every time our timer function goes off such that it increases over time. We'll use that increasing value to define the current place and orientation in the spiral using some trigonometry (don't worry about the trig here-- the purpose of this example is to get you using a timer).

```
#Animate the spider travelling
#in a spiral path.
increment = 0
C = 0.01
import math
def move_spider():
   global increment
   #Increase the increment every
   #time the function is executed.
   increment += .02
   #Use some trig to place the
   #spider on the spiral.
```

```
   x = C*increment*math.cos( increment )
   z = C*increment*math.sin( increment )
   spider.setPosition( [x,0,z] )
   #Find the increment the spider
   #should be facing.
   face_angle = vizmat.AngleToPoint([0,0], [x,z])-90
   spider.setEuler( [face_angle, 0, 0] )
#Call the move_spider function every
#hundredth of a second.
vizact.ontimer(.01,move_spider )
```

Run the script now and see how it looks.

In the next bit of code we'll add a web so we can better see the spider's path. The web that we're adding here is an "on-the-fly" object. On-the-fly objects can be generated dynamically, so they're handy for situations like this where we're making the object as we go. This on-the-fly object consists of vertices connected by lines. In our "lay_web" function we set the vertex, that the spider has a hold on, down in a fixed place and then link the next vertex to the spider's body. We call that function repeatedly again with our "ontimer" command.

```
#Start an on-the-fly object for the web.
viz.startlayer( viz.LINE_STRIP )
viz.vertex(0,0,0)
myweb = viz.endlayer()

#Make the object dynamic, since we'll
#be adding to it.
myweb.dynamic()

#Add the next vertex and link it to
#the spider.
current_vertex = myweb.addVertex([0,0,0])
web_link = viz.link( spider, myweb.Vertex( 0 ) )

def lay_web():
   #This function will lay the most
   #recent vertex of the web down.
   #We make these variables global
   #because we'll be changing them.
   global web_link, current_vertex

   #Remove the current link between
   #web and spider.
   web_link.remove()
```

```
   #Set the vertex at the spider's
   #current location.
   myweb.setVertex( current_vertex, spider.getPosition() )

   #Add a new vertex and link it
   #to the spider.
   current_vertex = myweb.addVertex( spider.getPosition() )
   web_link = viz.link( spider_bone, myweb.Vertex( current_vertex ) )

#Call the function on a timer.
vizact.ontimer(.5,lay_web)
```

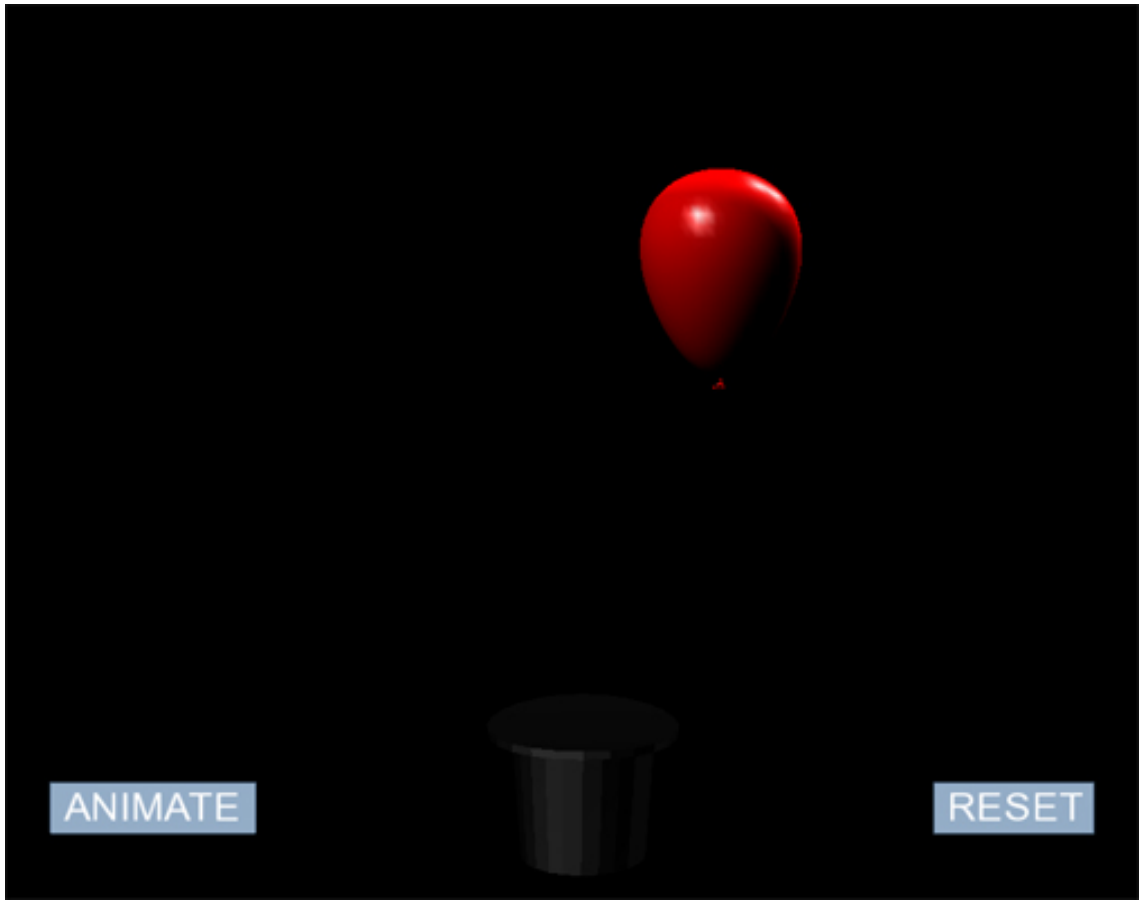
Run the script again and see how it looks.

Try changing the value that's added to "increment" in the function ("increment += .01"). A small increase in that value produces dramatic increases in the amount of ground the spider covers over time. Also try changing the expiration rates on the two timers we're using the first argument in the vizact.ontimer and see how those tweaks affect the spider's movement and the shape of the web.

CANNED ANIMATIONS

While timers are at the root of smooth animations, 3D simulation engines often have built-in methods for simulating fluid transitions that take care of the timer coding for you. In Vizard, these include functions in the action library (vizact.py) and the physics engine. Right now we'll go over the basic concepts of an action library. We'll cover the physics engine in the "Modeling physics" section.

The action library allows you to set up a queue of actions for each node. Once a node has a queue of actions, each action will get executed on the node in the order in which it appears in the queue. This queuing system is a handy way to perform a series of animations without needing to know exactly when one is over so that you can begin the next one. The queue can also be changed dynamically such that you can clear out all the actions in a queue or pause the procession through the queue.



Run the animation demo to get an idea of how actions work. Once the demo's running, go to the menus on top. Under the "actions" menu choose which actions you'd like to perform. Then go to the "sequence style" menu and choose a mode for the actions to run in. If you perform the actions in sequence, then the object will perform each action, wait for that action to finish, and then perform the next action. If you perform the actions in parallel, then all the actions will be combined into one action that will affect the object simultaneously.

The third menu in this demo allows you to add a waiting action into the object's queue. Waiting actions sit in a node's queue and wait for a particular event to occur. After the event occurs, the queue continues to be executed. The action demo uses two types of these waiting actions-- one that waits for a press of the spacebar and another that simply waits until 2 seconds have passed.

One more kind of action that's handy for animating multiple objects is signalling. The action library allows you to add signal-sending and signal-waiting actions to a node's

queue. When a node comes to a signal-sending action in its queue, it sends out a global signal. Any node that's waiting for that signal to proceed will then continue to proceed through its queue. For an example of this process, go under the "signal others" menu and select "signal other objects". Now when you hit animate, your first object will go through its cue of animations. When it's done, it will send out a signal which another node is waiting for.

EXAMPLE SCRIPT: USING ACTIONS

This example uses a handful of different actions and shows how the action library can be used to synchronize animations. It also shows how to create a custom action that can be applied to multiple objects.

First, set the scene by adding a few models to the world and setting the lighting. When we add the balloons in this code, we use the Python random module to choose a random color for each individual balloon. We also set various lighting attributes of each balloon with commands that are covered in more detail in the lighting section. Next

we add an environmental map, a texture that wraps around the whole scene. After that, we add some audio files to the scene with the addAudio command. We then add some lights with commands that are covered in the lighting section. Finally, we place the viewpoint in the world at a position and orientation that will allow us to see the scene when it loads.

```
import viz
viz.go()

#Add the ground and an avatar.
ground = viz.add('art/sphere_ground.ive')
avatar = viz.addAvatar( 'vcc_female.cfg' )
#Make the avatar idle.
avatar.state(1)

#Add some balloons.
balloons = []
#Import the python random module to make
#some features of the balloon random.
import random
for i in range(-5,6):
   for j in range(1,6):
      #Add and adjust the appearance and
      #position of several balloons.
       balloon viz.add('art/balloon.ive')
       balloon.setPosition( i*.8,.1,j*.8 )
       R= random.random()
       G= random.random()
       B= random.random()
       balloon.color( R, G, B )
       balloon.specular( viz.WHITE )
       balloon.shininess( 128 )
       balloons.append( balloon )

#Add a sky with an environment map.
env = viz.add(viz.ENVIRONMENT_MAP,'sky.jpg')
dome = viz.add('skydome.dlc')
dome.texture(env)

#Add some audio files.
inflate_sound = viz.addAudio( 'art/blowballoon.wav')
deflate_sound = viz.addAudio( 'art/deflateballoon.wav')

#Add lighting and remove the head light.
for p in [1,-1]:
```

```
   light = viz.addLight()
   light.position( p,1,p,0 )
viz.MainView.getHeadLight().disable()

#Position the viewpoint.
viz.MainView.setPosition([0,1.2,-8.9])
viz.MainView.setEuler([0,-12,0])
```

Now we'll get into the interesting stuff by adding some actions. We'll use these actions to animate the inflation of a balloon. First add some constants that we'll use in the actions,

```
#Set constants for actions.
INFLATED = [2,2,2]
DEFLATED = [.2,.2,.2]
BREATH_LENGTH = 3
DEFLATE_LENGTH = .1
```

Now we'll build some actions using the vizact library. This first set of actions will animate the inflation of the balloon, making the balloon larger and more transparent and playing a sound. We pull the actions together into one animation using "vizact.parallel". The rest of the code in this snippet pulls together animations for making the balloons float away, deflate and fall.

```
#Create actions to animation the inflation of a balloon (any balloon).
grow = vizact.sizeTo( INFLATED, BREATH_LENGTH, viz.TIME )
play_blowing_sound = vizact.call( inflate_sound.play )
inc_transparent = vizact.fadeTo( .7, begin=1, time = BREATH_LENGTH )
#Pull together parallel actions into single actions.
inflate = vizact.parallel( grow, play_blowing_sound, inc_transparent )

#Create actions for floating away.
float_away = vizact.move( vizact.randfloat(-.2,.2), 1,
vizact.randfloat(-.2,.2),8 )
float_away_forever = vizact.move( vizact.randfloat(-.2,.2), 1,
vizact.randfloat(-.2,.2) )


#Create actions to animate a balloon deflating.
shrink = vizact.sizeTo( DEFLATED, time = DEFLATE_LENGTH)
play_def = vizact.call( deflate_sound.play )
dec_transparent = vizact.fadeTo( 1,begin=.7, time = DEFLATE_LENGTH )
#Pull together parallel actions into a single action for deflation.
deflate = vizact.parallel( shrink, play_def, dec_transparent )

#Create a falling action.
```

```
fall = vizact.fall( 0 )

#Create an action that will wait a random amount of time.
random_wait = vizact.waittime( vizact.randfloat(.5,7) )
```

This next bit of code ties together the whole sequence of animations from inflation to floating away to deflating and falling. We use "vizact.sequence" here because we want the actions to affect the objects in a serial fashion.

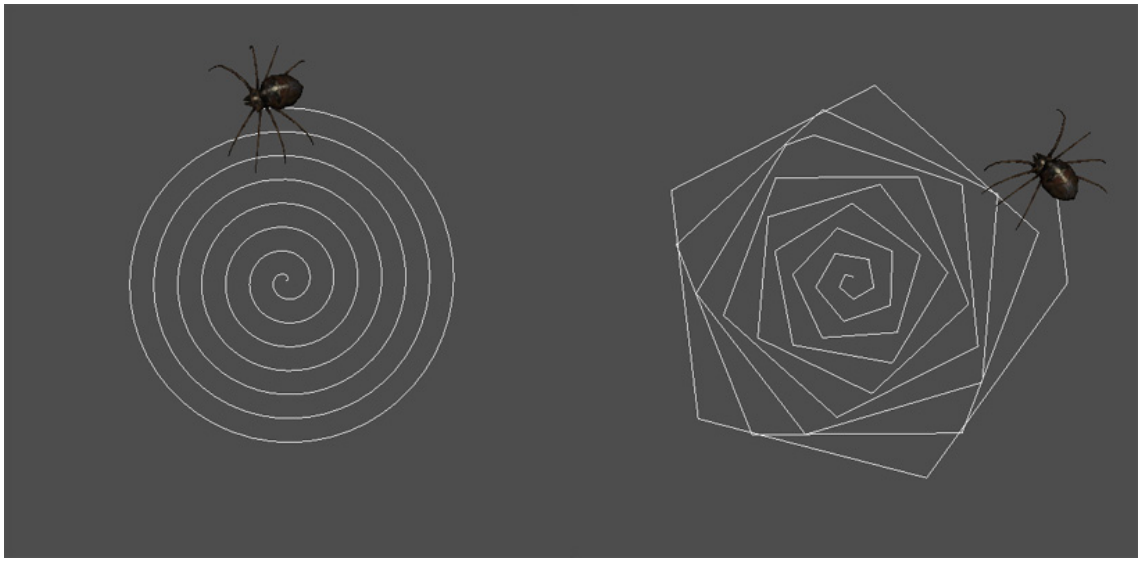
```
#Create the lifecycle of the balloon with a sequence.
life_cycle = vizact.sequence( [random_wait, inflate, float_away,
deflate, fall], 1 )
```

Finally, we want to add the whole sequence to each balloon with a keystroke. Here, the "vizact.onkeydown" sets it up so that when we hit the space bar (' '), the 'balloon.addAction' function will be called and will add the 'life_cycle' sequence to each balloon.

```
#Add the actions to our balloons.
for balloon in balloons:
   balloon.setScale( DEFLATED )
   vizact.onkeydown( ' ', balloon.addAction, life_cycle )
```

EXERCISES

1. Tweak the "Using timers" example so that the spider produces a very smooth web quickly, and then a very linear web slowly.



