Vizard 4
Teacher in a Book

作者 **Cade McCall**

翻译 李晓鸥

WorldViz

2012 年 7 月印制

目录

前言

本教程介绍了通过 Vizard 来构造虚拟世界的基本方法。教程中的所有实例都可以从 WorldViz 的网站上下载 www.worldviz.com/learn。该教程分为四个部分。在基础编程这个部分介绍基于 Python 的编程基础（Python 是 Vizard 软件的程序语言）。建模部分介绍如何构建和模拟动态三维环境。硬件部分介绍多种输入和输出设备。最后一章，规划你的世界部分介绍如何为大型程序收集资源并设置各种事件序列。

基础编程

简介

在这一部分我们主要学习使用 Vizard 写一些简单的代码，Vizard 是构建虚拟场景的一个软件工具包。了解了 Vizard 的基本代码结构后本书中用到的实例代码会更容易理解。如果你已经有编程经验，并且了解 Python 程序语言，那么你可以略过这一章。如果你对你的编程技能信心不足，请认真阅读此章。完成本章节的阅读后你可以轻松理解本章节的实例代码并完成章后作业练习。

首先，你需要知道在 Vizard 里的程序编写是完全通过 Python 程序语言来实现的。Python 程序语言应用广泛，Vizard 是 Python 的一个应用程序。Python 作为开源语言，拥有巨大的用户群以及海量的免费资源。基于此你可以轻松的从它处获得相关的编程指导和脚本工具。关于本章节中介绍的 Python 编程基础和技巧你也可以通过阅览 Vizard 帮助文件中的 Python 部分获得，或者阅览 Python 官网（www.python.org）. Python 本身的帮助文件里面就包括它的经典教程。建议初学者在开始编写 Vizard 脚本之前认真阅读 Python 帮助文档中的 3、4、5 节。

Vizard 软件提供了编写 Python 代码的界面，并且它为用户提供一个巨大的虚拟现实函数库。让我们从最简单的开始吧。运行 Vizard 4.0（通常在开始菜单中的 Worldviz 目录下可以找到 Vizard）。这一章不详细介绍 Vizard 的界面功能，但如果你想了解的话，请参考 Vizard 帮助文档。我们这里先关注基本功能。

首先建立一个新脚本。选择“文件”，在下拉菜单中选择“New Vizard File”。这时会弹出一个对话框，选择“Empty File”然后点击“OK”。一个空的 Python 脚本文件就出现在 Vizard 界面中了。再次选择“文件”，在下拉菜单中选择“Save”保存文件。然后我们开始下面一节。

常量、变量和数据类型

一个脚本有若干个组成部分，我们下面一一介绍。首先，这些组成部分中最基本的概念就是常量和变量。常量和变量就是在脚本中你要涉及到的对象的名字（例如，一个数字）。一旦你定义了一个变量或者常量，那么当你要用到这些变量和常量时就可以直接调用他们的名字。变量和常量的区别是在程序的任意一个环节你都可以改变变量的值，但是一旦赋予一个值给一个常量，这个值将不能再改变。

在下面一段代码中定义了一个常量（MY_CONSTANT）和一个变量（my_variable）的值，并且打印他们的值。所有以#开头的段落都是**注释部分**。#被 Python 自动识别为注释符，在注释符后面的部分不会被执行。你可以在笔记的部分前加注释符，也可以在不想被执行的代码前加。拷贝下面这段代码到 Vizard 中然后点击面板中的“run”，看看会有什么结果。

```
#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
```

上段程序中 Print 命令告诉 Vizard 在输入输出窗口打印变量和常量的值。运行该段程序之后，检查输入输出窗口中打印的四个值。如果代码有误，你会看到窗口中的报错语句。一般情况下，通过报错语句可得到具体的错误原因。无论在何时运行程序，检查窗口中是否有错误信息非常重要。（提示：输入输出窗口是一个用来检测单行代码功能的非常方便的工具）

你可能注意到了上面代码中的变量名是小写而常量名是大写。这是一个约定俗成的习惯，但也非必须这样写。然而，要注意 Python 程序语言区分大小写。所以无论你习惯用大写还是小写命名变量和常量，一定要保持在整个脚本中一致，否则 Python 会认为是不同的名字。

在没有定义一个变量或者常量之前，你不能调用它。如果调用一个未被定义的变量或常量，程序运行会出错。在下面这段代码中，加进一个语句即打印一个未被定义的变量（我们在这里叫它“new_variable”）

```
#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
```

现在再运行该段代码，你会发现在输入输出窗口中有条报错信息。那么我们如何解决这个问题呢？我们在下段代码中添加一条定义 new_variable 的语句。

```
#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
```

从上段代码中你可能已经了解了，变量和常量可以处理任意一种数据类型。最常见的就是数字、字符串、列表和字典。字符串简单理解为一组字。上面的代码中 MY_CONSTANT 的值就是**字符串**。**列表**和**字典**就是有序的一组值。一旦定义了一个列表，你就可以通过索引的办法来提取列表里面的值，索引就是用来定义它对应值的一组数字。列表中第一个项目的索引值为 0，第二个项目的索引值为 1，以此类推。请看下面的代码。

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

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

字典和列表类似，但是字典里的一组值没有顺序。通过键来获取值。从这一点上，字典这种数据结构和真正的字典很像，通过查找字（键）来获取它的定义（值）。下面这段代码中定义了一个字典，并通过键来取值。

```
#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' ]
```

这里有很多方法来处理字典和字符串，例如拼接 slice、添加 append。在继续这个教程之前，建议你去阅读 Python 关于不同数据结构和方法的文档。你可以直接去查找在 Vizard 帮助文档下的 Python 文档，具体内容在 section 3 “An Informal Introduction to Python”。

基本逻辑结构

本章主要介绍程序的逻辑结构。我们只主要介绍其中的一些，读者若要掌握更全面的逻辑结构知识需要参考 Python 教程中的 3.2 章和 4.1 到 4.5 章（Python 文档在 Vizard 的 help 文档的下拉菜单中可以找到）。

在本章中的例子中，我们会用到“**while**”、“**for**”和“**if**”语句。“while”和“for”都是循环语句，循环语句内的命令在满足循环条件时会循环执行，也就是“满足条件时运行下面的语句”。运行下面的脚本，注意“while”下面的语句前要有缩进：

```
my_variable = 0

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

运行这段程序，你会看到在输入输出窗口打印了一列数字。注意“while”中的语句会一直执行直到“while”的条件（my_variable < 5）不再满足。

重要提示：在 Python 程序语言中缩进非常重要。循环内部的语句只有在前面有缩进的时候才会运行（可以试着去掉上段代码中最后两行的缩进，运行程序会报错）。缩进对其他逻辑结构同样重要，我们会在这一章以及本书的其他实例中介绍。

“for”语句和“while”语句原理相同，但“for”是遍历列表或者数组中的元素，也就是“当元素在列表或者数组中时运行下面的语句”。运行下面的脚本：

```
#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
```

这段脚本遍历了在“my_array”这个列表中的每一个元素。每一次运行循环语句时，“element”就被设定为列表中的一个值。你可以试着将“for”后面的条件语句中的“my_array”替换为“range(5)”，如下面这段脚本所示：

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

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

range 这个函数用来生成一个数组，数组的元素个数由它的参数决定（在上个例子中为 5）。除非特殊声明，数组的元素从 0 计起然后依次递增。如果你还是不清楚的话，可以尝试在输

入输出窗口键入“range(5)”看看结果是什么。输入输出窗口是一个非常方便的地方可以用来检查单个命令。

“if”语句用来检验一个条件。如果条件为真，执行 if 内部的语句。否则不执行。如果“if”下面配有“else”语句，当“if”条件语句为假时，“else”内部的语句被执行。试着运行下面的代码。

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

函数

想象将一个公式在一段代码中重复使用，你可以在每一处需要这个公式的地方把它写出来。如果有上百处需要这个公式的地方，代码不仅冗长而且会给编程带来很多麻烦。然而，你可以利用**函数**来解决这一问题。假设你的代码是一个公司，函数就是有专门技能的雇员，每一次当公司遇到这方面的问题时就会找这个雇员来解决这个问题。同样，当需要一个函数时，就**调用**它。当调用函数的时候，你可以传递函数的参数然后该函数会**返回**一个值。

定义一个函数，该函数就被创建。当一个函数被定义后，你可以随意次数的调用它。让我们从定义一个最简单的函数开始。这个函数的功能是 20 乘以我们传递的变量的值，然后返回结果值。我们定义函数的时候用一个 def 语句：

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

函数在调用时才执行。这里我们再添加一些代码来调用之前定义的函数，在每一次调用之前定义的函数时都将传递下面这个数组的每一个成员。将下面的代码添加到之前的代码中，然后运行。

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

在程序中其中一个重要的概念就是**全局**（global）变量和**局域**（local）变量。全局变量在整个代码中都可被访问，而局域变量仅可在局部被访问。也即是一个函数可以访问全局变量和自己内部的局域变量，该函数不可以访问其他函数中的局域变量。还是用办公室来举例，局域变量就是有专门技能的雇员掌握的知识，而全局变量就是办公室里所有人都知道的知识。一个函数内部

的局域变量仅在该函数内有效。运行下段代码。“worker_bee”这个函数调用了一个全局变量并返回一个字符串。