2. Add four different colored balloons to a world and then animate them each inflating individually with a different keystroke (add a red balloon that inflates when you press 'r', a yellow balloon that inflates when you press 'y', and etc.)

3. Tweak the "Using actions" example to make the avatar float away with the balloons.

LIGHTING

In order for 3D models to be visible, you need light. The easiest way to get a grasp on how virtual light works is to keep in mind how light works in the physical world. First off, light sources project rays of light. Each of these rays travels in a specific direction and varies in its color and intensity. When these rays reach an object, they bounce off the surfaces of that object into the world. Properties of the object's surface, such as color and shininess, work together to determine the nature of the reflected rays. These reflected rays continue to bounce around off of other objects. Eventually a few rays end up striking the surface of your retina and, behold, you see the object.

Digital simulations try to take these properties of light into account. In this section, we'll go over how to add light sources to a scene to produce a range of effects. We'll also go over various properties of object surfaces that effect the simulation of light reflected off those surfaces. See "LIGHTING" on page 59

One important distinction in light simulation is between simulating **local** and **global** illumination. Local illumination deals only with an individual object and the light

sources that effect it. Global illumination, on the other hand, attempts to account for the effects of all the objects in a scene on each other. So, global illumination simulates light bouncing off of one object onto another or the casting of shadows. While including global lighting effects makes for a more realistic simulation, calculating the many interrelationships between objects in real time is complex and costly to processing. There are, however, simple tricks to simulating global illumination that we will describe below.

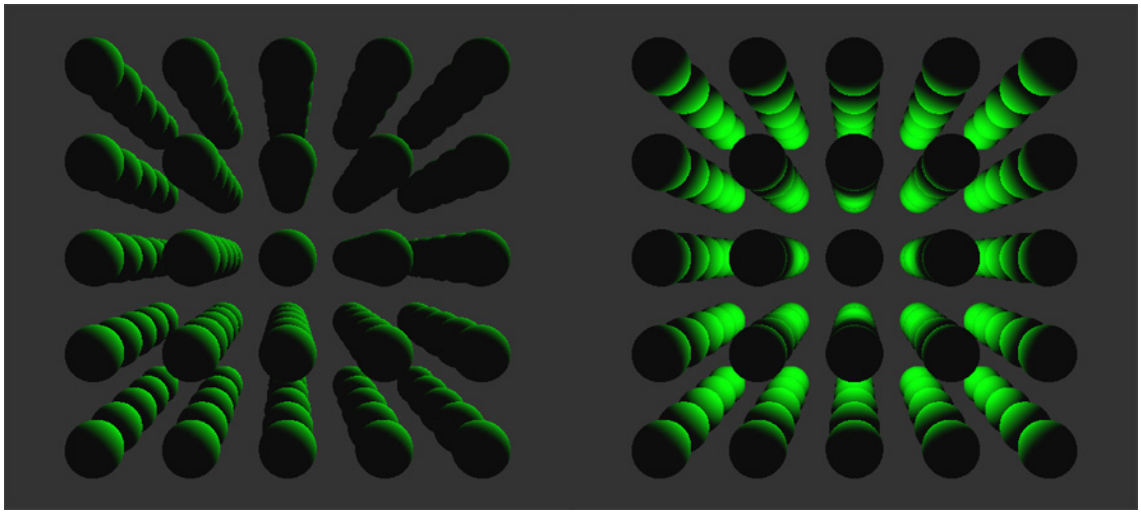
Note: when you create a world in Vizard, the program automatically adds a "headlight" to light scenes in the simulation. It's located at the viewpoint and lights the scene accordingly. In the sections below, we'll remove that headlight and explore other options for adding lights into a scene.

LIGHT SOURCES

OpenGL allows you to add up to eight light sources to a scene. Lights can be any color or intensity. One main distinction among light sources is between **directional** and **positional** lights. Directional light comes from a given direction but has no position in space. The rays from directional light sources all come in parallel from the same direction. This light source will strike all objects in the scene from this same direction. In this sense, directional light simulates light that's coming from a long distance away, like light from the sun or the moon.

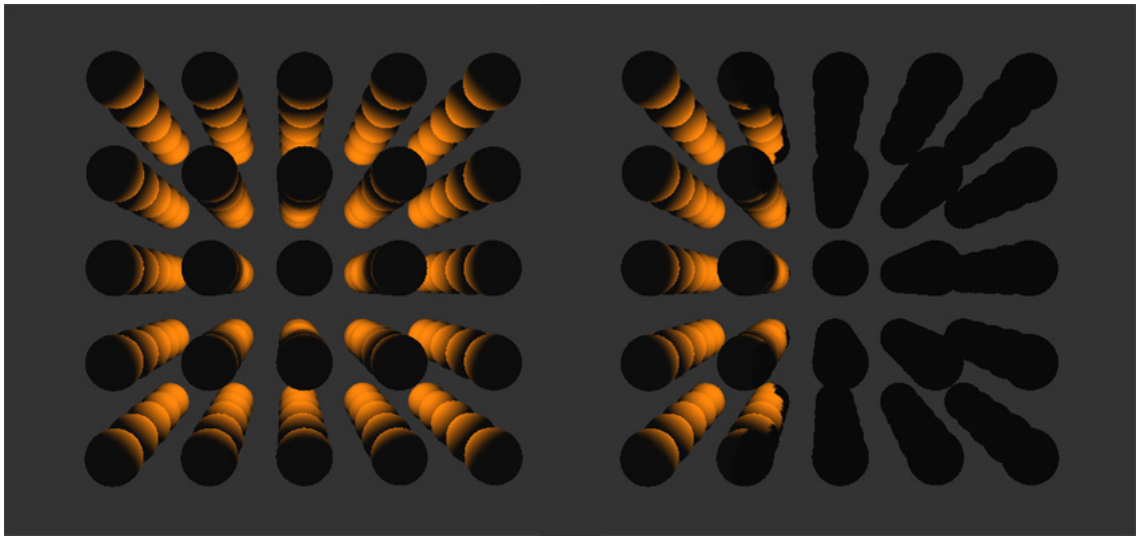
Positional lights, on the other hand, come from one point in space. The rays of light from this kind of source emanate from that one spot such that the bottom of objects above the source will be lit, the top of objects below the source will be lit, the left side of objects to the right of sources will be lit, and etc.

To get a better idea of the difference between directional and positional lights, run the light source demo. When it starts, go to "positional or directional" on the pop-down menu and try flipping back and forth between the two options. Notice how each sphere is illuminated in the same way with directional lighting but that with positional lighting, a sphere's illumination depends upon it's location with regards to the light source. (In this demo the light source is located in the middle of the cube of spheres.)



When you're dealing with positional lights, you have a number of different options available to you. You can make the source emanate light in all directions (a **point light**) or you can narrow it's focus and point it in a specific direction (a **spotlight**). With positional lights, you can also change the way in which light attenuates over distance. This attenuation can be based on a variety of formulas. If you go to the "attenuation" drop-down in the light source demo, you can change the attenuation from none to quadratic attenuation (one of the standard attenuation options).

Because spotlights have a narrow range of illumination, they have a number of adjustable parameters including the spread of the spotlight, the direction in which it points, and the degree to which the intensity of light decreases as you move away from the center of the spot. The direction and spread in the light demo are fixed, but you can vary the spot exponent by selecting different values under the "spot exponent" drop-down.



LIGHTING AND SURFACES

Intuitively we tend to think of lighting as coming from a source-- a light bulb, the sun, etc., but when modelling a virtual world it's also important to consider the relationship between light and surfaces that it bounces off. Four important categories of surfaces effects are diffuse light, specular light, ambient light, and emissive light.

Diffuse light comes from a specific direction. It illuminates the surfaces that face that direction and does not illuminate the surfaces that don't (and illuminates with decreasing intensity in between). Diffuse light reflects off a surface in all directions equally so that the lighting doesn't change as the viewing angle changes. This kind of light provides shading that makes models look 3-dimensional. Diffuse light needs a source (as discussed in the previous section) to define the direction from which the light is coming. The previous section used diffuse lighting in its demo (light source demo).

Specular light is similar to diffuse light in that its illumination depends upon the direction of the incoming rays on the surface. Specular light, however, does not reflect equally in all directions. Instead, the intensity of specular light varies as the angle of view changes. In practical terms, specular light is important because it gives glossy surfaces their shininess.

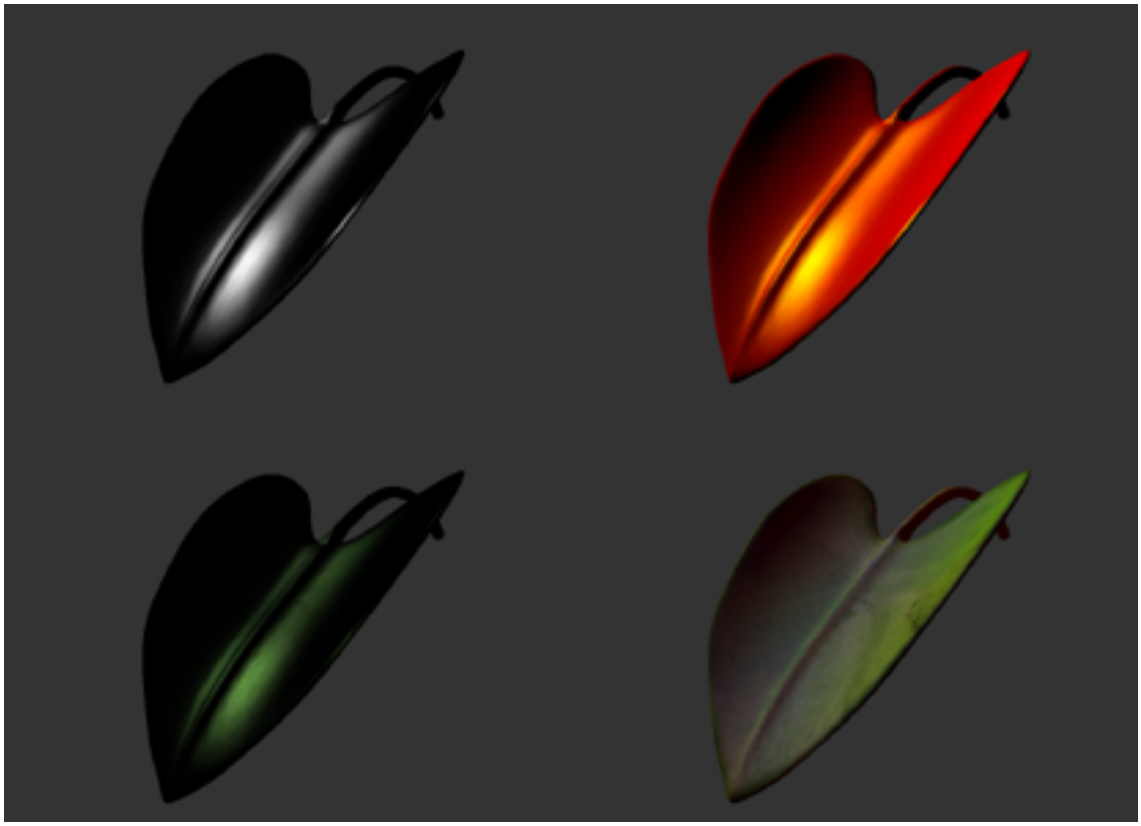
To check out the effects of specular light, run the light surfaces demo. Go to the drop-down menus at the top of the screen and go to the "specular" drop-down and choose a color for specular highlights. Then, go to the "shininess" drop-down and pick "128". This setting affects the sharpness of the specular highlights. Now rotate the viewpoint with the mouse. You should see your specular highlights move over the surface of the

leaf as you rotate the view. If you have trouble seeing the highlights, try removing the leaf's texture with the 'texture' drop-down.

Ambient light is a feature of an object's surface that emulates light coming from all directions and is scattered in all directions. Because ambient light doesn't come from any one direction, it needs no source.

To play a little bit with ambient light, run the light surfaces demo again and try changing the color of the object's ambient light with the "ambient" drop-down menu.

The fourth main kind of light is **emissive light**. An emissive light simulates light emanating from an object. Emissive illumination, light comes from all surfaces uniformly. Emissive light is a good way of simulating objects that are, themselves, simulating light sources (e.g. a light bulb). Keep in mind, though, that we're still only talking about local lighting here. So, although changing the emissive property of an object might make the object appear to glow, the light it appears to produce will not illuminate neighboring objects.



Now play a with emissive lighting by playing with the "emissive" drop-down menu. As you try different settings in this demo, note that these surface light parameters work together in the rendering of an object. So, keep in mind that the end result is a combination of all of these factors.

The kinds of lighting we've gone over in this section can be changed dynamically in a scene, but these are also aspects of an object's material (in addition to its texture). 3D modelling programs (Maya, 3DS Max, etc.) have a host of tools that allow you to manipulate these aspects of a model's material. When the object is rendered in a simulation, these components of a material will be combined with the effect of whatever light sources are in the scene.

EXAMPLE: LIGHTING A SCENE

In this example, we will use a variety of different kinds of lights to illuminate a scene.

First, let's open a Vizard script and add some models to the world and position the main viewpoint so that we can see them.

```
import viz
viz.go()

#Add a model of a forest.
forest = viz.add('art/forest.ive' )

#Add an avatar to stand there idly.
a = viz.addAvatar( 'vcc_male.cfg')
a.setEuler( [20,0,0] )
a.setPosition( [-.78,0,.3] )
a.state(1)
#Set the viewpoint's position and
#orientation so that we'll be able to see our scene.
viz.MainView.setPosition( [-.75,1.8,4.2 ] )
viz.MainView.setEuler( [-180,4, 0] )
```

Vizard has a default headlight that is linked to the main viewpoint. If you want to see what the world will look like with that head light, run your script now. After you've checked that out, disable this light so that we can see the effects of adding different kinds of lights. We do this by grabbing the main view's headlight with the viewpoint library's "getHeadLight" command. Once we have the light, we disable it with the node3d:light command "disable".

```
#Disable the default head light.
viz.MainView.getHeadLight().disable()
```

Now let's add a directional light to create the appearance of moonlight in the world. Note the arguments we use in the position command. That fourth number (0) makes the light directional. If we put a 1 there, the light will be positional. The other three numbers define what direction that light will be coming from in [x,y,z] coordinates. Since we set the y to 1 and the x and z to 0, the light will come from straight above.

```
#Add a directional light source.
moon_light = viz.addLight()
moon_light.position(0,1,0,0)
#Give the light a moonish color
#and intensity.
moon_light.color( [.6,.7,.9] )
moon_light.intensity( 1 )
```

Notice how this directional light illuminates all the upward facing surfaces. Play around with tweaking the intensity and color of the light with the lines we just added. When you're done, let's disable this light:

```
#Disable the moon light for a moment.
moon_light.disable()
```

We will attach the next light source to an object so that the object itself appears to be the light source. To help with this effect, we'll also add some emissive light to a portion of our model.

```
#Add a model of a lantern and place
#it so that it appears to hang on a the tree.
lantern = viz.add('art/lantern.ive')
lantern_position = [ 0.14 , 1.5 , 0.5 ]
lantern.setPosition( lantern_position )

#Add a light source to put inside the lantern.
lantern_light = viz.addLight()

#Define the light as a point,
#positional light. This is done
#with the last '1' in this command's
#arguments.
lantern_light.position( 0,0,0,1 )

#Link the light to the lantern.
viz.link( lantern, lantern_light )

#Grab the flame part of the lantern model
#and give an emissive quality to emulate light.
flame = lantern.getChild( 'flame' )
flame.emissive( viz.YELLOW )

#Play with the light source's parameters.
lantern_light.color( viz.YELLOW )
lantern_light.quadraticattenuation( 1 )
lantern_light.intensity( 8 )

#Give the lantern some shine.
lantern.specular(viz.YELLOW)
lantern.shininess(10)
```

Now run the script and see how it looks. To explore the effects we covered above, try tweaking the attenuation factor of the light source or the specular highlights of the lantern and run the script again. Once you're done, disable the lantern with the following line:

```
#Disable the lantern light.
lantern_light.disable()
```

Now we'll add a spotlight to the scene, linking it to a torch and animating the torch spinning.

```
#Add a model of a torch and place it in the scene.
torch = viz.add('art/flashlight.IVE')
torch.setPosition( [ -1.16 , 1.78 , 1.63 ])
#Add a light for the torch.
flash_light = viz.addLight()
#Make the light positional.
flash_light.position(0,0,0,1)
#Make this positional light a spot light by
#limiting its spread.
```

```
flash_light.spread(45)
flash_light.spotexponent( 40 )

#Link the light source to the torch.
viz.link( torch, flash_light )

torch.addAction( vizact.spin( 0,1,0,90, viz.FOREVER ) )
```

EXERCISES

1. Remove all the lights in the "Lighting a scene" example script and then add a spot-light that shines down on the avatar like a light from a helicopter.

2. Add a sphere to a world ("art/white_ball.wrl") and vary its diffuse, specular, ambient, and emissive parameters (using the <node3d>.color, <node3d>.specular, <node3d>.shininess, <node3d>.ambient, and <node3d>.emissive). Try to make it look like a bowling ball, a sun, and a cherry).

MODELING PHYSICS

The earlier section on modeling action went over some methods to get objects spinning or moving around your virtual worlds, but if you actually want to simulate the kinds of physical events that occur in the real world, you're going to need a little more in the way of everyday physics. So, how do you get a bunch of vertices and planes to act like an object in the physical world? How do you get a virtual ball to sail through the air in a gentle parabola, bounce off the ground for a few meters, and then roll for a while before coming to a rest?

Fortunately for us, physics engines have been developed to handle See "MODELING PHYSICS" on page 69these kinds of behavior. **Physics engines** come in a wide variety, from the computationally sophisticated and processing intensive scientific sort used to predict the effects of physical relationships in the real world, to more simple, realistic-looking but not necessarily precise simulations that work well in real-time. In this section we'll go over one of these real-time physics engines and show how you can use it to detect collisions and simulate forces in your virtual worlds.

COLLISION SHAPES

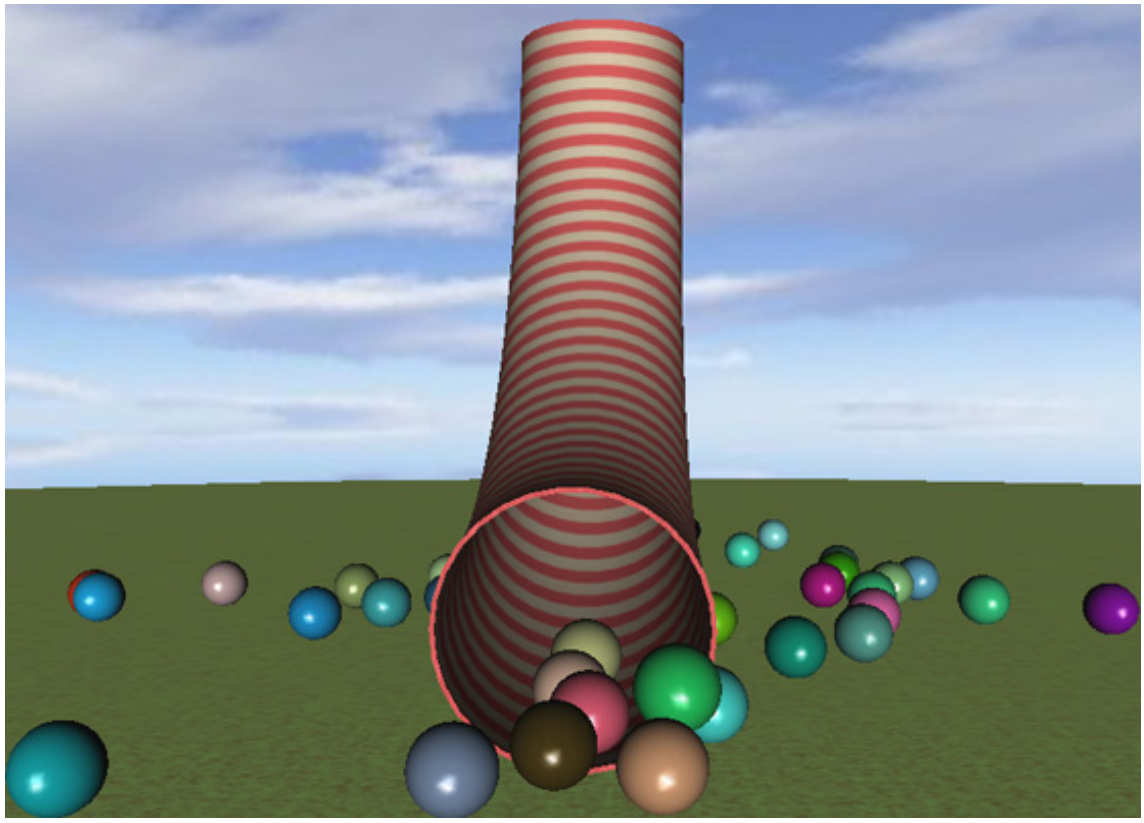
Collision detection is an important capability of a physics engine. To make collision detection possible in real time, physics engines use **collision shapes**. A collision shape surrounds an object and defines the physical boundaries of that object. In an ideal world the collision shape is the exact same shape as the model itself. Unfortunately, the more complex the collision shape, the more intensive the processing of collision detection is for your rendering computer since the computer needs to calculate in real time all the possible intersections between all collision shapes in the world. Given that, collision shapes are often not exactly the same shape as the objects they surround.

An object's collision shape is used to determine when that object is colliding with another object or objects. When two collision shapes overlap, a collision is detected. The geometry of the collision shape will further affect the way the shape behaves when it collides with something-- a sphere is likely to roll along a surface whereas a box might not. As such, it's important to pick the best shape possible when you're choosing a collision shape.

Run the physics demo now to get an idea of how collision shapes work. When you first load the demo, a grid of cubes is falling from a point in the sky. Those cubes (the dropping objects) have collision boxes around them and they're falling onto the barrel model (the target object) which also has a collision box around it. As you can see, the cubes that fall from the sky fall around the collision box and not the barrel itself. Go under "target object" and try changing the collision shape of the barrel (the target) to

see which collision shape works best. Also try changing the target model and the dropping models.

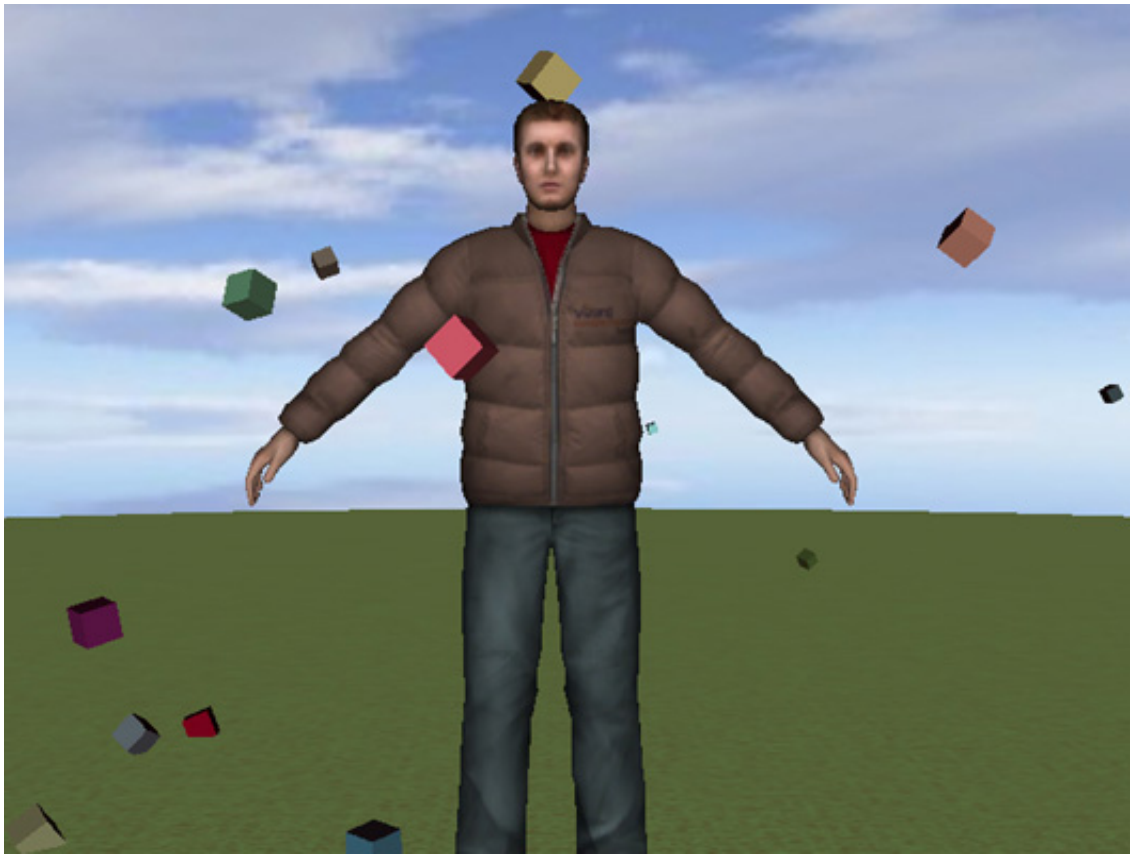
Note: As you could probably tell from running this demo, the collide mesh collision shape is the best fitting shape. In fact, it's the only mode that will allow the dropping objects to pass through the tube model. That's because it's based on the mesh of the model itself. The disadvantage to using the collide mesh, however, is that dynamic physics (forces) cannot be applied to a collide mesh. So, that method works the best when you want other objects to bounce off the model, but you don't mind if that model remains unaffected.



FORCES

In addition to collision detection, the other benefit of physics engines is the ability to apply **forces** to objects. This ability is sometimes referred to as the **dynamic effects** of a physics simulation. The most obvious force at work in a physics engine is gravity. When gravity is turned on, objects whose dynamic capabilities are enabled are pulled straight downward by a force of -9.8 m/s2 (although you can tweak this value or remove it entirely). As you saw in the physics demo, forces aren't applied in isolation. The combination of collision detection and force calculation between objects with physical parameters allows the physics engine to simulate the interrelationships of physical objects.

In addition to their geometry, collision shapes have a number of different qualities that determine how forces will affect them. They have hardness, friction, bounce, and density, all of which will work together to determine how much and how fast an object moves and how that object will affect the objects with which it collides. Run the physics demo again and try changing some of the parameters under "target model options" and under "dropping objects options".