```
#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
```

全局变量和局域变量的区别非常重要，如果你在一个函数外（或者其他函数内）访问局域变量的话，代码就会报错。运行下段代码。

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

fruitless()
print apple
```

“apple”这个变量在一个函数内被定义，它是属于该函数内的局域变量。当在函数外调用时，程序并不知道这个变量。如果我们想将这个局域变量定义为全局变量，可以在这个变量前作一个特别声明：

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

fruitless()
print apple
```

需要记住，尽管你可以在任何地方去调用一个全局变量，但是如果你对它的值做了任何修改的话，它的变化会体现在整个程序中。运行下段代码。注意尽管下面的函数可以调用全局变量“value”，但是在函数体内对它的值做改动并不影响全局变量。

```
value = 'hello'

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

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

但是，如果我们定义函数体内的变量为全局变量，它的值的改变则会影响之前定义的值：

```
value = 'hello'

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

change_variable()

print 'global ',value
```

提示：当一个函数被反复执行时，它体内的局域变量的值不会变化。因为每一次函数的调用都是独立的。

类

类是另外一个有用的程序语言结构。当一个类被定义后，你可以像函数一样调用类。当类被调用时，类的一个**实例**（**instance**）就被创建，我们称之为**对象**（**object**）。做个比喻，你可以想象类就是一个汽车工厂而对象就是车，车被这个工厂生产出来。一辆车被生产出来，它应该具有这个工厂给它装配的所有性能，但车和汽车工厂又是两码事。同时，这个工厂可以生产出很多车，每一辆车生产出来后它就成为一个独立的个体。

在下段代码中，我们首先定义一个类然后调用它两次。之后我们改变其中一个对象中的变量值，然后输出每个对象中的变量值，我们看他们的值有什么不同（通过*[对象名].[变量名]*来获取对象中的变量的值）。

```
#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
```

类内部也可以定义函数，我们称为**方法**（在类内部的函数称为方法）。一旦你创建了一个类的实例即对象，你可以调用对象的方法，调用对象的方法和调用对象的变量一样，即用[对象名].[方法名]。在下面的这段代码中，我们定义了一个有方法的类，然后我们创建了一个该类的对象。最后调用这个对象的方法。

```
#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()
```

一个特殊的类的方法就是**类的初始化**（initialization）。当一个类的对象被创建时这个方法就被立即执行。所以你可以用这个方法来为第一次创建的对象定义变量。当一个类被第一次调用时，类的这个方法接受任何传递来的参数。定义这个方法名为“def __init__ (self, [任何你想传递给该类的参数])：”来区分该方法和其他方法。

全局变量和局域变量的区别对编写类的过程很重要。在同一个类中调用变量和办法时，你需要用“self”作为前缀（“self”用来指该类中的一个特殊对象）。关于变量，如果你想要在类的其他地方调用一个变量，你需要在该变量前加上“self.”。在变量前加“self”就意味着该变量在这个类中都为全局变量。否则，该变量只在定义它中的方法中可见。

同样，我们用“self”作为类中所有方法的第一个变量。当你调用该方法时，用“self.”作为前缀。一旦你用一个类来创建对象，你可以把“self”替换为该对象来调用“self”变量和方法。

可能听起来依然很混乱，我们先从解释下面的例子来时。下面的例子中定义了一个类。类中的第一个方法就是初始化方法用来定义类的“moving”这个变量。在该变量前加上“self.”用来让该变量成为类中的全局变量，这样我们在代码后面的部分在“peel_out”这个方法中就可以调

用“moving”。在定义类后，我们用它来创建一个对象，然后调用对象的“peel_out”这个方法和“moving”这个变量。注意，当我们在类外调用“peel_out”这个方法时，我们并没有传递一个“self”参数。

```
#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
```

另外一个类的特征就是**继承**。一个类可以从其他类继承变量和方法，获得其他类所有的属性。一个类若要继承其他类可以在类名后面的括号中写入要继承类的名字也就是父类的名字。在下面这个例子中，我们定义了一个新的类“driver”，它要继承“make_car”这个类。然后在新类中用继承到的办法“peel_out。

```
#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
```

一个类可以继承很多类，但要清楚父类中有时会包含很多相同名字的办法或者变量。更多关于类的信息请参见 Python 教程的第九章。

模块

模块是一个包含所有定义的函数和变量的文件。模块可以被其他程序导入，以实用该模块中的函数功能等。例如，Python 中有一个名为“math”的模块用来提供主要的数学计算功能。当你导入 math，你的代码就可以**继承**所有在这个模块中的函数。试着运行下面的代码：

```
#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 )
```

我们用 Vizard 一类软件的原因是它提供一些有用的模块可以用来渲染虚拟世界。在这本教程中我们会在很多脚本中用到 **viz 模块**。这个模块包括一个用来创建虚拟世界的函数和类的库。下段代码用到 viz 模块用来打开一个图形窗口并在窗口中渲染出一个虚拟世界。这段代码中还用到模块的添加函数用来渲染一个三维模型并返回一个 **node3d 对象**。Node3d 对象提供一系列的方法，这些方法可以操作模型。这里我们用“setPosition”node3d 对象来设定我们添加的虚拟物体的位置。

```
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)
```

最快的方法来了解 Vizard 的所有模块内容就是去参阅 **Vizard 命令索引（Vizard Command Index）**（在 help 文档的下拉菜单中）。那里列出每个模块或对象的方法。

事件

交互功能对完善虚拟世界很重要。虚拟环境的图像输出随着用户在环境中的位置改变而不同并且环境中的不同元素要对应于应用户的输入。在程序中实现交互性具有一定的挑战性。在 Vizard 中处理这些交互性的工具为**回调（callbacks）**和**事件（events）**

回调就如一个守门人而事件就像是警铃。当你在程序中发起一个回调，也就是告诉程序注意一个特殊事件会发生。当这个事件发生时，守门人按响警铃并调用一个与该事件相关的函数。你可以为不同的事件触发不同的回调—不同的事件可以是计时器（“计算经过的时间”）或是键盘（“等待按键反应”）或是模型之间的碰撞（“等待模型 A 和模型 B 之间的碰撞”）。当给定的事件发生时，回调就立刻调用相应的函数并传递给函数相应的数据。

例如，你想实现一个功能即用户按“a”键时虚拟环境呈现。你可以先让这个虚拟环境不可见然后发起一个回调用来等待用户按键。当事件发生时，调用一个函数用以呈现虚拟环境。创建一个新的文件将下面这段代码加进去：

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

上面几行代码的功能就是先添加一个房间的模型并让这个模型不可见。现在我们添加一个回调用来等待按键，再添加一个函数。当事件发生时调用这个函数。下面这段代码中我们先写函数然后再写回调：

```
#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)
```

运行下面的代码并试着按“a”键。注意只有当按键后虚拟环境才呈现。在上面这段代码中，我们用了一个全局回调。你也可以从 vizact 这个模块中调用回调功能。下面这段代码实现的是相同的功能：

```
#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 )
```

下面我们用计时器事件来触发虚拟环境的呈现。当设定的时间过去后计时器事件取消。下面我们从 vizact 模块中来调用计时器事件功能。

```
#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 )
```

更多关于 Vizard 的事件功能请参见帮助文档中的“Event Basics”和“Event Reference”部分。

三维导航模式

在这本教程中我们主要用两种方法用在三维世界中导航。Vizard 的默认方式是用鼠标来实现导航。

打开名为“using actions example.py”这个文件（该文件包含在 Vizard Teacher in a Book Resources 中，请在 http://www.worldviz.com/download/index.php?id=9 下载），运行它。现在试着用鼠标来操控在虚拟环境中的视角和位置。把光标放在渲染窗口的正中间然后按住鼠标左键。稍微往前挪动光标就是向前走，鼠标移动的越靠前移动的速度就越快。稍微往后移动光标就是向后走，鼠标越远离屏幕中心则移动的速度越快。按住鼠标向左或向右运动即可旋转。按住鼠标右键后将光标向上下左右运动则视角可以相应的上下左右移动。同时按住鼠标左右键并点击屏幕上方或下方，则视角向上或向下旋转；点击屏幕左侧或右侧，则视角在屏幕所在面逆时针旋转或顺时针旋转（我们称视角向上或向下旋转为 pitch，视角在屏幕所在面顺时针旋转或顺时针旋转为 roll）。刚开始用鼠标来实现导航可能有些不顺手，需要多练习几次就会熟练。

另外一个导航的方式是以某点位置为中心（pivot）的旋转视角的导航方式。这种导航方式在本书的很多例子中都有应用。运行“transformations demo.py”尝试进行 pivot 导航方式。以某点为中心旋转视角（向左旋转或向右旋转），按住鼠标左键并向目标方向移动鼠标。向某方向移动来改变视角（向前、后或两侧），按住鼠标右键向目标方向移动鼠标。若要放大图像，按住鼠标右键并滚动鼠标滑轮。

实例

现在让我们看一个例子，这里面包含一些之前没有讲到的知识。但是不要紧，我们在后面的教程中还会涉及到。这里请读者体会一下一个完整的程序的逻辑结构。

在下面的例子中我们首先就是导入 viz 这个模块。这本教材中的例子的首句大部分都是导入 viz 模块，因为它包含了大部分的用于创建虚拟世界的函数和类。下一句代码，我们用到 viz 模块中的“go”这个函数用来打开一个新的图形窗口，在这个窗口中显示虚拟世界。