In addition to gravity, a physics engine will allow you to apply a broad range of forces to objects. The forces that you can apply tend to vary in a few dimensions-- in the duration for which the force is applied, the direction vector of the force, the coordinate system in which it is applied (local versus world), where on the object the force is applied (at the center of gravity or elsewhere). These general parameters are combined to produce the effects of springs, motors, and thrusters as well as more basic commands for changing an object's velocity or applying torque. Physics engines also allow you to create joints so that objects can rotate about points in space when a force is applied to them as a door would swing when it's pushed.

EXAMPLE SCRIPT: PHYSICS

In this example we'll use collision detection events to trigger animations and forces to animate a projectile. We'll do this to create a circus world where the user is trying to hit balloons on a spinning wheel while avoiding the avatar attached to that wheel.

In the first chunk of code, we get a world running and then set up the lighting. We also turn mouse navigation off so that we can use the mouse for other purposes and set the viewpoint in a good place. Finally, we add some models.

```
import viz
viz.go()

#Add spotlights and remove the head light.
viz.MainView.getHeadLight().disable()
spot_light = viz.addLight()
spot_light.position( 0, 0,0, 1)
spot_light.setEuler( 180,65,180)
spot_light.setPosition(0,4.55,2.5)
spot_light.spread(45)
spot_light.spotexponent(5)
spot_light.intensity(1.5)

#Set up the viewpoint.
viz.MainView.setEuler(180,0,0)
viz.MainView.setPosition(0,1.5,6)

#Disable mouse navigation
viz.mouse(viz.OFF)

#Add the models.
tent = viz.add('art/tent.ive')
stand = viz.add('art/stand.ive')
wheel = viz.add('art/wheel.ive')
avatar = viz.add( 'vcc_male.cfg' )
knife = viz.add('art/knife.ive')

#Put the wheel in position.
wheel.setPosition([.02,1.62,.24])
wheel.setEuler([0,-20,0])
```

Next we create a link object with the wheel as the source and the avatar as the destination. We apply a pre translation on that link to get the avatar in the correct position on the wheel. Because the avatar is linked to the wheel, he will now share all the transforms with the wheel so that if the wheel spins later, the avatar will spin as well.

```
#Link the avatar to the wheel.
link = viz.link( wheel, avatar  )
#Use a pre-translation on the link to
#get the avatar in the correct
#position on the wheel.
```

```
link.preTrans( [0,-1,.1 ])
#Get the wheel spinning.
wheel.addAction( vizact.spin(0,0,1,90) )
```

Next we'll grab some of the avatar bones and put them in the positions we want them (these commands are covered in more detail in the Avatars section.

```
#Grab a few of the avatar's bones and
#put them into position.
bones = [' L Thigh', ' R Thigh',
' L UpperArm', ' R UpperArm' ]
rolls = [-20,20,-40,40 ]
for i in range(len(bones)):
   bone = avatar.getBone( 'Bip01' + bones[i] )
   bone.lock()
   bone.setEuler( [0,0,rolls[i]], viz.AVATAR_LOCAL)
#Set the idlepose of the avatar to -1 so
#that it doesn't go to
#animation #1 when it is idleing.
avatar.idlepose( -1 )
```

Now we'll add the balloons, adjust their appearance, and link them to the wheel.

```
#Add the balloons.
import random
balloons = []
balloon_coords = [[0,-.75],[.75,-.75],
[-.75,-.75],[.5,0],[-.5,0],[.4,.9],
[-.4,.9], [.9,.45],[-.9,.45]]
for coord in balloon_coords:
   balloon = viz.add('art/balloon2.ive')
   R = random.random()
   G = random.random()
   B = random.random()
   balloon.color( R, G, B )
   balloon.alpha(.8)
   balloon.specular( viz.WHITE )
   balloon.shininess(128)
   #Link the balloons to the wheel.
   link = viz.link( wheel, balloon )
   #Do pre transformations on the link
   #to put them in different places.
   link.preTrans( [coord[0],coord[1],0] )
   link.preEuler( [random.randrange(-
60,60),random.randrange(100,150),0] )
   balloons.append( balloon )
```

We're going to want some actions to use to animate our world, so we'll define those now. First, we add an audio file for the avatar and create a sequence to animate him if he gets hit by the knife. We'll also create the actions to animate a balloon if it gets hit.

```
#Add audio for the avatar.
cry = viz.addAudio( 'art/grunt.wav' )
#Create the actions for a wounded avatar.
avatar_hit  = vizact.parallel(  vizact.fadeTo(0,speed = 1),
vizact.call( cry.play ))
avatar_sequence = vizact.sequence([ avatar_hit, vizact.fadeTo(1,speed =
1 )], 1 )

#Create the actions for popping balloons.
popping_sound = viz.addAudio('art/pop.wav')
play_pop = vizact.call( popping_sound.play )
popping_action = vizact.sizeTo([.1,.2,.1],time = .5)
popping = vizact.parallel( [play_pop, popping_action] )
```

Now we'll define the collision shapes for our objects. First, we use a collide mesh on the wheel, the balloon, and the avatars. We add a collision plane to the ground. Because we want the knife to be affected by forces (dynamic physics), we'll put a collide box on it. We're also going to enable collide notification for the knife. This means that when we register a callback for collision events, Vizard will monitor this object.

```
#Add a collide mesh for the wheel,
#balloons and avatar.
wheel.collideMesh()
for balloon in balloons:
   balloon.collideMesh()
avatar.collideMesh()
#Grab the ground and add a collision plane.
ground = tent.getChild( 'ground' )
ground.collidePlane(0,1,0,0)
#Add a collide box for the knife.
knife.collideBox()
#Add a collide notify flag so that
#collisions with the knife will trigger events.
knife.enable( viz.COLLIDE_NOTIFY  )
```

In the next bit of code we define a function that will deal with sticking the knife to whatever it hits. This function takes a 3D node as its argument. The function disables the knife's physics so that it won't be affected by forces anymore and then links the knife to that node.

```
#Name a variable for the link that
#will be created whenever the knife
#sticks to the wheel.
knife_link = 0
#Define a function that will stick
#the knife to something.
def stick_knife( into_what ):
   global knife_link
   #Disable the knife's physics so
   #that the link won't compete with forces.
   knife.disable( viz.PHYSICS )
   #If the knife is linked to anything else,
   #remove that link.
   if knife_link:
        knife_link.remove()
   #Create a new link to whatever the
   #knife has struck.
   knife_link = viz.grab(into_what, knife )
```

Now we'll define a function for throwing the knife. This function takes the position of the mouse on the screen and creates a vector from that point on the screen into the world. It then unlinks the knife in case it's linked to something. Then it uses "reset" to clear the knife's current velocities. "Throw_knife" also enables the physics capabilities of the knife in case they've been disabled. Then it puts the knife in position and sets its velocity along that vector. That should set the knife sailing. We'll trigger this function with a mouse event so that the function gets called every time the user lifts the mouse button.

```
#This function will throw the knife when the
#mouse is clicked.
def throw_knife():
   #Convert the current mouse position from
   #screen coordinates to world coordinates
   line = viz.screentoworld(viz.mousepos())
   #Create a vector that points along the
   #line pointing from the screen into the world.
   vector = viz.Vector(line.dir)
   #Set length of this vector.
   vector.setLength(20)
   #Remove the link if the knife is linked
   #to something.
   if knife_link:
         knife_link.remove()
   #Reset the knife. This will get rid of any
   #forces being applied to the knife.
```

```
   knife.reset()
   #Enable physics on the knife in case
   #they've been disabled.
   knife.enable(viz.PHYSICS)
   #Move the knife into position at the
   #beginning of our line.
   knife.setPosition( line.begin )
   #Set the orientation of the knife.
   knife.setEuler(180,90,0)
   #Set the velocity for the knife to travel
   #in (the vector we calculated above).
   knife.setVelocity( vector )
vizact.onmouseup(viz.MOUSEBUTTON_LEFT,throw_knife)
```

Our next function will be called whenever there is a collision. Because the knife is the only object for which we enabled collision notifications, this function will only be called when the knife hits something. The argument passed by the collision event ("e" in the function we're writing) contains data about the collision. The value e.obj2 is the model that the knife collided with. We'll use this data in our function to determine what happens to the knife. If it hits the avatar, it'll stick to him and the avatar will be animated with the actions we defined earlier. If the knife hits the wheel, it will stick to the wheel. If the knife hits a balloon, we'll add popping actions to the balloon, stick the knife to it, and disable its physics capabilities so that it can't get hit again.

```
#Define a function to handle collision events.
def onCollideBegin(e):
   #If the knife hits the avatar or the wheel,
   #stick it to them.
   if e.obj2 == avatar:
        avatar.addAction(avatar_sequence)
        stick_knife( avatar )
   if e.obj2 == wheel:
        stick_knife( wheel )
   #If the knife hits a balloon, pop it and
   #disable its physics
   #so it can't get hit again.
   if balloons.count( e.obj2 ):
        e.obj2.disable( viz.PHYSICS )
        e.obj2.addAction( popping )
        stick_knife( e.obj2 )
#Callback for collision events.
viz.callback(viz.COLLIDE_BEGIN_EVENT,onCollideBegin)
```

We're going to want to reset the balloons once they're all popped, so we'll write a quick function handle that. This function will simply scale the balloons back to size and re-enable their physics capabilities.

```
#Define a function to reset the balloons.
def reset_balloons():
   for balloon in balloons:
      balloon.addAction( vizact.sizeTo([1,1,1], time = .5 ))
      balloon.enable( viz.PHYSICS )
vizact.onkeydown('r', reset_balloons )
```

Finally, we'll enable physics in the world with the viz.phys.enable() command. Note that the physics engine does not kick into gear until you call this function.

```
#Enable the physics engine.
viz.phys.enable()
```

EXERCISES

1. Write a script that drops a ball ("art/white_ball.wrl") from a few meters off the ground on a keypress. Add a ground to the world and give it a collision plane. Give the ball a collision shape and define its bounce such that it will bounce up and down for a while on the plane when you drop it.

2. Add a collision event callback to the code that you wrote in the previous exercise. Use that event to change the color of the ball so that it changes every time the ball collides with something.

AVATARS

Avatars are a special case of 3D models. They're used to represent humans (and other living things) and, as such, they have a number of unique capabilities. Specifically, avatars have the ability to move and rotate their various parts (arms, legs, spine, face, etc.) in ways that, ideally, approximate the movement of living things.

Note: In technical terms, avatars are human representations that are being controlled (or inhabited) in real time by real humans. A human representation that's being controlled by a computer is called an **agent**. For simplicity's sake, however, we refer to digital models of humans as avatars throughout this text.

THE STRUCTURE OF AN AVATAR

Avatars tend to be constructed a little differently than other models which affords them their capabilities. Specifically, avatars have skeletons that can be used to animate them. **Skeletons** are hierarchical structures similar to those discussed in the scene graph chapter. The individual units of skeletons are **bones**. Each bone inherits the transformations of the bone above it in the hierarchy. The bones of an avatar are not actually visible in a simulation-- they just control the movement of the **meshes** that make up the avatar's body. The **meshes** of an avatar are **rigged** to the skeleton. Rigging can be quite complex in that a given mesh might be rigged to multiple bones or different parts of a given bone. When a bone moves, the parts of a mesh that are rigged to it move with the bone. As a consequence, the movement of a bone or bones will twist and stretch the meshes of the avatar's body.

Run the avatar demo script to play with moving an avatar's bones. Under the "Pick a bone" menu you'll find three more drop down menus for picking a bone from the left, right, or center of the body. Pick one of these bones and then try rotating it with the sliders on the menu on the upper right of the screen. Try positioning the avatar with its hands on its hips or with it's hand in a wave. Notice that when you rotate a bone such as "Bip01 Spine1" that the mesh of the avatar stretches and twists. This internal malleability is one of the benefits of having the avatar's structure.

(When you close the avatar demo, it will save a record of the avatar's pose in a text file named "avatar_bones.txt" in your folder. You can use that pose with the same avatar later with the pose_avatar command from the debaser module. Check out the "links example.py" script for a demonstration.)

AVATAR ANIMATION

Avatar animation is a complicated business. Basic human patterns in motion, such as walking, are difficult to animate realistically. Many bones are moving at a variety of

speeds and any number of flaws in an animation can create a frightening distortion of the human body which will, at best, detract from a user's sense of immersion and, at worst, frighten the user.

While manipulating the bones of an avatar can be handy for real-time, interactive animation, complex animations are best handled by animation software. Animation software can help you create a "canned" animation that you can then play in your simulation when you need it. These canned animations are convenient because they can be played on any avatar of the same format.

One of the big challenges in animating an avatar is keeping the avatar's movements within the boundaries of reality. Even the world's greatest contortionist couldn't get into the positions that you could get an avatar in using the avatar demo with just a few degrees rotation around a joint or two. Animation software nowadays helps with realistic animations by solving the **inverse kinematics** of a given position. Inverse kinematics solves the problem of given the position of the end of a structure (e.g. the hand), what angles should the joints in the rest of the structure be (e.g. the arm and the torso. This is a complex problem given the fact that there are many possible solutions but only a small range of realistic looking ones for the human body. Nevertheless, animation software will help solve these problems such that you can place the hand up and over from the body and the software will fill in the blanks to move and rotate the arm and body so that the hand can appear in that position.

The two main ways of producing animations for avatars are keyframing and motion capture. With keyframing, the animator poses the avatar in different positions along a timeline (at different keys). Through inverse kinematics and using realistic movement curves for the body, the animation software infers the position of the body between keys. What results is a smooth animation of the avatar.

While current animation software does a remarkable job handling keyframing, the most realistic avatar animations are produced through motion capture. Through motion capture, the movement of an actual person (or thing) is recorded digitally. This recording is then used to animate the 3D model (the avatar or whatever). Because motion capture uses an actual recording of human motion to animate an avatar, it produces the most realistic movements for avatars.