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

试着运行这小段代码，你会看到一个新的图形窗口。因为我们没有在里面添加任何虚拟物体，所以这个图形窗口是黑色的不会显示任何东西。

接下来，我们通过 viz 模块“add”一个气球的三维模型。我们将三维模型的名字和位置传递给“add”函数，这个三维模型“balloom.ive”从“art”文件夹里提取，“art”文件夹在“Teacher in a Book Demos and Examples”文件夹中。注意你需要将这个脚本保存到“Teacher in a Book Demos and Examples”这个文件夹下。因为在下面的语句中我们让程序直接在脚本所在的文件夹中查找相应文件。如果你将脚本保存到其他目录下，下面的这段代码运行时会报错。

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

调用“add”这个函数可以创建一个 **node3d 对象**。它属于 Vizard 对象的一种并且它有众多的办法可以用来操作三维世界。在这个例子中我们命名这个对象为 balloon，之后我们调用它时就可以直接调用它的名字。我们调用该对象的第一个方法时“setPosition”，用来设定三维对象的位置。把下面的这段代码加到前面的代码中然后运行：

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

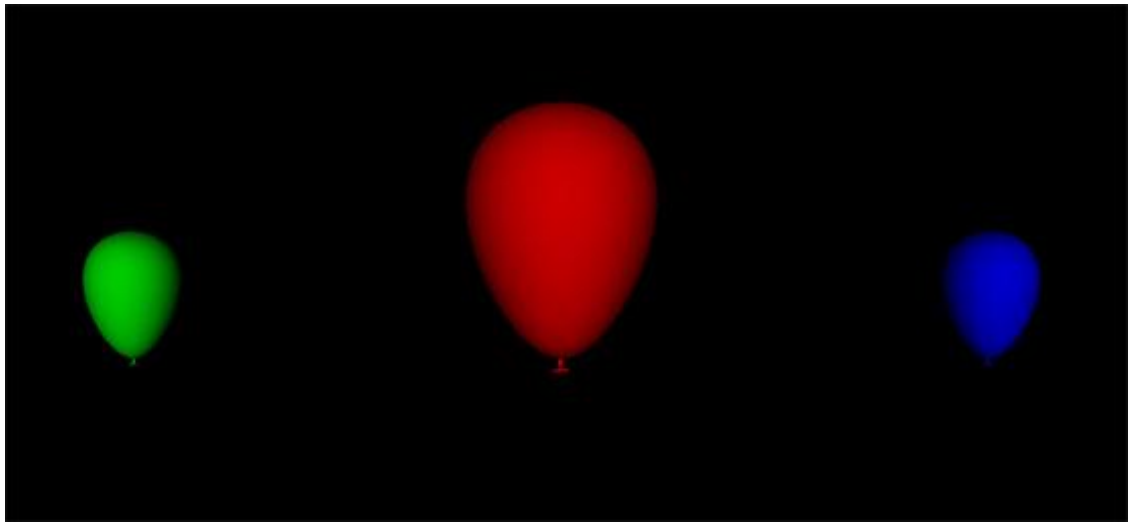
现在再添加两个气球。我们可以很简单的直接复制两次上段代码。但是如果用一个循环语句，可以使代码更有效。每循环一次，一个气球就被添加到环境中，随之这个气球的变量也被添加到“balloons”这个数组中。所以，这里我们用下段代码来代替上面的一句：

```
#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 )
```

注意我们添加的球呈现在屏幕的不同位置。这是因为在上段代码中设置球的位置时用到了“i”这个变量。而每个球被赋予不同的“i”值（-1，0，1 其中的一个值），所以他们的位置各不相同。

下面，我们赋予每个球不同颜色。通过调用“balloons”数组中球的索引来调用每一个球。我们会用到“color”这个 node3d 对象的方法，并传递给这个方法不同的变量来设置颜色。这里的变量是**标记**的形式。标记就是 Vizard 中的标准常量的名字。Vizard 中基本的颜色都有标记。

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



最后，我们定义一个让气球膨胀的函数。这个函数接受一个气球的对象（在函数内部被定义为“who”）。在这个函数内部创建一个**动作**，我们在下面的建模部分详细讲到。在创建这个动作之后，我们会将这个动作应用到传递过来的气球变量。

```
#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 )
```

现在我们已经创建了这个函数，下面调用这段函数并将其应用到中间的这个气球。

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

运行上段的例子看看结果是怎样的，然后将下面的一段加到程序里。下面这段创建了一个键盘事件的回调，当你按键盘上的‘b’键时其中的一个气球就会膨胀起来。

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

练习

1. 修改上面的代码，在虚拟环境中添加 6 个气球。

注意：你可以通过鼠标改变视角和导航来观看所有的气球。

2. 添加一句代码，功能是当气球膨胀时(可用上段气球膨胀的函数)气球的颜色变为紫色（可用标记 viz.PURPLE）

Vizard 提示

当用 Vizard 编写程序时，我们会经常用到和查找 Vizard 帮助文档的内容。Vizard 所有的命令索引（在 Vizard 面板中的帮助文档）都可以在文档里找到。这些命令按照不同的功能分组，例如属于变换视角或光或虚拟人骨骼的各为一组。这些索引的条目都会列出它需要哪类参数和返回什么值。

对于更多内容，请查找 Vizard 帮助文档。你在帮助文档中会找到一个名为“Reference”的部分。这部分按照不同主题分类，每一个主题都有一个基本内容页面来帮助你开始熟悉这部分。每个主题下还包括一些相关内容的连接。帮助文档中还包括一个非常有用的具体例子的部分。