EXAMPLE SCRIPT: ANIMATING AVATARS

In this example we will animate a male and a female avatar with a variety of methods.

First let's set up the world with the models we'll need, placing the viewpoint in a good place.

```
import viz
viz.go()
```

```
#Add the models.
ground = viz.add('art/sphere_ground3.ive')
male = viz.add('vcc_male.cfg')
female = viz.add('vcc_female.cfg')
female.setPosition(-1,0,1 )

#Add a sky.
env = viz.add(viz.ENVIRONMENT_MAP,'sky.jpg')
dome = viz.add('skydome.dlc')
dome.texture(env)

#Set the viewpoint.
viz.MainView.setPosition(0,1.8,5)
viz.MainView.setEuler(180,0,0)
```

Now we'll start the avatars in one of their animations. This state command will loop that animation until we call another.

```
#Start the avatars in an idling animation.
#This will loop.
female.state(1)
male.state(1)
```

Next we'll write a little code that will take the male's head bone and then rotate it to follow the female avatar wherever she is. First we grab the bone with the "getBone" command, using the bone's name as the argument. Next, we lock the bone so that it won't be affected by animations. (If you don't lock the bone, then you will not be able to apply any transforms .) Within the function, we get the male's and female's positions and find the angle between. We'll set the orientation of the male's head with this angle (excluding angles that are outside of the natural realm). Finally, we'll call this function with a timer.

```
#Get a handle on the male avatar's head
#and lock it.
bone_head = male.getBone( 'Bip01 Head' )
bone_head.lock()
def follow_target():
   #This function will rotate the male's
   #head to face the female.
   #First get the position of the female
   #and male.
   f_pos = female.getPosition()
   m_pos = male.getPosition()
   #Pull out their x and z.
```

```
   f_point = [f_pos[0], f_pos[2]]
   m_point = [m_pos[0], m_pos[2]]
   #Get the angle between those.
   angle = vizmat.AngleToPoint( m_point, f_point )
   #Rotate the male's head that angle
   #(as long as it's in the natural
   #range.
   if angle < 90 and angle >-90:
        bone_head.setEuler( angle,0,0, viz.AVATAR_LOCAL )
vizact.ontimer(.01,follow_target )
```

In the next snippet of code we'll give the user the ability to cue some animations with keypresses. First, we'll call the blend function when the user presses the 'd' key. We'll blend in animation 5 over 5 seconds, blending the dancing animation with the current animation (which we set as animation #1 earlier). We'll also call a blend function when the user presses "s". This blend command will blend out the #1 animation.

```
#Blend in the male avatar dancing
vizact.onkeydown( 'd', male.blend, 5,1,5 )
#Blend out the male's idle animation.
vizact.onkeydown( 's', male.blend, 1,0,5 )
```

We will now animate the female avatar walking. For this we'll use the walkto command from the vizact library. This command accepts a point in space. It then handles rotating the avatar toward that position and animating her walking there. We'll put a couple of these commands in sequence and add them to the female avatar so that she's walking back and forth.

```
#Put the female avatar into position and
#define some walking actions for her.
female.setPosition(5,0,4)
walk_left = vizact.walkto(-2,0,1)
walk_right = vizact.walkto(3,0,1)
walking_sequence = vizact.sequence( [walk_left, walk_right], viz.FO-
REVER )
female.addAction( walking_sequence )
```

Now we'll define some physics shapes and define a function that will be called when there is a collision event. The physics commands are covered in more detail in the Modeling physics section. The important part to look at for understanding avatar animation is in the "onCollideBegin" function. When a collision happens with the male avatar, we execute animation #16. In this animation, the avatar falls onto his back. We include "freeze = True" in the arguments so that the avatar will freeze when it comes to the end of the animation. If we don't include this argument, the avatar will

pop back into his idle animation when the falling animation is complete. Also within this function, we will clear the female avatar's actions and have her clap (animation #4). When she is done clapping, she will return to her idling animation.

```
#Define some physics components.
male.collideMesh()
ground.collideMesh()
#Add a ball whose collisions we
#will watch for.
ball = viz.add('soccerball.IVE')
ball.collideSphere()
ball.enable( viz.COLLIDE_NOTIFY )
ball.setPosition(100,100,100)

#When the ball hits the male avatar,
#have him look around and then stand still.
#Also, clear the female avatar's
#actions and have her clap.
def onCollideBegin(e):
   if e.obj2 == male:
        male.execute(9, freeze = True )
        female.clearActions()
        female.execute(4)
viz.callback(viz.COLLIDE_BEGIN_EVENT,onCollideBegin)
```

We'll add a little more interactivity with a function that is cued by the spacebar. This function will send the ball flying toward the male avatar (the commands within this function are covered in more detail in the physics section).

```
#Shoot the ball with a keypress.
def shoot_ball():
   ball.reset()
   ball.setPosition( [1,2,5])
   ball.setVelocity([-2.8,.8,-15])
vizact.onkeydown(' ',shoot_ball )
```

Finally, we'll define a function that will reset the avatars to their original states so that the fun can begin again. Here the "stopAction" command unfreezes animation #16 and the "state" command puts the avatar in a loop of animation #1. We also add the walking animations to the female again. In the last line of this snippet, we enable the physics engine.

```
#Reset the avatars to
def reset():
   male.stopAction(16)
   male.state(1)
```

```
   female.addAction( walking_sequence )
vizact.onkeydown( 'r', reset )

#Enable the physics engine.
viz.phys.enable()
```

`

EXERCISES

1. Use the avatar demo to get the avatar in the position of waving. Write down the names of the bones you manipulated and the Euler angles that the bones were in. Now write a script that adds the avatar and puts the bones in those angles (use '<bone>.-setEuler' to rotate the bones with viz.AVATAR_LOCAL as the coordinate system).

2. Take the script from exercise #1 and add a timer that makes the hand wave back and forth.

3. Write a script where you add an avatar who's clapping (that animation is part of the "vcc_male.cfg"' and "vcc_female.cfg" models). Now add code that will make the avatar's animation blend into a cheer when you hit the 'c' key and blend out of cheering when you hit the 's' key.

HARDWARE

INPUT DEVICES

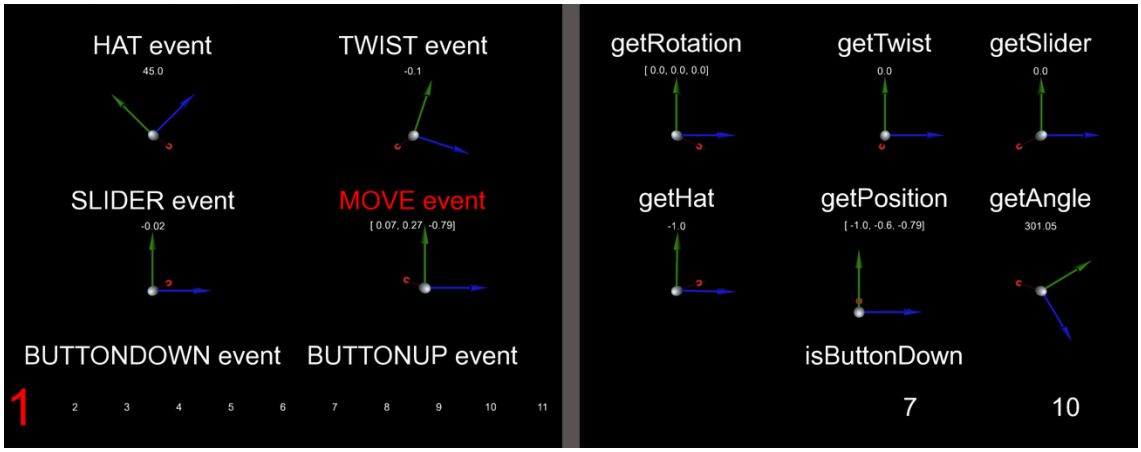
Hardware connects users to virtual worlds. Input devices allow us to navigate through virtual worlds and to interact with virtual objects, while output devices allow us to see, hear, and otherwise sense the content of those worlds. The main input devices we've used up until this point are the keyboard and the mouse. A variety of other input devices exist, mainly to track naturalistic human movement with orientation and position sensors. In this section, we'll go over how to use a simple input/output device like a joystick to interact with virtual objects or to navigate through a world.

The main output device we've used thus far in this workbook is the computer monitor. Nowadays, a wide variety of other kinds of displays are available to people creating See "HARDWARE" on page 89virtual environments. In the "Display devices" chapter of this section, we'll touch on a couple of these.

USING INPUT AND OUTPUT

In general, you need a plug-in for whatever device you're going to use so that you can access it (and its data) through your vr toolkit. Vizard supports a broad range of input and output devices (for a discussion of the hardware supported by Vizard, check out the Hardware chapter in the Vizard Help Reference chapter). Vizard has a module, vizjoy, to handle gamepad and joystick type USB devices. We'll use this module throughout this section. You'll need a joystick or a gamepad to go through the demo and tutorials in this section.

To get an idea of what your joystick can do, run the joystick demo. At the menu at the top of the screen, select "Joystick capabilities". This drop-down lists the vizjoy module's assessment of your joystick's capabilities, whether it has force feedback, which axes it has movement and rotation about, and how many buttons it has. We'll return to these capabilities in a moment.



There are two main strategies for incorporating data from input devices in your script -- **events** and **sampling**. when you register a callback for an event, you tell the program to wait until a specific event has occurred and then call a specific function. As such, the given function is only called at critical moments. (Events are described in greater detail in the "Events" section of the "Basic Scripting" chapter).

With sampling, the script draws the input data continuously. In your script, you use a timer to repeatedly sample data from the input device. Because of it's continuous quality, sampling is ideal for navigation using input devices.

Run the joystick demo again. Use the "Events vs sampling" menu to choose whether the animations are driven by events or sampling. The events in this demo are specific vizjoy events. Their texts will turn red when that specific event has occurred. The numbers at the bottom of the screen represent the buttons pressed. The sampling in this demo is done with the listed functions.

Many joysticks are also output devices in that you can apply force feedback to them. If your joystick has force feedback, go to the "Force feedback" menu and try applying a force.

EXAMPLE SCRIPT: USING SAMPLING AND EVENTS FOR HARDWARE INPUT

In this example, we'll use the vizjoy module to add a joystick and then to use that joystick to control the model of a cue in a virtual game of pool. We'll use both events and sampling to handle input from the joystick. In this example, we'll also use the pool module which can be found in the demos and examples folder. Start the script by importing the viz, math, and pool modules and by instantiating a pool_game object. We'll also enable physics and get the viewpoint in the right position.

```
import viz
import math
viz.go()
```

```
#Import the pool module and
#instantiate a game object.
import pool
game = pool.pool_game()

#Enable physics.
viz.phys.enable()

#Place the viewpoint in position.
viz.MainView.setPosition(0,4,-1.7)
viz.MainView.setEuler(0,68,0)
```

Next we'll use vizjoy to add a joystick. We also issue a callback here to watch for hat events on the joystick. (Hats are a kind of controller on a joystick or gamepad. You should be able to figure out which part of your joystick is the hat using the joystick demo above.) When the hat is moved, the joyhat function will be called with the "e" argument. The "e.hat" value gives the rotation of the hat button, which the joyhat function uses to rotate the cue. Try running the script and see if your joystick rotates the cue.

```
#Import the vizjoy module and
#add one joystick.
import vizjoy
joy = vizjoy.add()

#This function will handle hat events.
def joyhat( e ):
   game.cue.setEuler( e.hat,0,0 )
#Place a callbacks to watch for
#hat events. These will call
#joyhat whenever a hat event
#occurs.
viz.callback(vizjoy.HAT_EVENT,joyhat)
```

Now we'll add another callback, this time for button events on the joystick. Now joybutton will be called whenever a button is pressed. It will be passed the argument "e" which includes the identity of the pressed button ("e.button"). Try running the script and pressing the buttons on the joystick to rack the balls and set up and shoot the cueball. Depending on the nature of your joystick, you might choose to put different button numbers in this function.

```
#This function will handle joystick
#button events.
def joybutton( e ):
   #Depending upon which button is
   #used, called one of the pool
   #methods.
   if e.button == 7:
      game.set_up_shot()
   elif e.button == 8:
      game.shoot()
   elif e.button == 1:
      game.rack_balls()
   elif e.button == 2:
      game.set_cueball()
#Place a callbacks to watch for
#joystick button events. These will
```

```
#call joybutton whenever a hat event
#occurs.
viz.callback(vizjoy.BUTTONDOWN_EVENT,joybutton)
```

EXAMPLE SCRIPT: JOYSTICK NAVIGATION

In the next example, we'll play pool again but this time we'll control the viewpoint with the joystick. First we'll create a module for joystick navigation. This is a simple module with one function running on a timer. The function samples position and orientation data from the joystick and applies those values to the viewpoint.

```
import vizact
import viz

#Import the vizjoy module and add one joystick.
import vizjoy
joy = vizjoy.add()

def update_joystick():
   #Get the joystick position
   x,y,z = joy.getPosition()
   #Get the twist of the joystick
   twist = joy.getTwist()
   #Move the viewpoint forward/
   #backward based on y-axis value
   #Make sure value is above a certain
   #threshold.
   if abs(x) > 0.2:
         viz.MainView.move(0,0,x*0.01,viz.BODY_ORI)
   #Move the viewpoint left/right based
   #on x-axis value. Make sure value is
   #above a certain threshold
   if abs(y) > 0.2:
         viz.MainView.move(y*0.01,0,0,viz.BODY_ORI)
   #Turn the viewpoint left/right based
   #on twist value. Make sure value is
   #above a certain threshold.
   if abs(twist) > 0.2:
         viz.MainView.rotate(0,1,0,twist,viz.BODY_ORI,viz.RELATIVE_
WORLD)

#UpdateJoystick every frame
vizact.ontimer(0,update_joystick)
```

Save this module as joystick_navigator.py but keep it open because you might want to tweak it to fit your joystick's capabilities once you see how it works in the world. Now open a new script and set up a pool game using the pool module. Import the joystick_navigator.py module you just made and run the script. Play with the joystick and see how well the module works with your joystick. If things don't feel right, try tweaking the joystick_navigator to improve the relationship between joystick movement and movement in the world. It's worth getting right because you can use this module in the future in any world in which you want joystick navigation.

```
import viz
import math

viz.go()

#Import the pool module and
#instantiate a game object.
import pool
game = pool.pool_game()
#Remove the cue.
game.cue.remove()

#Use the viz module to
#enable physics.
viz.phys.enable()

#Set the viewpoint in a good place.
vpos = game.cueball.getPosition()
viz.MainView.setPosition(vpos)
viz.MainView.setEuler(-90,0,0, viz.BODY_EULER)

#Import your joystick navigator.
import joystick_navigator
```

Now we'll attach the cueball to the viewpoint so that we can knock the other balls around. To do this, we'll use linking. The position and orientation of the viewpoint will then be applied to the link destination.

```

#Link the cueball to the view.
link = viz.link( viz.MainView, game.cueball )
#Perform a pre transformation
#on the link in order
#to put the head in front
#of, and slightly below,
#the viewpoint.
```

```
link.preTrans( [0,-.059,.2] )

#Make the cueball transparent
#so we can see through it.
game.cueball.alpha(.2)
```

TRACKING DEVICES

Tracking is a crucial component of all the most powerful virtual reality systems. By tracking the position and orientation of the user's head, we can control the simulation's viewpoint such that the user can navigate through the virtual world just as naturally as they would navigate through the physical world. Tracking devices can also be used to track whole bodies through **motion capture** allowing us to take the movements of an individual in the physical world and map them onto the movements of an **avatar** in the virtual world.