建模

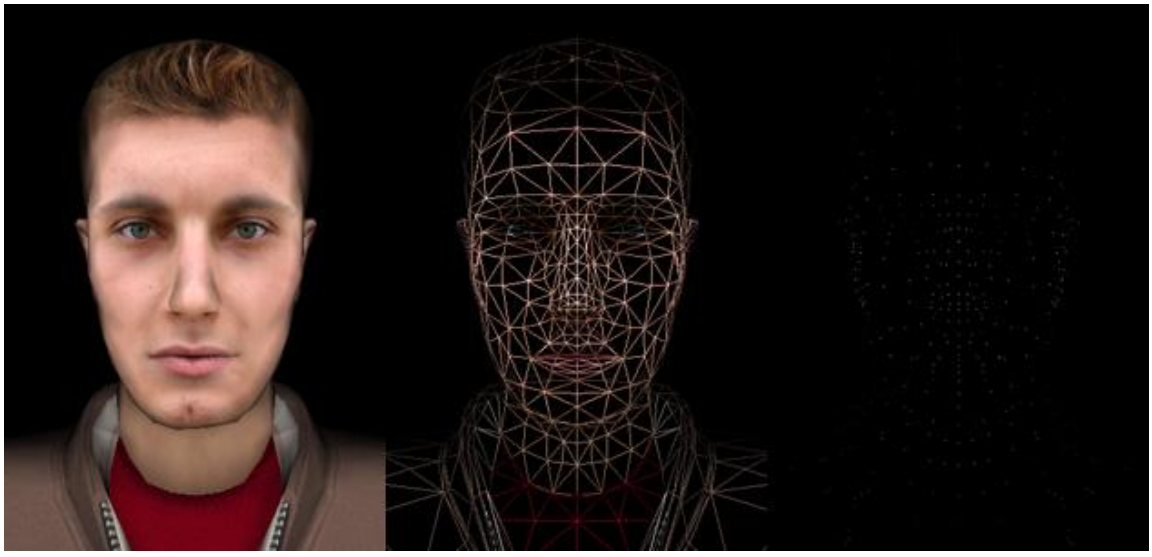
几何建模

虚拟的空间充满着模拟现实世界物体的三维虚拟物体，从太阳系到人体。这一章节中我们先快速阅览一下这些虚拟物体的组成部分。

点、线和多边形

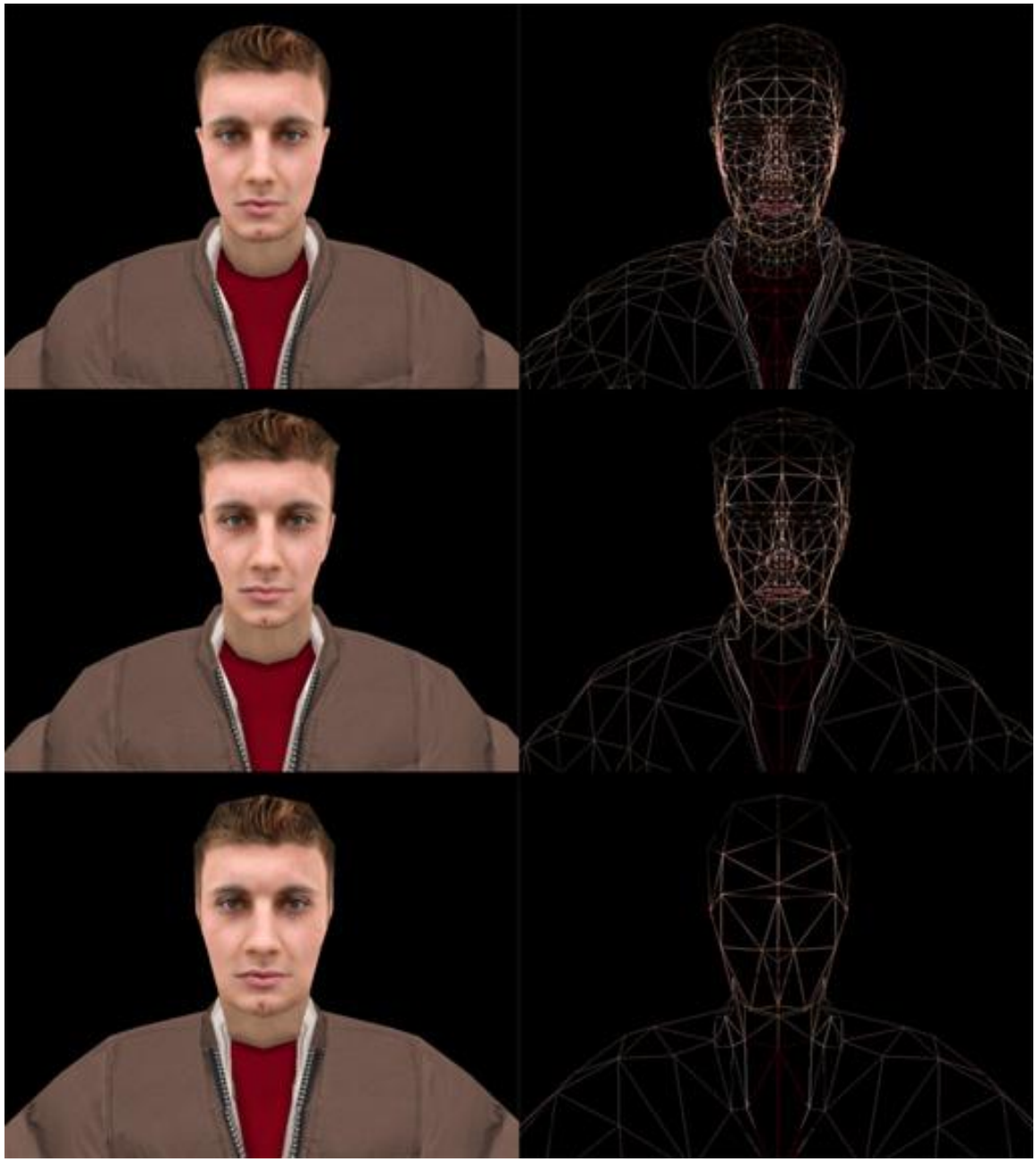
你可以将虚拟物体想象为一个三维平面的集合。尽管他们看起来像是实心的固体，实际上更准确来讲这些虚拟物体更像是被带有材质的线架结构包裹的中空物体。这些三维平面由一系列点和线构成。线又用来定义各种图形，这里我们叫做**多边形**（**polygons**）。组成一个模型的多边形拼在一起形成一个物体的面。运用到这些面上的**材质**（**texture**）又予以物体一个更逼真的外表。

试着运行 geometric modeling demo，你可以看到一个模型的不同部分。这个例子运行后，你会看到三个带有材质的模型。点击 F3 你可以看到组成这个模型的多边形的网络结构。再点击 F3 你会看到组成网络结构的点。



可以想象一个模型有越多的多边形就会有越多的面。再次运行这个例子，注意从右到左的三个模型，组成他们的多边形依次增多，这也会使模型的面增多从而使模型看起来更加生动。所以为了使模型更加生动也许你会在建模时创建更多的多边形结构。然而，程序处理这些复杂模型是有代价的。程序在渲染多边形结构的相应帧时会消耗系统资源。如果一个模型有过多的多边形结构就会造成非常慢的**帧频**（每秒渲染的帧数）。如果我们创建的环境包含时时的交互功能，就需要非常快的帧频。所以我们要平衡多边形数量和处理三维模型所消耗的系统资源。当然这里也有一些可以解决这些问题的技巧，例如提高模型贴图和材质的质量使模型看起来更生动，升级显卡，

以便处理更复杂的图形渲染。现在显卡的性能提升的很快，高性能的显卡可以处理包含大量多边形的虚拟场景。



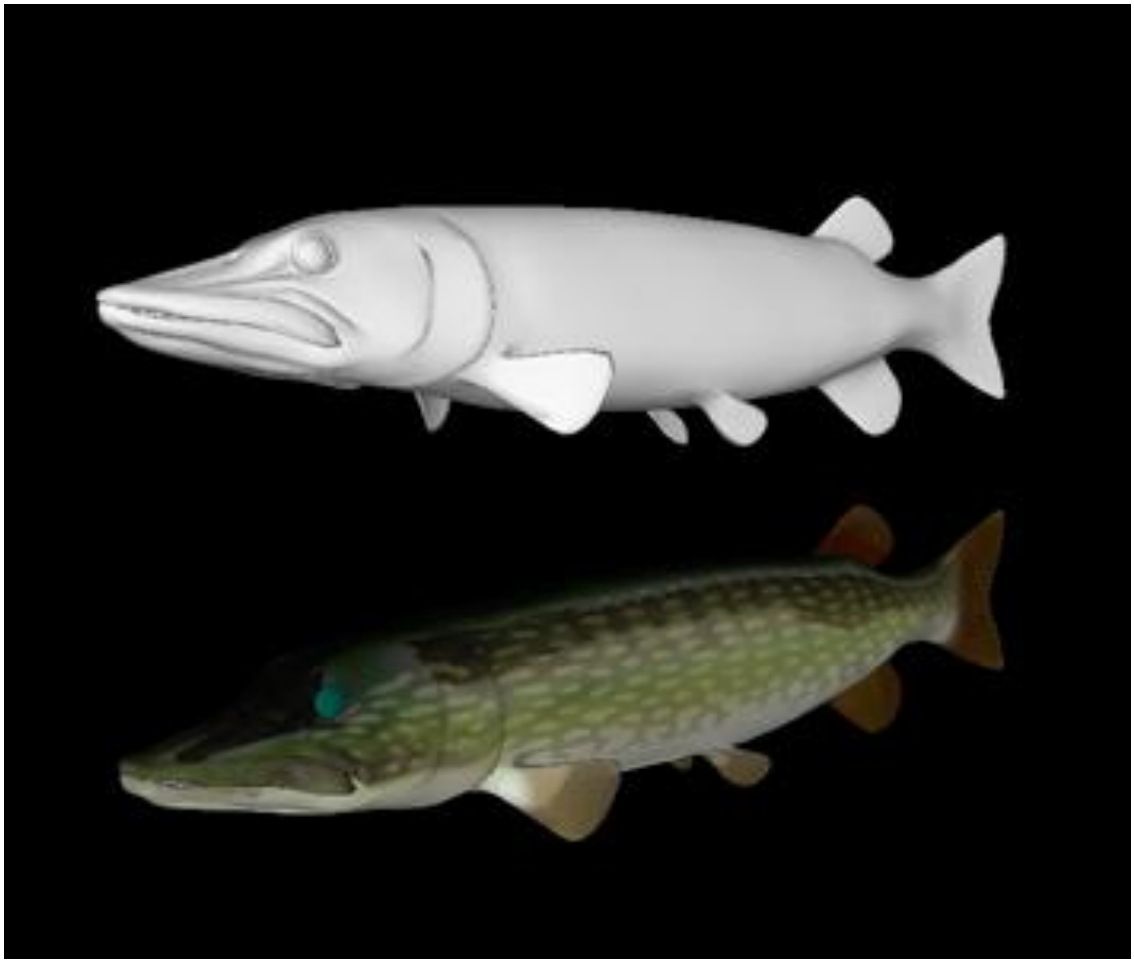
大多时候我们会用一些专门的建模软件来创建三维环境（3DS Max, Maya 等等）。然后导入这些模型来用于三维呈现。提示：你可以创建一个“on the fly”模式的模型。当一个模型在你

的程序中要经常变换形状时（例如一个钓鱼线，钓鱼线被甩出或者收回的过程中形状一直在变化），创建此类模型可以简化建模过程。更多信息请参阅 Vizard 帮助文档。

材质贴图

我们在几何建模的部分已经提到了三维模型实际上就是由一系列的点线面组合起来的几何体。如果你要构建一个砖块的模型，并想展示出砖块表面的不同颜色和凹凸不平的表面，你可能需要大量的几何体来组成这个砖块。这个构建模型的过程应该是极为繁琐，并且也不利于计算机的快速渲染，如果你还要添加一些时时的交互功能的话会将渲染过程拖的更慢。

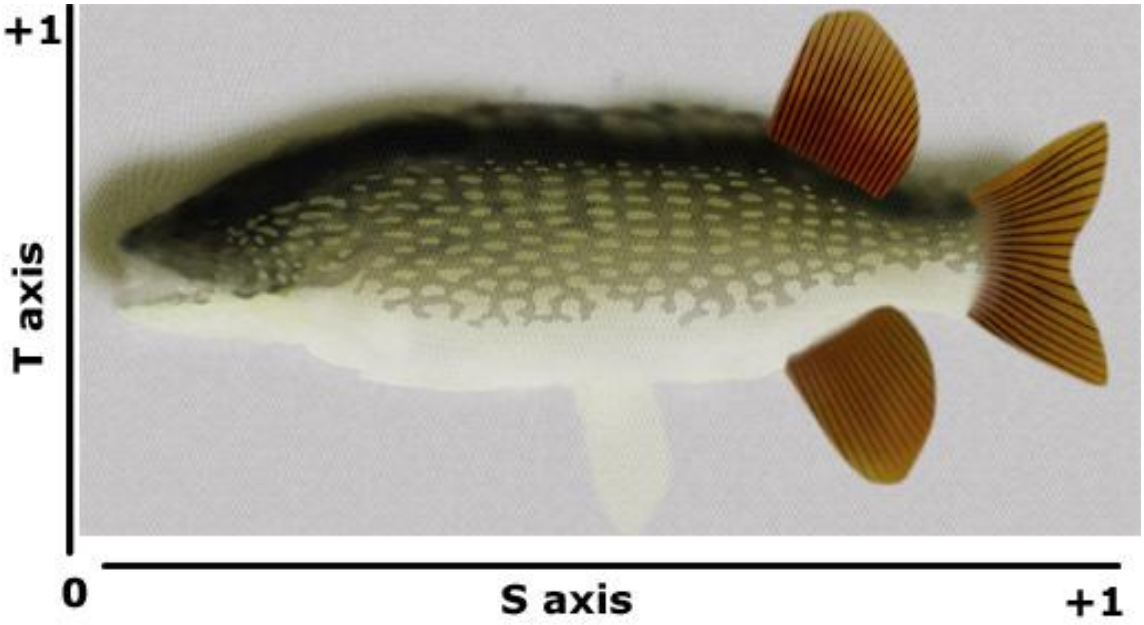
为了解决这一问题，我们用到材质贴图。材质就是一些“包在”三维模型表面的图片，我们还可以对这些图片进行颜色和图案的修改。将材质贴图在模型表面会给模型带来非常丰富的细节但不会增加冗余的几何结构。材质也可以用于在虚拟世界中呈现二维媒体，例如图画和视频。



材质的坐标系

材质一般都是二维的图片文件（例如 jpeg 和 tif 文件）。这些二维的图片贴到三维物体上时就需要一个表面去包裹（在这部分的例子中，我们会将材质贴到一个很简单的三维平面上。）

为了知道二维文件是如何包裹到三维表面的，我们用到**材质的坐标系**。一个三维模型表面的材质坐标系定义了这个材质如何的覆盖到这个表面上。材质坐标系有两个轴：S 轴代表材质图片的宽边，T 轴代表高边。无论材质或者模型的大小是多少，材质坐标系的两个轴值的范围都在 0 到 1 之间。 对于这两个轴，（0, 0）都代表左下角的原点。



一个三维模型的表面上每个点都被赋予一个 ST 坐标值。这样当材质贴到模型表面的时候，材质上对应的该坐标点的颜色就会被赋予到模型上对应的坐标点上。例如，我们有一个头部的三维模型，鼻尖上的一点的 ST 坐标值为（.5, .5）。当我们要给这个模型贴图时，材质贴图的 ST 点的颜色就会赋予给鼻尖上对应的这个坐标点。

为了将材质贴到模型上，需要 ST 坐标系来标记模型。建模软件（例如，3DSMax）一般会用到 S-T 或者 U-V 纹理映射来对几何体进行材质包裹。

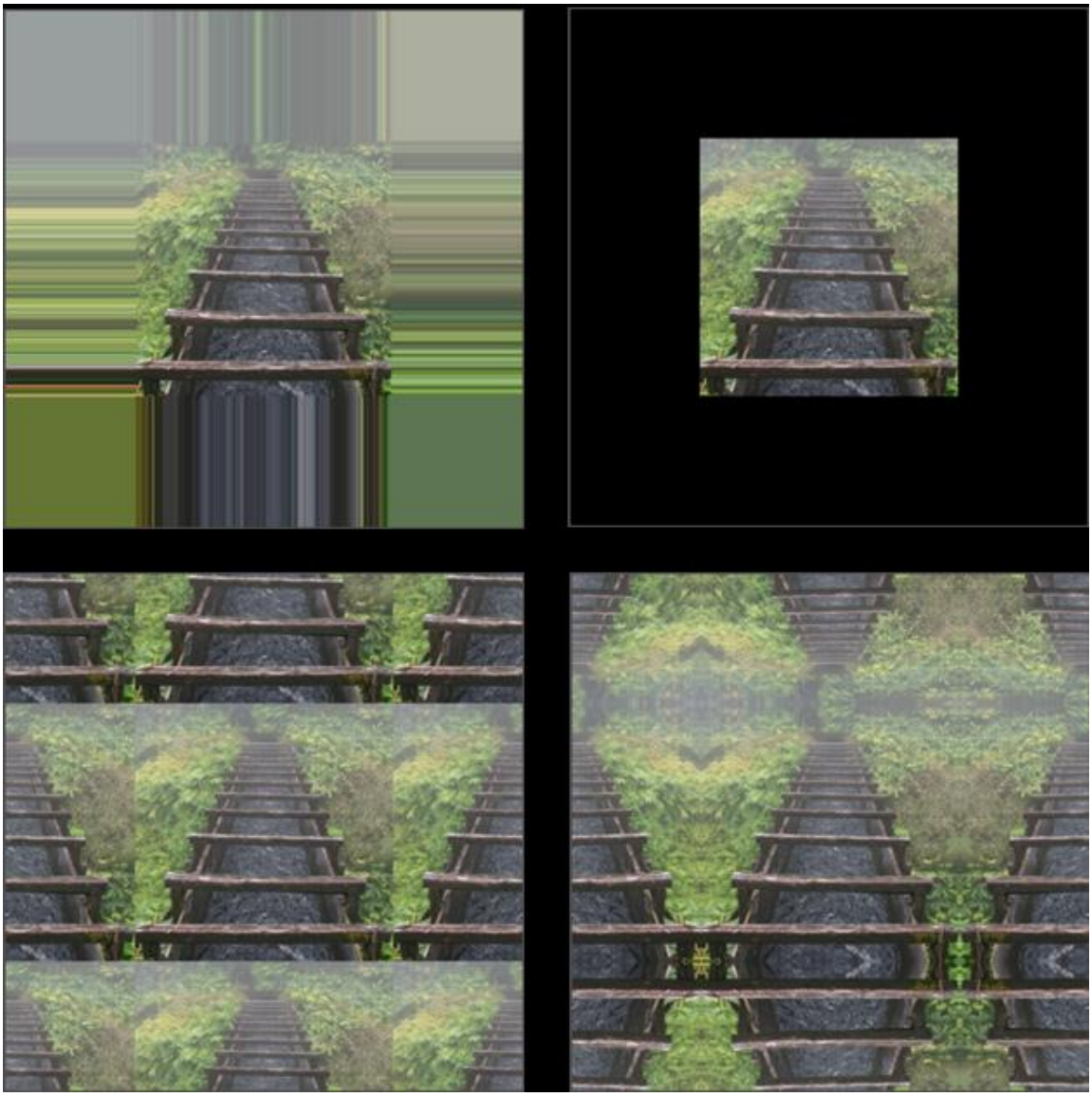
运行‘texture coordinates demo’看看材质坐标系是如何工作的。在这个程序里面，材质被赋予给一个四边形（一个简单的三维平面模型）。当该程序启动时，一个材质坐标系就被赋予给这个材质贴图从而使这个材质贴图可以很好的覆盖在这个四边形上。贴图图片的 0, 0 点对应四边形的坐下角，1, 1 点对应四边形的右上角，以此类推。在这个例子中，你也可以通过改变贴图坐标系的大小和位置来变换贴图的位置。在”texture coordinates”菜单下将 ST 坐标系的数值范围（scale）改为 2。改变后坐标系在 ST 两个维度上都被拉伸，所以我们可以看到贴图只会覆

盖到一半。（这个效果看起来像是把贴图图片缩小了一半，但是我们改变的是材质表面的坐标系的数值范围，而不是材质图片本身的大小。所以原来的（.5, .5）坐标点在改变后的坐标系里对应的是 （1,1）点）

按照同样的原理，我们可以通过移动材质贴图的坐标系来改变贴图位置。试着将 ST 坐标轴移动到-0.5 的位置，贴图会随之向右和向上移动。（在这个过程中，贴图表面的（1,1）点在移动后变为（.5, .5）点）。