Pulling in tracking data from any of the many devices that are available nowadays can be just as straightforward as linking the viewpoint to the cueball in the previous example.

To see **6DOF** tracking (6 degrees of freedom--3 for position and 3 for orientation) in action, check out the AR demo. This demo is designed to show an example of **augmented reality** (AR), a mix of virtual reality and the physical world. The AR in this demo relies on tracking the position and orientation of a **marker** in the physical world. The program then uses those data to place an avatar in the world. Because her body moves and rotates with the marker, the avatar appears to be standing on the marker in the physical world. For this demo, you'll need to have a web cam working on your computer. You'll also need to print out the pattSample1.pdf from the Teacher in a book demos and examples/art folder and mount it on something rigid (the tracker won't find the marker if the pattern is bent).

EXERCISES

1. Go back to the physics example script from the "Modeling physics" section and adapt it so that you throw knives with joystick events instead of with mouse events.

2. Create a class that uses the joystick to control the position and orientation of any node that you pass it.

OUTPUT DEVICES

Up until now the main output device we've used in this workbook is your computer monitor. Of course, virtual reality hardware goes far beyond 2D monitors. In the visual domain, **head-mounted displays** (**HMD**s) and **cave** systems immerse users to the degree that the virtual world appears to surround them. With HMDs, a set of "goggles" presents the world to each eye separately, affording See "OUTPUT DEVICES" on page 96a stereoscopic experience of the world. When coupled with orientation and position tracking, users can move around the virtual world as naturally as they would move around the physical world, with the scene around them adjusting appropriately to each movement.

Cave environments immerse individuals in worlds by projecting images (usually 3D) of those worlds on the walls surrounding the user. Cave environments also use position tracking to render the worlds appropriately so that users feel as though they're looking beyond the wall into the virtual world. In other words, the images adapt to movement so that users feel more like they are looking through a window than looking at a picture. In this same sense, when you take the users' position into account when rendering a virtual environment on desktop monitors and "power walls", you can also create the feeling of looking through the fixed 2D surface into the virtual world.

Output devices aren't limited to visual displays. They also include audio systems that simulate 3D sound. The most basic of these is traditional stereo sound systems, although more advanced technology does a better job of simulating depth and direction of sound and sound reflection. In recent years, **haptic** devices have also emerged to simulate touch, allowing users to feel the surfaces of virtual models. While they are not yet widely distributed, other technologies even simulate scents and flavors.

While some hardware requires special plug-ins, other hardware is universal enough to have generic modules in Vizard's library of commands. Either way, Vizard will likely have a method for interfacing with whatever output devices you have handy. Check out the Hardware section of the Vizard documentation for information on specific devices.

EXAMPLE SCRIPT: STEREO VISION, STEREO SOUND AND HAPTIC OUTPUT

In this example, we'll add more domains of output to the pool game we were working on in the previous section. So, let's start by dropping in the code from the joystick navigation tutorial.

```
import viz
import math
viz.go()
```

```
#Import the pool module and
#instantiate a game object.
import pool
game = pool.pool_game()

#Remove the cue.
game.cue.remove()

#Import your joystick navigator.
import joystick_navigator

#Link the cueball to the view.
link = viz.link( viz.MainView, game.cueball )
#Perform a pre transformation on
#the link in order to put the
#head in front of, and slightly
#below, the viewpoint.
link.preTrans( [0,-.059,.2] )

#Make the cueball transparent
#so we can see through it.
game.cueball.alpha(.2)

#Set the viewpoint in a good place.
viz.MainView.setPosition( game.cueball.getPosition() )
viz.MainView.setEuler(-90,0,0, viz.BODY_EULER)

#Use the viz module to enable physics.
viz.phys.enable()
```

Now let's add sound to the world. In general, there are two ways to add sound to a world. The first way is to simply play a sound file within the world. If you do this, the user will hear sound just as they would if you were playing it on any other software on the computer. As such, the sound is not localized to any specific part of the world. This works well for adding ambient noise. The other way to add sound is to use localized sound. Localized sound emanates from a node such if the node is to your right, the sound will come out of the right speaker and the closer you are to that node, the louder the sound will be. First we'll add some ambient noise to the world by simply playing a sound file in the world. After adding the code below, run the script.

```
#Add ambient noise. Set it to a looping mode
#play it and set the volume with methods
#from the multimedia:av library.
ambient_noise = viz.addAudio( 'art/pool hall.wav' )
```

```
ambient_noise.loop( viz.ON )
ambient_noise.play( )
ambient_noise.volume( .5 )
```

Next we'll add some localized sound. We'll use a collision event to do this so that every time there's a collision with one of the pool balls, the sound of two pool balls colliding will play from that node. Notice also that we set a 3D sound parameter when we play the sound here. The minmax parameter sets the distance at which sound no longer gets louder as you get closer (min) and the distance at which sound no longer gets softer the further you go away (max). (If you're not working on a fast computer, you might want to comment out the code for adding ambient noise. Sound processing takes memory and doing this all at once might overwhelm your computer.)

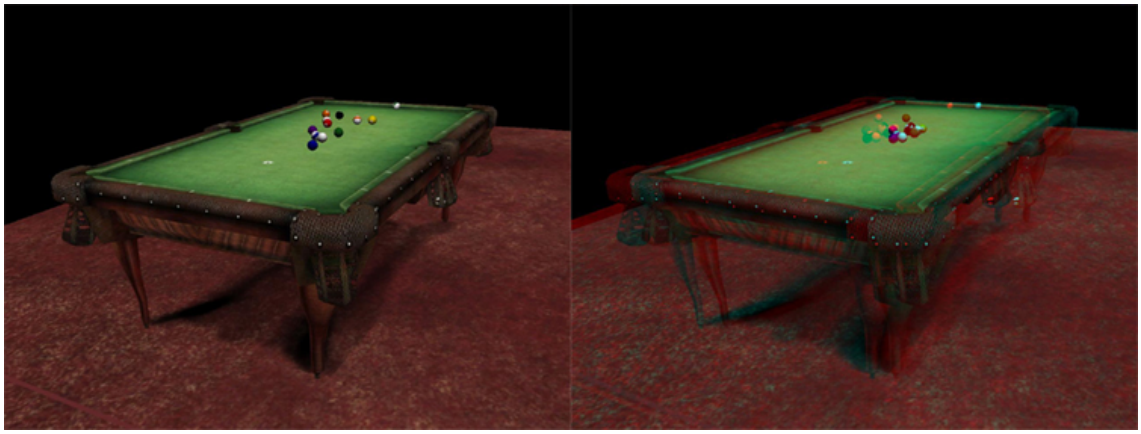
```
def collision(e):
   #If one of the balls hits,
   if game.balls.count( e.obj1 ):
      #Play a sound at the node.
      s= e.obj1.playsound( 'art/pool ball sound.wav' )
      #Set the min and max distances
      #for the sound.
      s.minmax(0,.2)
viz.callback(viz.COLLIDE_BEGIN_EVENT, collision )
```

Now run the script while wearing headphones. You should hear the sound of collisions on either side of you.

Now let's add haptic feedback to this world. This part of the script will only work if the joystick you're using has force feedback (or rumble effects). We'll add the feedback to the collision event function we just created. Any time that the cueball hits another ball, the collision task will apply a force to the joystick and then remove the force a fraction of a second later.

```
#Define a task to handle the force feedback.
import viztask
def force_task():
   #Add the force to the joystick. Since we
   #already added a joystick with joystick_navigator
   #we'll use that one.
   joystick_navigator.joy.addForce(0,1)
   #Wait .2 seconds.
   yield viztask.waitTime(.2)
   #Set the joystick's force at 0.
   joystick_navigator.joy.setForce(0,0)

def collision(e):
   #If one of the balls hits,
```

```
   if game.balls.count( e.obj1 ):
      #Play a sound at the node.
      s = e.obj1.playsound( 'art/pool ball sound.wav' )
      #Set the min and max distances
      #for the sound.
       s.minmax(0,.2)
   #If the object that gets hit
   if e.obj2 == game.cueball:
      #is the cueball, then schedule the force_task to
      #apply a force to the joystick.
      viztask .schedule( force_task() )
viz.callback(viz.COLLIDE_BEGIN_EVENT, collision )
```



Finally, we'll add stereo visuals to the world. This effect will only work if you have a pair of anaglyphic glasses (the red and blue glasses that they used to use for 3D movies) or an HMD. While the ways in which Vizard renders stereo is different between an HMD and a fixed surface, adding them to your script is similar. Just pass Vizard the appropriate flag when you call viz.go.

```
viz.go( viz.HMD | viz.STEREO )
```

or

```
viz.go( viz.ANAGLYPHIC )
```

EXERCISES

1. Go back to the "Transformations and the scene graph"example from the Modeling chapter's section, "The scene graph". Add the 'buzzer.wav' as 3D sound to the bees node. Make sure to loop the sound so they continue to buzz

2. If you have an HMD or stereo glasses, try playing with the viz.ipd and viz.fov commands to manipulate the interpupillary distance and the field of view.

NETWORKING WORLDS

Networking greatly expands the potential of virtual environments. For one thing, it allows users to inhabit the same virtual world. As such, people can socialize, negotiate, play games, work together or otherwise interact in an entirely digital context. Moreover, you can be in a completely different part of the planet and still experience the "physical" presence of another person. In addition to creating shared environments, you can set up a netSee "NETWORKING WORLDS" on page 100work so that the events on a **client's** computer are controlled by a **master**, giving you the ability to make one user the puppeteer of another user's environment.

See "NETWORKING WORLDS" on page 100Networking also helps solve some technical challenges. For example, networked computers can be used to render the same scene. So, if you are using multiple monitors or projectors to display your world (i.e. a cave virtual environment) each computer can handle the rendering of its own display while staying in sync with the others. For more on using networking to render the same See "NETWORKING WORLDS" on page 100viewpoint on multiple computers, check out the Vizard documentation on **vizcave** and **clustering**.

Another technical advantage to networking worlds is that you can spread out some of the computing responsibilities to other computers in your network if you are running a computationally expensive world . For example, if you are doing a great deal of physics modelling, you can delegate those computations to one computer and let another computer (or computers) deal only with the more basic transformations of the scene graph (see the "Networking worlds" example below for a demonstration).

SHARING DATA

For the most part, networking worlds is straightforward. The key is to minimize the amount of data that each computer sends and receives. We do that, in part, by running as much as we can locally. So, we load the world on each machine and then pass only the data that is unique to the local computer.

For example, let's say we want to make a world with a vast topiary maze with chirping birds and slowly moving clouds. Let's also say that we want to put a group of users in that maze and represent them as little wheelbarrows driving around. The maze model, the bird sound, the cloud animation, and even the wheelbarrows are going to be the same for each of the users. As such, we can load and run all of those things locally on each computer. The only thing that each computer will need to know from

each other computer is the position and orientation of the other users' wheelbarrows. So, the data we send back and forth is just the position and orientation of the users.

Run the network demo now to see which computers you can access in your network. Use the textbox to enter the IP address of a computer you'd like to hook up to. If you make the connection, the wheelbarrow in the world should move with the orientation and position of that other user's viewpoint.

EXAMPLE: NETWORKING WORLDS

In this example, we'll put three users into the pool game world. Two of the users will be **clients,** who each control a cueball. The other computer will be the **master**. Because the outcome of physics simulations is variable, it's a good idea to have all the physics modelling done at one, central location and sent out to the clients. Cutting down on the physics calculations will also help the clients run more efficiently. So, we'll have the master handle physics.