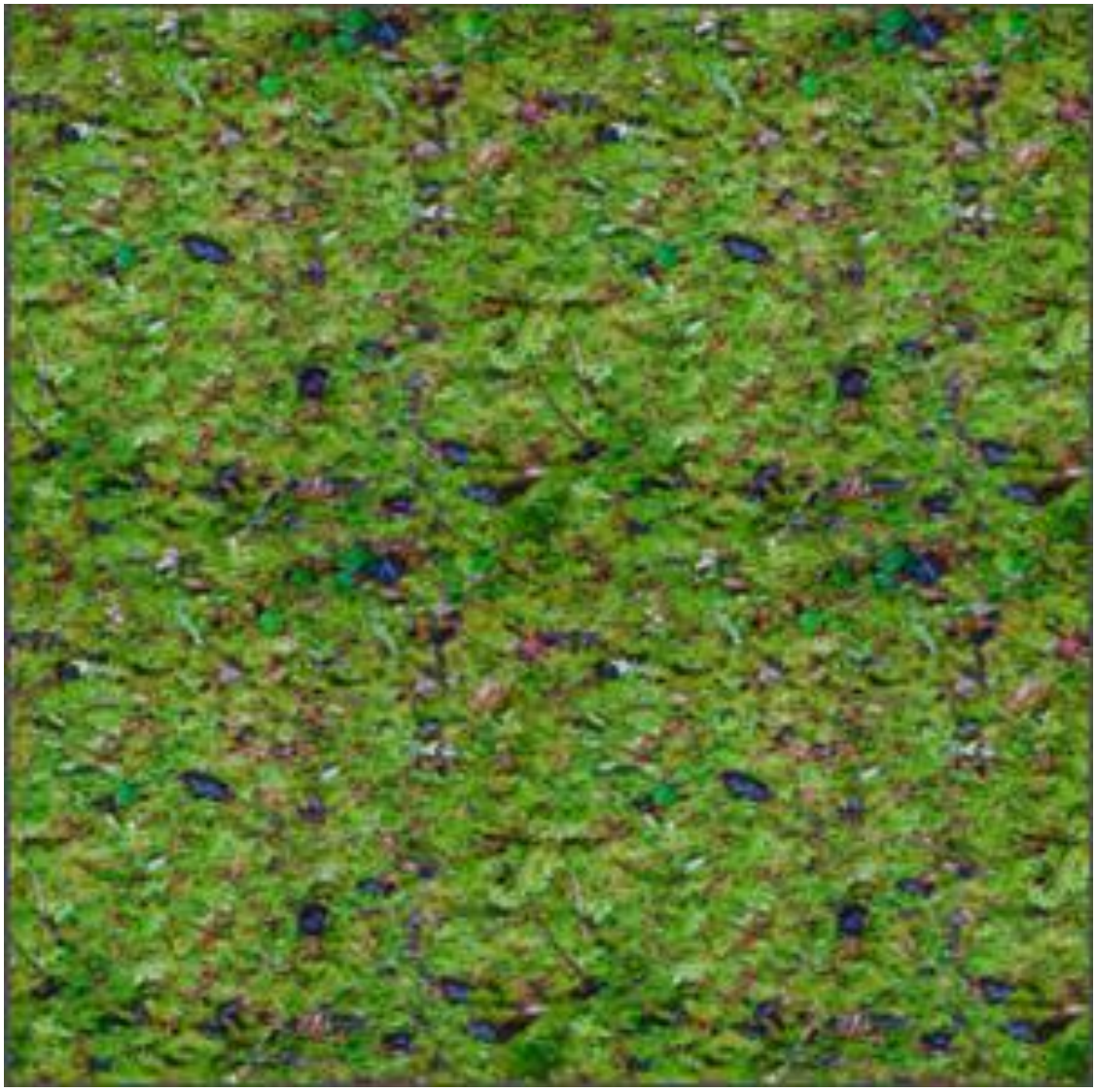
在这个例子中，贴图坐标可以通过某种方式的变换而无需覆盖整个三维表面。**贴图包裹模式**（texture wrap modes）定义了一个没有被完全覆盖的三维物体的剩余部分（也就是贴图坐标系在 0 到 1 之外的范围）应该如何被贴图。一个包裹的方法就是在物体的剩余表面重复相同的贴图。另外一个方法就是镜像，实际上镜像也是在重复已有的贴图。如果你不想重复贴图， 你可以**夹住**材质贴图然后用该贴图最边缘的颜色或者是像素，赋予给三维物体的剩余的没有被贴图的部分。

运行 texture coordinates demo 程序。变换材质坐标系的数值范围让材质不会完全覆盖在模型表面，然后在“wrap modes”下变换选项的数值。夹住贴图边缘，在“border color”菜单下试着变换边缘的颜色。



当贴图模式变为重复贴图时，贴图是否可以**无缝连接**就变得很重要。当一个贴图足够简单时，把多个贴图拷贝连接到一起，它们之间连接的缝隙很难看出来。

重新运行 texture coordinates demo，并变化材质贴图坐标的数值范围让该贴图不能完全覆盖模型表面。设置包裹模式（wrap mode）为“REPEAT”，到“blend”菜单下并移动滑动条到最右侧。注意现在已经非常难辨别重复贴图之间的缝隙了。这样的贴图方式对于大型模型非常有用，例如地面模型。



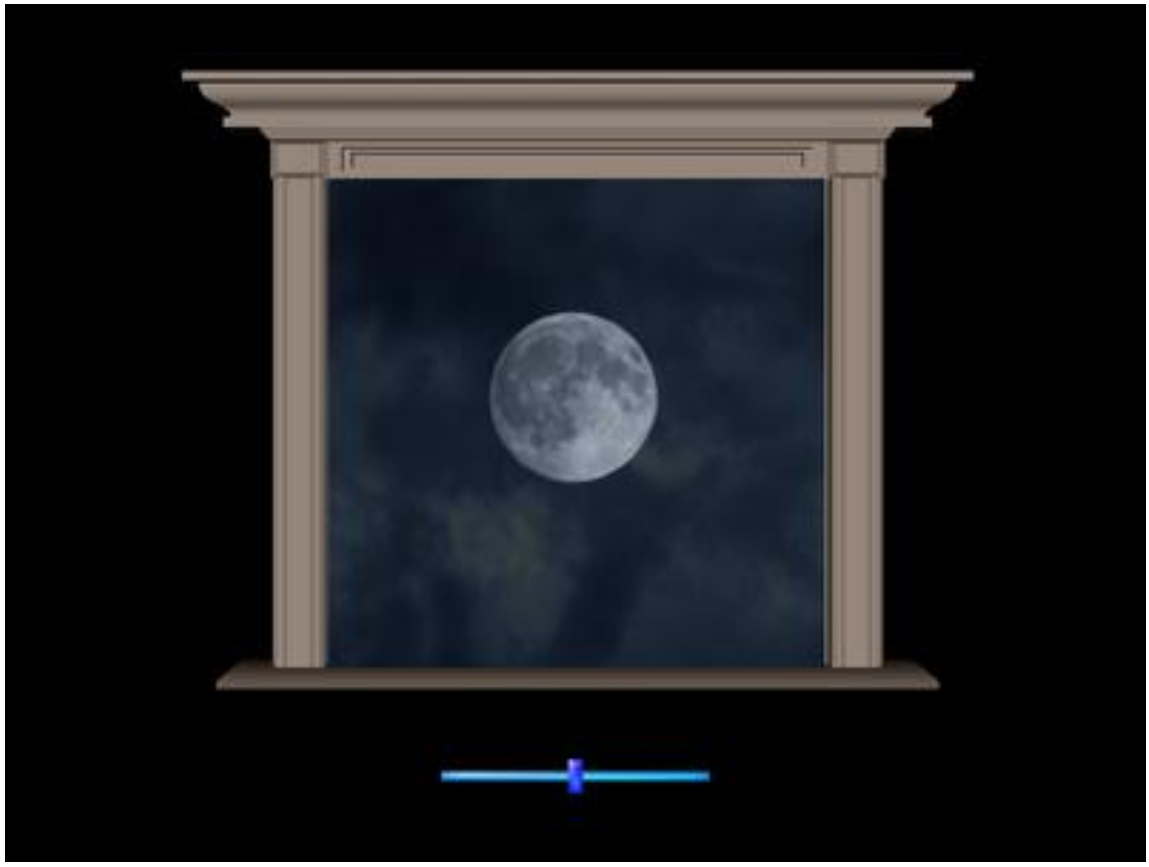
多重贴图

多重贴图（multitexturing）的方法可以赋予模型表面多个贴图。多重贴图结合光可以使模型看起来更真实和复杂。你也可以将多个贴图混合起来模拟变化的表面（例如颜色变化的叶子）。3D 建模软件有很多类似的选项可以用来多重贴图。（例如，凹凸贴图 bump-maps 和透明贴图 alpha maps 等等）。

在 texture coordinates demo 中，“blend”菜单用到的就是多重贴图。当你移动滚动条，使最原始的贴图不断重复时，实际上你在改变每个贴图映射到模型上的程度。

练习：运用贴图

在这个例子中，我们会用到一个模型的一小部分并将一些贴图覆盖到它上面，接下来会对材质坐标系做一些调整。打开一个新的脚本，我们会一步一步的编写整个程序，然后检查程序运行的结果。



首先，我们导入 viz 库然后用 go 命令渲染一个场景。之后我们设置主要观察视角的位置从而能看到渲染的三维创景。接下来在这个场景中添加一个模型并抓取它的子元素即 **sub-node** 名字为“glass”。这些命令都会返回一个我们可以控制的 node3d 对象（我们之前会将它们标记为“model”和“window”）

```
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' )
```

接下来我们用 viz 库中的“addTexture”命令添加材质贴图文件。这两个文件为 jpeg 格式，存储在 art 文件夹里。

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

现在我们用“texture”命令将这些材质添加到视窗内。这个命令可以允许你将已有的材质覆盖到模型上，同时也允许你定义材质可以覆盖的范围，例如命名为 1 这个单元区。定义覆盖范围对于材质贴图非常重要。

```
#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 )
```

我们用 wrap 命令来定义材质包裹的模式。这个命令接受两个参数：一个为 ST 坐标参数，另外一个为贴图模式。这里我们用重复贴图模式。

```
#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 )
```

接下来我们在屏幕中添加一个滑动条这样使用者可以随时调整贴图效果。具体来讲，使用者可以左右移动这个滑动条用来控制两个材质的混合程度。首先我们添加一个滑动条对象（它是一个 node3d GUI 对象）然后设置它在屏幕中的位置。然后我们添加一个函数，这样当使用者移动滑动条时函数就被调用。再用 onslider 函数设置一个移动事件。当移动事件发生时，它就会调用 swap_textures 这个函数并告诉该函数滑动条被移动的位置。Swap_textures 命令会接受传递来的滑动条的位置数据，然后将材质按照相应的数据贴到物体表面，这个程度是从 0（不覆盖）到 1（物体完全被材质覆盖）。

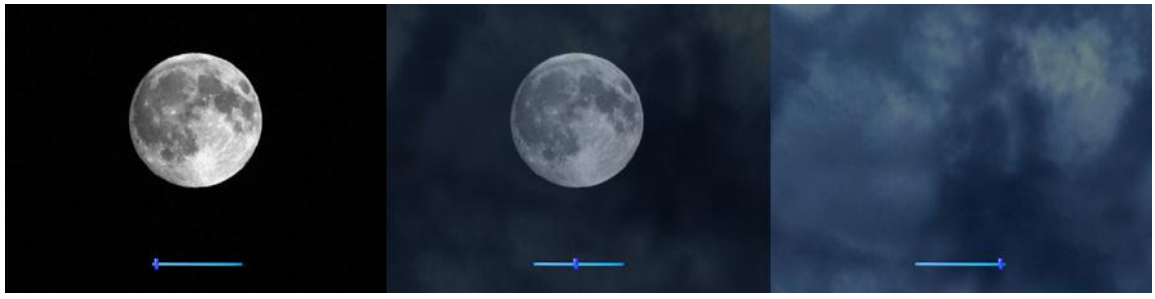
```
#Add a slider and put it on
#the bottom of the screen.
slider = viz.addSlider()
slider.setPosition(.5,.1)
#This function will be called
#every time the slider
```

```
#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)
```

运行这部分的程序看看结果如何。试着移动屏幕中的滑动条，调整到你可以看到整个模型为止。

下面我们通过调整窗口的贴图坐标系来移动 clouds 这个贴图。首先我们用 vizmat 这个函数库中的命令来创建一个变换矩阵。这个矩阵包括所有材质坐标系的范围以及位置信息。之后我们用一个计时器来调用 move_clouds 这个命令（计时器功能在建模部分有详细解释）。Move_clouds 这个函数中包含一个“postTrans”命令用来移动变换矩阵（也就是移动材质坐标系），然后用 texmat 命令将材质贴图依据变换矩阵的值贴到物体上。

```
#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 )
```



重新运行以上的代码，当你向右移动滑动条时你会看到 clouds 这个贴图随之移动。

练习

1. 将“art/wall.ive”这个模型添加到一个场景中，将“art/stone wall.jpg”材质贴图贴到这个模型上的一个单元区。

2. 改变材质坐标系的数值范围，然后将材质重复的贴到物体上。

3. 将另外一个材质“art/pine needles.jpg”贴到模型的另外一个单元区并将两个材质融合成下图这个效果。

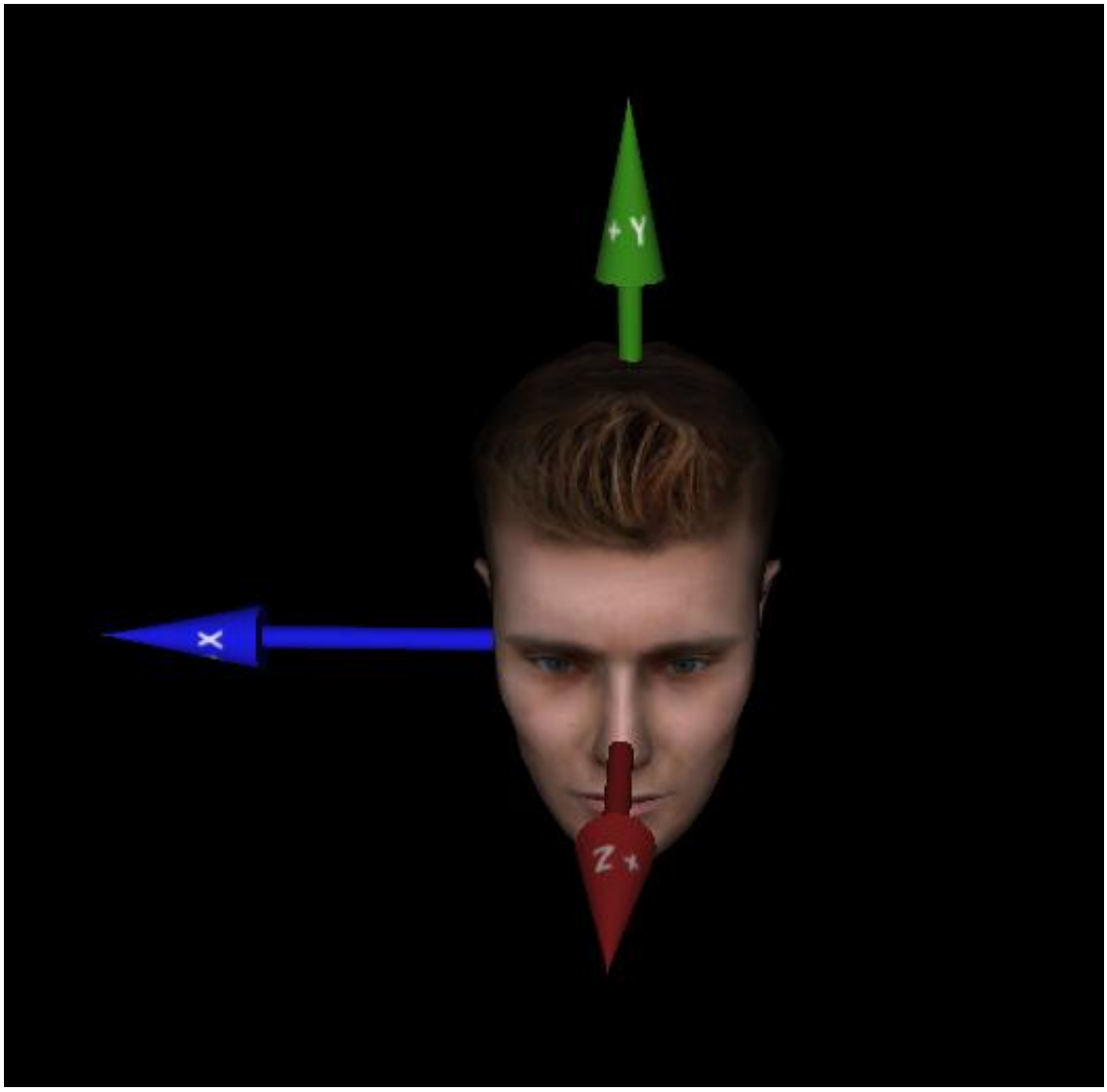
场景图

场景图定义了一个虚拟世界的基本结构。场景图是一系列由虚拟场景中的物体组成的**节点 nodes** 构成的。在场景图中的节点可以通过 **3D 转换**来改变它的特征。这些转换包括位置、朝向和大小。在这一章我们会详细介绍利用 3D 转换来改变物体和场景图的特征的基础知识。

转换

当你在一个三维空间中工作时，对三维结构的了解是非常重要的。如果你已经清楚什么是三维结构，你可以略过这部分。如果你还不清楚，请认真阅读此部分。

三维世界是由[x, y, z]坐标系定义的。x，y, z 轴分别从**原点**向外延伸。在 Vizard 的参照系统中，x 和 z 轴定义在水平面上而 y 轴定义在垂直面上。如果你将你的头部放在坐标系的原点，那么 z 轴的正方向应该从你的鼻尖部位向外延伸，x 轴的正方向应该从你的右耳部位向外延伸，y 轴的正方向从你的头顶向外延伸。（注意：不是所有的三维建模软件都用同样的定义方式，有一些软件定义 x 和 y 轴在水平面上而 z 轴在垂直面上）。



我们用[x,y,z]坐标系来定义位置、朝向和大小的转换。如果你想向上移动一个物体一米，你可以定义它的位置在[0, 1, 0]。如果你再想继续向右移动该物体一米，你可以定义它的位置在[1, 1, 0]上。大小的定义也可以通过改变坐标系的数值来完成，例如将一个物体的大小在特定的轴上扩大。将一个物体的高扩大两倍，我们可以用[1, 2, 1]来表示它的扩大倍数。

旋转的变化也可以通过[x, y, z]的值来表示。想象你的头部放在坐标系原点，绕着 x 轴的头部旋转也就是逆时针或顺时针摇晃你的头我们称为 **pitch**。绕着 z 轴的头部旋转类似于点头的旋转，我们成为 **roll**。绕着 y 轴的头部旋转类似于摇头的旋转，我们成为 **yaw**。这里有几种数学方法来定义这几种旋转。一种方法就是通过坐标轴的方式来定义在哪个轴上（x, y, z）旋转了多少度。另外一种方法就是用欧拉方法 **eulers**。用 eulers 方法表示 yaw、pitch 和 roll 的值，记住这里面有一个特定的顺序[yaw, pitch, roll]。第三种方法为四元法 **quaternions**。四元法

quaternions 就是拥有四个维度的一个向量。可以认为它是一个在四维球体中的半径向量，它指向该球体中的一个点。

这些节点位置、大小和朝向的信息包括在一个矩阵中，我们成为**转换矩阵**。当你转换一个节点时，你实际上变换的是该节点的转换矩阵。但通常在程序中你不会直接操作这些转换矩阵，但是知道转换矩阵的概念和背后的原理是很重要的。