We'll write different scripts for the clients and the master, starting with the clients' script. To set up the world, we'll use some of the same content that we used in the "Joystick navigation" tutorial from the "Input devices" section. These will load the pool game on each client computer (although we won't activate physics on the client end):

```
import viz
viz.go()

#Import the pool module and
#instantiate a game object.
import pool
game = pool.pool_game()

#Remove the cue.
game.cue.remove()

#Import your joystick navigator.
import joystick_navigator

#Link the cueball to the view.
link = viz.link( viz.MainView, game.cueball )
#Perform a pre transformation on
#the link in order to put the head
#in front of, and slightly below,
#the viewpoint.
link.preTrans( [0,-.059,.2] )
```

```
#Make the cueball transparent so
#we can see through it.
game.cueball.alpha(.2)

#Set the viewpoint in a good place.
viz.MainView.setPosition( game.cueball.getPosition() )
viz.MainView.setEuler(-90,0,0, viz.BODY_EULER)

#Link the cueball to the view.
link = viz.link( viz.MainView, game.cueball )
link.preTrans( [0,-.059,.2] )
```

Because we want the client to see the other opponent's cueball, we'll add an extra one.

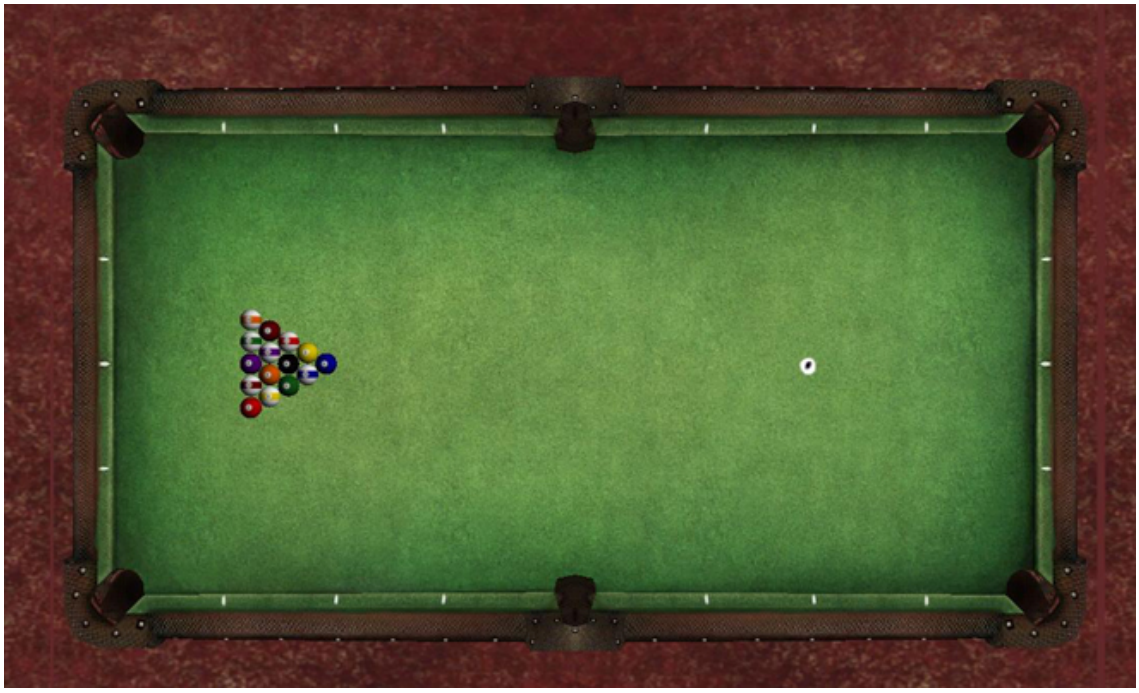
```
#Add a cueball to represent
#the other client.
other_cue = pool.pool_ball( 0 )
```

Now we'll add the networking component. Since the clients need to communicate with the master, we'll set up a network object for them to do so. We'll start by giving the name of the master computer. This can be the name on your local network or its IP address.

```
#Name the master computer.
MASTER = 'WARRIOR'

#Add a network for the master.
master_net = viz.addNetwork( MASTER )
```

Next we'll add a function to send the master the position of the client's own cueball. We'll execute this function every frame so that the master has as much information coming in as possible about the cueball's whereabouts.

```
def send_box():
   #Send the position of the
   #cueball to the master.
   pos = game.cueball.getPosition()
   master_net.send( cue_data = pos )
#Send the data to the master as
#frequently as possible.
vizact.ontimer(0, send_box )
```



Finally, we'll set up a network event to handle incoming messages from the master. All the data coming in from the master are in the "e" object. The master will be sending data in two variables, "balls_data" and "cue_pos". As such, they'll show up as e.balls_data and e.cue_pos in our network event. We'll then use those data to animate all balls on the table.

```
def onNetwork(e):
   #This function will handle network
   #events.
   if e.sender == MASTER:
      #If the event is caused by a
      #message from the master,
      #Take the data and animate
      #all the non-cueballs.
      for i in range(len( game.balls ) ):
         game.balls[ i ].setEuler( e.balls_data[ i ][ 0 ] )
         game.balls[ i ].setPosition( e.balls_data[ i ][ 1 ] )
      #and take the data about the
      #other player's cueball to
      #animate that cueball.
      other_cue.setPosition( e.cue_pos )
viz.callback(viz.NETWORK_EVENT,onNetwork)
```

Now save that as the client script and open a new script for the master. This script will again start by loading a pool game.

```
#Import the pool module and
#instantiate a game object.
import pool
game = pool.pool_game()

#Remove the cue.
game.cue.remove()

#Use the viz module to enable physics.
viz.phys.enable()

#Set the viewpoint in a good place.
viz.MainView.setPosition( 0,5,0 )
viz.MainView.setEuler(0,90,0)
```

Next we'll set up outgoing networks for the two clients. For the two CLIENT constants, you should put the IP addresses or names of two computers in your network.

```
#Name the client computers
CLIENT1 = 'DREADNOUGHT'
CLIENT2 = 'CALEDONIA'
clients = [CLIENT1, CLIENT2 ]

#Set up a dictionary of networks with
#an entry for each computer.
nets = {}
nets[ CLIENT1 ] = viz.addNetwork( CLIENT1 )
nets[ CLIENT2 ] = viz.addNetwork( CLIENT2 )
```

Then we'll grab a couple cueballs to assign to the two clients and put them in a dictionary.

```
#Get cueballs for each client.
cueballs = {}
cueballs[ CLIENT1 ] = game.cueball
cueballs[ CLIENT2 ] = pool.pool_ball( 0 )
```

Then we'll write a little function that grabs the position and orientation data of all the poolballs (besides the cueballs) and puts those data into an array.

```
def ball_info():
   #Gather the orientation and positions of all the
   #non-cueballs.
   array = []
```

```
   for ball in game.balls:
      this_ball = []
      this_ballappend( ball.getEuler() )
      this_ball.append( ball.getPosition() )
      array.append( this_ball )
   return array
```

When we send the data, we'll send it in two variables, cue_pos (the position of the opposite player's cueball) and balls_data (the positions and orientations of all the other balls as gathered by the ball_info function above) to the two clients.

```
def send_box():
   #Send the data to the two clients.
   #We send them the positions and
   #orientations of all the non-cueballs
   #along with the position and orientation
   #of the other client's cueball.
   b = ball_info()
   nets[ CLIENT1 ].send( balls_data = b, cue_pos = cueballs[ CLIENT2
].getPosition() )
   nets[ CLIENT2 ].send( balls_data = b, cue_pos = cueballs[ CLIENT1
].getPosition() )
#Send the data as frequently as possible.
vizact.ontimer(0, send_box )
```

Finally, we'll set up a network callback to watch for data coming in from either client. We'll use these data to position that client's cueball.

```
def onNetwork(e):
   #This function will handle network events.
   if clients.count( e.sender ):
      print e.sender
      #If the data are from a known sender, use it
      #to translate and orient the sender's cueball.
      cueballs[ e.sender ].setPosition( e.cue_data )
#Callback for network events.
viz.callback(viz.NETWORK_EVENT,onNetwork)
```

EXERCISES

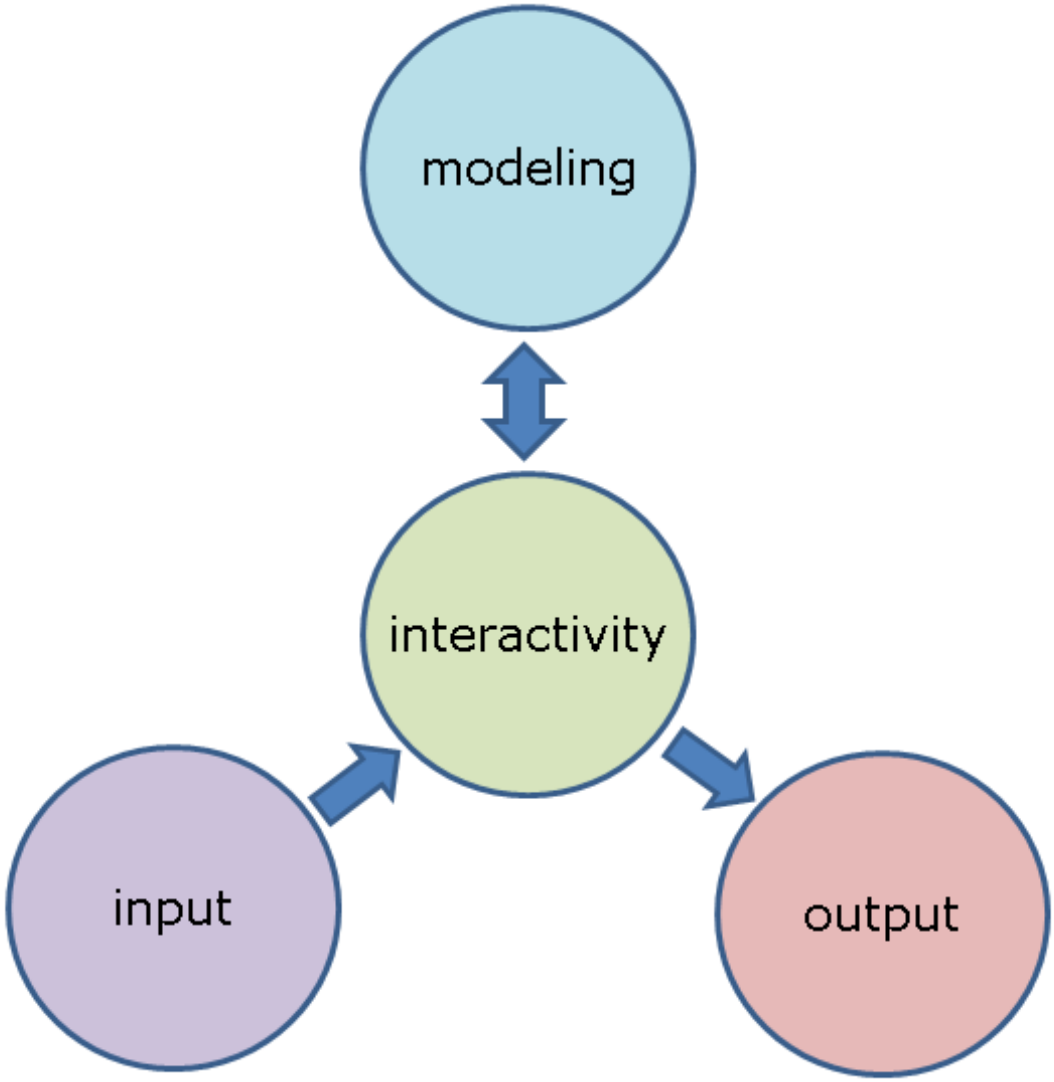
1. Write a networked world for two users. In the script, add an avatar to represent the other user. Send and receive the orientation and position of the viewpoint. Use the incoming data to animate the behavior of the avatar.

2. Tweak the networked pool game above so that both master and client have a cue stick that is controlled by the master.

PLANNING YOUR WORLD

GATHERING RESOURCES

As Gathering resources the many sections in this book illustrate, there's a lot going on in virtual worlds. Pulling together all those details takes planning. When you're beginning a new project, it helps to think about the basic components of a virtual world: what do you need to model, what input devices will you be using, what output devices will you be using, and how will you pull it all together so that it produces an interactive and immersive experience?



The "Project Plan- EXAMPLE" workbook in the Teacher in a Book demos and examples directory provides a sample plan for a script. To check out the world that this plan was written for, run "program flow example.py" from the same folder.

It's helpful to think of a virtual world in terms of the different states it might go through, states in which the nature of the input, output, and modeling vary. The first page in the project plan workbook ("program flow") breaks down the sequence of the example program with a flowchart. The leftmost sequence shows the basic states that the program goes through. To the right on the sample flowchart are the more specific states that happen within each of the four general states. As we'll go over in the following section, a flowchart like this can be translated directly into code through tasks. As such, a flowchart is not just a good brainstorming tool but will help determine your code.

Note that for each state, the flowchart describes how that state ends. Determining what will drive the transition of one state of your program to another is obviously critical to your program and will help you determine the nature of the input you'll need to use to push your program forward.

In the other pages of the project plan are worksheets for breaking down what kinds of models you need, what kind of input and output devices you'll use, and what kinds of events and sampling you will incorporate to make your world interactive. Included in this example are potential sources for things like 3D models or sound samples. When it comes time for you to create your own virtual world, try filling out this worksheet.

Program flow

USING TASKS TO CONTROL PROGRAM FLOW

A lot can happen between the beginning and ending of a virtual world. Making sure everything happens in order and in response to the user's actions can be a challenge. One tool that's particularly helpful in this domain is **tasks**. Tasks (created with the viztask module) are functions that can pause and wait for a **condition** to be met before they proceed. They do this without interrupting the flow of the rest of the program. So, let's say we're creating a simulation of a city with traffic travelling through an intersection. We may want the cars to pause until the user has crossed the street. As such, we'll need to pause traffic's movement once the user begins to cross the street (the condition) and start it up again once he or she has crossed. We don't, however, want to pause everything else in the simulation. A task might be helpful in this situation.

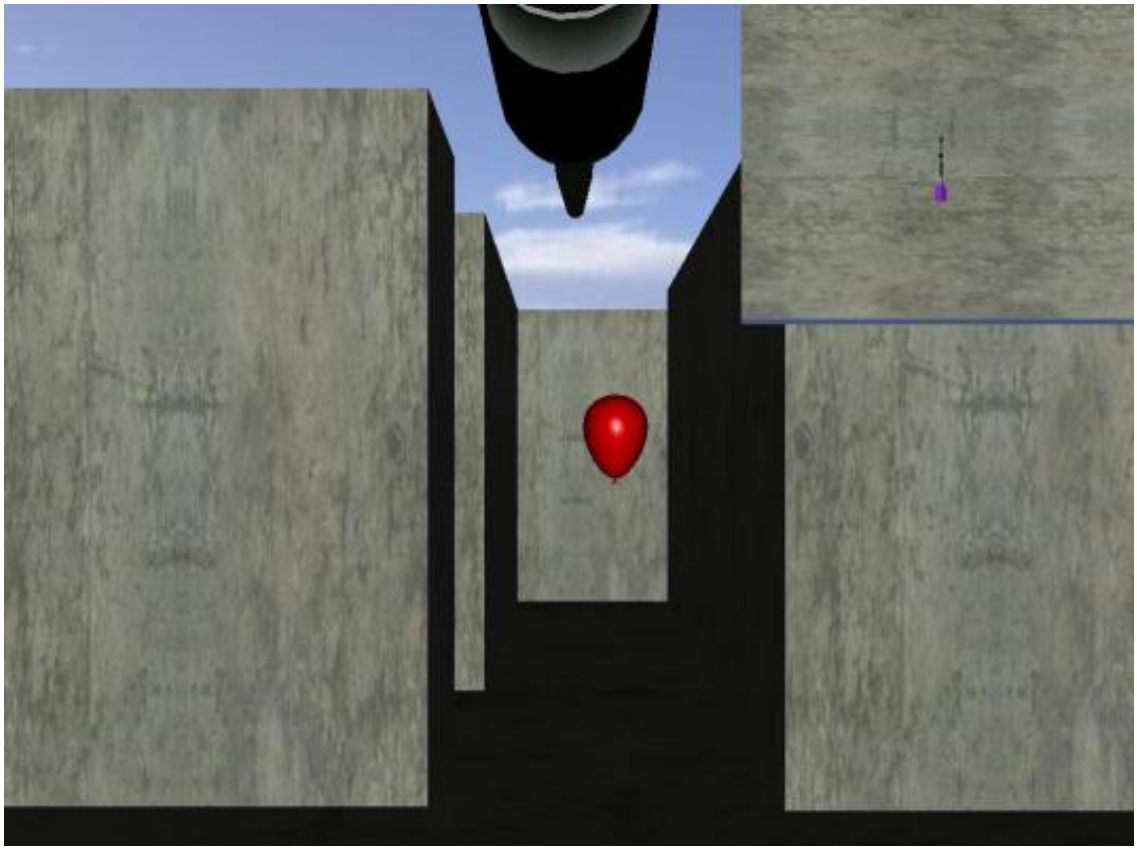
Tasks are also helpful at splitting up your program into its different periods of activity. For example, imagine you're building a world in which people are learning to complete a welding task. During the first period in the world, users might repeatedly work on a set of materials to gain skills. In the next period, you might want to test their knowledge with a novel set of materials. In a case like this, you could break your world up into two phases and use a task function to first set everything into motion for the training phase and then wait until that phase is over before setting everything into motion for the learning phase. As we mentioned in the previous section, this kind of breakdown of a program can be done with a flowchart and translated directly into a task.

Tasks can be nested such that one task can call on **subtasks** and wait for them to be completed before proceeding. As such, individual states of a world can have their own task and then signal back to the main task that they have finished. There are a variety of conditions for which a task can pause, or **yield**. Most of these conditions are standard events, although you can define your own conditions if they are unique to the situation.

In the following examples, we'll go over how the world described in the flowchart from the "Project plan - EXAMPLE" file translates into a set of tasks.

EXAMPLE: PROGRAM FLOW

While going through this example, we'll refer to the first sheet of the "Project plan - EXAMPLE" file, so you might want to get it out. This world is a game. Once the user is ready, they hit a key and the world begins. First, they receive some instructions and then the game itself starts. During the game, players navigate around a balloon-filled maze with a dart attached to their head. Their goal is to pop as many balloons as possible before their time runs out. Once the game is over, the program tells the user if they won and asks if they want to play again.



To get things started, we'll add all the resources we need to the script:

```
import viz
import viztask
viz.go()

### Add all the resources. ###############

#Add a sky with an environment map.
env = viz.add(viz.ENVIRONMENT_MAP,'sky.jpg')
dome = viz.add('skydome.dlc')
dome.texture(env)

#Add a maze model.
maze = viz.add('art/maze.ive')

#Add balloons.
balloons = []
for pos in [[.2,3.4],[-3.2,6.8],[-9.8,23.4],
```

```
   [6.4,30.2],[-13.0,33.4] ]:
   balloon = viz.add('art/balloon.ive' )
   balloon.setScale( 2,2,2 )
   balloon.setPosition( pos[0],1.7,pos[1] )
   balloon.color( viz.RED )
   balloon.specular( viz.WHITE )
   balloon.shininess( 128 )
   balloons.append( balloon )

#Add a popping sound.
pop = viz.addAudio( 'art/pop.wav' )

#Turn on viewpoint collision and set
#its distance buffer.
viz.collision( viz.ON )
viz.collisionbuffer( .1 )

#Add a subwindow and associated it
#with a viewpoint.
subwindow = viz.addWindow()
subview = viz.addView()
subwindow.setView( subview )
subwindow.setSize( .35,.35 )
subwindow.setPosition( .65,1)
subwindow.visible( viz.OFF )

#Link the subview to the position
#of the main view but put it up a distance.
subview_link = viz.link( viz.MainView, subview )
subview_link.setMask( viz.LINK_POS )
subview_link.setOffset( [0,8,0] )
subview.setEuler( [0, 90, 0 ])

#Link a dart to the main view.
dart = viz.add( 'art/dart.ive' )
dart.setScale( 2,2,2)
link = viz.link( viz.MainView, dart )
link.preTrans( [0,.15,0] )

#Add text fields to a dictionary.
text_dict = {}
for kind in ['score','instructions','time' ]:
   text = viz.addText('', viz.SCREEN )
   text.setScale( .5,.5)
```

```
   text.alignment( viz.TEXT_CENTER_BASE )
   text.alpha( 1 )
   text_dict[ kind ] = text
text_dict['score'].setPosition( .1,.9 )
text_dict['instructions'].setPosition( .5,.5 )
text_dict['time'].setPosition( .1,.85 )


#Add a blank screen to the viewpoint to
#block out everything in the beginning.
blank_screen = viz.addTexQuad( viz.SCREEN )
blank_screen.color( viz.BLACK )
blank_screen.setPosition( .5, .5 )
blank_screen.setScale( 100,100 )
```

Now we're ready to add the functions that will run the show. We'll add the basic tasks now and then pull them together with one big task at the end. This first function will set the stage for the world:

```
def set_the_stage():
   #Put the viewpoint in the right position and freeze
   #navigation.
   viz.MainView.setPosition(0,1.8,-3)
   viz.MainView.setEuler(0,0,0)
   viz.mouse( viz.OFF )
   #Make the instructions text appear.
   text = text_dict[ 'instructions' ]
   text.alpha( 1 )
   #Put a message in that text.
   text.message( 'Press s to begin.' )
   #Wait for the s key to be hit.
   yield viztask.waitKeyDown( 's' )
   text.message( '' )
```

The yield statement is what makes this function a task. In this case, we're just yielding to a keydown event. To call a task function, you need to schedule it. Add the following line to your code to schedule the function above:

```
#Schedule the above task.
viztask.schedule( set_the_stage() )
```

Comment out those lines and add the next task. This task will display the instructions, a series of sentences that will fade in and out of view. Here we'll use yield statements that add an action to a node and then wait for it to complete. We'll also use a yield statement for a timer event.

```
def game_instructions():
   text = text_dict[ 'instructions' ]
   text.alpha( 1 )
   sentences = ['You will get one point for each balloon that you
pop.',
   'You are racing against the clock.',
   'Get ready . . .' ]
   for sentence in sentences:
      text.alpha(0)
      text.message( sentence )
      #Add a fading action to the text and wait.
      yield viztask.addAction( text, vizact.fadeTo(1, time = 1 ))
      #Wait a second.
      yield viztask.waitTime( 1 )
      #Wait to fade out.
      yield viztask.addAction( text, vizact.fadeTo(0, time = 1 ))
```

Try scheduling this task to see how it runs.

Next we'll write two different tasks to deal with the two possible ways the game might finish. One task will keep track of how much time has passed and the other will keep track of how high the score is. Whichever ends first will determine whether the player won or lost. Note in the balloon_popping_task that we yield to a collision event and are then able to get data from that event with a viz.Data object.

```
def game_timer_task():
   #Grab the text field for
   #time.
   text = text_dict[ 'time' ]
   text.alpha(1)
   text.message( 'Time: 0' )
   #Loop through as long as time
   #is under a certain number of
   #seconds.
   while time < 30:
      yield viztask.waitTime( 1 )
      time+= 1
      text.message( 'Time: ' + str( time ) )

def balloon_popping_task():
   #Grab the text field for
   #the score.
   text = text_dict[ 'score' ]
   text . alpha(1)
   score = 0
```

```
   text.message( 'Score: 0' )
   #Loop through as long as the score
   #is below the winning limit.
   while score <5:
      #Create a data object to accept
      #data from the event.
       data= viz.Data()
      #Yield for collision events.
      yield viztask.waitEvent( viz.COLLISION_EVENT, data )
      #From the data object, get the object
      #that the viewpoint collided with.
        intersected_object = data.data[0].object
      #If it was a balloon, pop it and
      #add a point to the score.
      if balloons.count( intersected_object ):
         pop .play()
         score += 1
         text .message( 'Score: ' + str( score ) )
         intersected_object.visible( viz.OFF )
```

The next task is the game itself. Here we'll schedule our two previous tasks and wait for either one of them to finish. This time a viz.Data object will contain the data for which condition was met (which task finished). That will tell us whether or not the player won.

```
def game():
   #Begin the game.
   #Turn on mouse navigation.
   viz.mouse( viz.ON )
   #Make the subwindow visible.
   subwindow.visible( viz.ON )
   #Get rid of the blank screen
   #that blocks the view.
   blank_screen.visible( viz.OFF )

   #Create two tasks for two outcomes of game.
   balloon_popping = viztask.waitTask( balloon_popping_task() )
   time_passing = viztask.waitTask( game_timer_task() )

   #Wait for the game to end.
   #Create a data object that
   #we can pass to the next yield
   #statement.
   data = viz.Data()
   #Wait for the game to end one way
```

```
   #or another.
   yield viztask.waitAny( [balloon_popping, time_passing], data )

   #Once the game has ended, hide things.
   viz.mouse( viz.OFF )
   blank_screen.visible( viz.ON )
   subwindow.visible( viz.OFF )
   viz.MainView.reset(viz.HEAD_ORI | viz.HEAD_POS| viz.BODY_ORI)

   #Give different feedback depending on
   #how the game ended.
   text= text_dict[ 'instructions' ]
   if data.condition == balloon_popping:
      text.message( 'GAME OVER, YOU WON!' )
   elif data.condition == time_passing:
      text.message( 'GAME OVER, YOU LOST.' )
   text.alpha( 1 )
   text_dict[ 'score' ].alpha(0)
   text_dict[ 'time' ].alpha(0)
   #Wait a moment.
   yield viztask.waitTime( 4 )
```

Now we'll add a another task that yields to a keydown event and then proceeds.

```
def play_again():
   #Ask a question.
   text_dict[ 'instructions' ].message( 'Want to play again (y/n)?' )
   #Create a data object to accept the
   #event's data.
   data= viz.Data()
   #Yield to a keydown event.
   yield viztask.waitKeyDown(('n','y'),data )
   #If the key that was pressed
   #is 'n', quit.
   if data.key == 'n':
      viz.quit()
   #Otherwise reset the world.
   if data.key == 'y':
      for balloon in balloons:
           balloon.visible( viz.ON )
      for value in text_dict.values():
           value.alpha(0)
```

Finally, we can add and schedule the main sequence of events. This task represents the furthest lefthand column of the flowchart. Here we'll yield to each of the individual

subtasks that we wrote above. Note that the "while:" statement at the beginning of the tasks allows it to loop indefinitely.

```
#Set up a task to handle the main
#sequence of events.
def main_sequence():
   while True:
      #Set the stage for the game.
      yield set_the_stage()

      #Begin with instructions.
      yield game_instructions()

      #Play the game.
      yield game()

      #See if the user wants to play
      #again.
      yield play_again()

#Schedule the main sequence task.
viztask.schedule( main_sequence() )
```

EXERCISE

In the "Teacher in a book demos and tutorials" directory, there is a script entitled "The viztask challenge setup.py" and a movie file of the same name. See if you can put the code within the .py file into a task such that when you run the program it produces the same outcome as the movie

GLOSSARY

A

array
An ordered list.

C

comments
Lines in a script that don't get executed, designated with a "#" sign.
constant
A value that stays constant throughout a script.

D

dictionary
A data type in Python. Dictionaries have keys and values. To pluck a value out of a dictionary, you provide the key.

F

frame rate
The number of frames rendered per second.

S

strings
Text values.

T

textures
Image files that are "wrapped" on 3D model geometries to simulate realistic surfaces.

V

variable
A value that changes throughout a script.

INDEX