运行 transformations demo，看看运行的结果是什么。再试着点击运行界面中的功能按键看看有什么效果产生。

场景图和坐标系

在场景图的层级结构中所有虚拟物体都是其中的节点。场景图有一个**场景根（scene root）**，所有的节点都从场景根中分支出来。每个分支在场景图中又可有多个节点。当你顺着一个分支找下去，在分支的一个节点下面的节点称为**子节点（child）**，上面的节点称为**父节点（parent）**。如果一个节点没有其他父节点，那么场景根就是它的父节点。

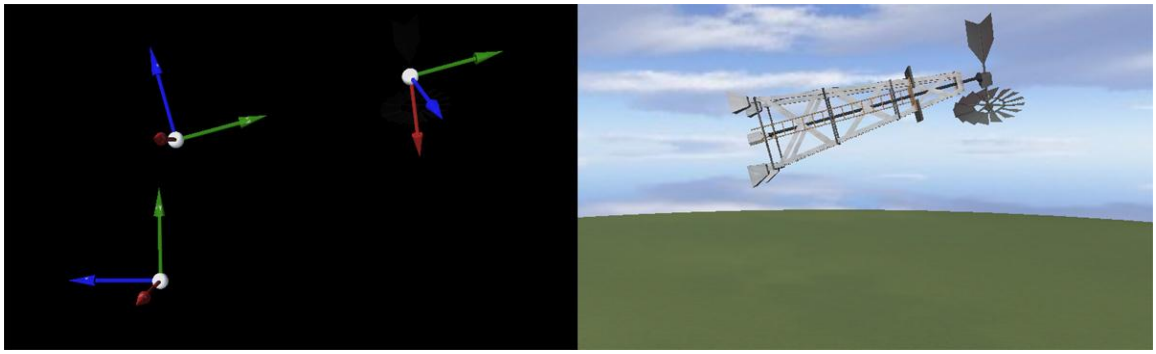
每个在场景图中的节点都有自己**本地的坐标系统（local coordinate system）**。这个节点的父节点也有坐标系称为**父坐标系（parent coordinate system）**。根节点的坐标系成为**全局坐标系（global coordinate system）**。如果一个节点直接类属于场景根，那么这个节点的父坐标系也就是全局坐标系。

对节点进行转换时有两个关键的概念需要掌握：1）转换是针对一个指定的参照系统来说的 2）节点继承来自于对父节点的转换。例如，在父坐标系下移动一个节点到[0, 1 ,0]的位置，之后在全局坐标系下移动它的父节点到[0 ,2, 0]的位置。因为该节点继承来自对父节点的一切转换，所以现在这个节点的位置应该在[0, 3, 0]的位置，尽管你只在第一步移动了子节点。这个例子可能还是有些抽象，那么想象你的父母开车从北京向北走了 100 公里（全局坐标系），你为了远离你的父母又向北走了 100 公里（父坐标系）。现在你在离北京 200 公里的地方（全局坐标系）。

考虑到转换矩阵，节点有自己本地的转换矩阵通过结合父坐标系的转换矩阵来变换本地坐标系统的位置、朝向以及大小。

注意：节点的坐标系随着节点运动。一个节点永远在它自己坐标系的原点[0, 0, 0] 位置。你可以在另一个坐标系中移动节点，但一旦节点移动，坐标系也会随之移动到新的位置或方向上。

另外关于转换过程，转换可以**相对（relative）**于物体现在的位置、朝向或大小也可以是**绝对（absolute）**的转换。例如，你就是一个节点，我把你绕着全局坐标系的 y 轴旋转 10 度。如果这个旋转是绝对旋转那么你会绕着这个坐标系的 y 轴旋转 10 度。如果我对你还做同样的转换也就是绝对转换，这时你的朝向就不会有任何变化。因为之前你已经在全局坐标系下转了 10 度了。如果我对你做的旋转是相对于你现在的位置的，你会在原来的基础上做相应的旋转。



为了对不同性质的转换有一个更清楚的认识，请运行 coordinate systems demo。在这个例子中，父节点是一座塔，子节点是一个在塔顶的涡轮。在屏幕上端的菜单中选择你想要转换的物体和参照系，并且选择你想要的转换时相对的还是绝对的，旋转是绕着哪一个轴。你可以选择“show coordinate systems”用来更清楚的看到坐标系。当你都选择好后，点击屏幕下方转换的按钮（“rotate”，“translate”，“scale”）。

实例：转换和场景图

现在我们具体的看一个例子。在例子中我们会添加一些模型到场景图中然后对他们进行转换。

打开一个 Vizard 脚本，使用 viz 库中的“add”语句来添加模型。这个语句返回一个“node3D”对象，我们在之后可以用这个对象来对这个模型进行一些操作。

```
#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')
```

接下来，我们会将场景背景设为蓝色然后将视角放在一个便利的位置。

```
#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)
```

现在运行这个脚本，你会看到一堆物体都位于全局坐标系的原点。接下来我们将这些物体摆放在合适的位置上。第一步先将鱼这个物体设置为鱼竿的子节点，然后设置鱼在鱼竿坐标系下的位置。属于 node3d 的命令“parent”可以将 node3d 对象（这里是鱼）放在特定节点（鱼竿）下。为了让鱼位于鱼竿的正确位置上，我们在鱼竿的坐标系（父坐标系）中设置它的位置和朝

向，这里会用到 node3d setPosition 和 setEuler 这个命令。之前提到过，eulers 会用到特殊的顺序[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 )
```

现在我们在鱼自己的坐标系下调整鱼的大小让它看起来大些。

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

接下来，我们在全局坐标系下移动鱼竿，使它倾斜到桶的外面。之后我们设置虚拟人的位置。我们在设置虚拟人位置的命令中并没有设置移动的性质。因为在父坐标系下的绝对移动 viz.ABS_PARENT 是默认值，这正是我们想要的。

```
#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)
```

现在在这个感觉像郊外的小场景中添加一些乐趣。首先我们添加一些蜜蜂，设置蜜蜂为虚拟人的子节点。然后设置蜜蜂的位置差不多在人头部的附近。

```
#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)
```

现在场景看起来还是有些单调，我们可以让这些蜜蜂绕着人飞。这里会用到计时器，计时器每 0.01 秒就会触发蜜蜂绕着一个轴旋转(yaw)5 度。我们用 vizact 库中的“ontimer”命令启动计时器功能。每 0.01 秒就调用 swarm 函数。

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

现在我们让虚拟人对这些飞舞的蜜蜂做出反应。首先，我们让虚拟人处于一个奔跑的状态（让虚拟人处于第十一种动作模拟状态也就是跑步的状态）。之后我们调用 Python math 和 time 库中的命令用一些数学的公式来计算虚拟人运动的路径，让他绕着一个圈跑但始终面朝前方。最后我们再次调用 ontimer 功能。

```
#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 )
```

连接

连接是建立对象之间的一种关系的动作但是这种关系又区别于父-子类关系。连接有一个**源**和**目标**，还有应用到目标的源的转换矩阵。例如一个源有一个旋转上升的动作，那么目标也会有同样的动作。很多东西都可以作为源包括 3d 节点、追踪感应器、视点而目标经常是 3d 节点和视点。连接非常有用因为它可以让你将同样的转换应用和控制到任何节点和视点。例如，可以将追踪设备的 3d 追踪数据连接到视点从而在三维世界中导航。也可以用连接功能模拟手抓取物体的过程，当手伸出去碰到模型的时候，模型可以依附在手上然后跟着手一起运动。

一旦你将一个源连接到了目标，你可以操作连接本身，而不用使用所有链接的数据执行转换。例如,你可以设置一个偏移量使一个目标的位置永远远离源[x, y, z]远。你也可以对连接运用运算符来移动连接数据。

为了进一步了解连接是如何工作的，打开 link demo。在这个例子中，手推车和足球是连接的。手推车是源，而足球是目标。首先到菜单中选择手推车旋转和移动的值。连接默认的动作是对目标朝向和位置的变动，而不是大小。注意对目标（球）的转换变化是没有任何效果的因为转换矩阵是被源的矩阵所定义的。选择连接所用到的数据，你可以选择连接的数据选项（在“link”菜单下）。如果你选择“viz.LINK_POS”选项，那么只有位置数据会应用到连接中。所以目标的位置可以被单独操作。

另外，在“link”菜单下的选项中还可以选择偏移量或者是运算符。每个选项都需要一个三个元素的数组（例如， [1, 0, 0]）才能实现功能。一般来讲，前运算符是在源坐标系中对目标进行操作，而后运算符是在目标的父坐标系中对目标进行操作。当你重设连接时，你会去掉连接的所有运算符。

在这个例子中，你可以试着让球出现在手推车的里面，当你移动或旋转手推车时球不动。

实例：连接

现在我们建立一个新的脚本文件来具体实现一个连接。首先，我们添加一些模型，设置他们的位置和朝向。然后我们添加一个虚拟人并赋予他一些动作。下面我们会详细介绍如何实现。

```
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)
```

下面我们将一个男性虚拟人的骨骼连接到球上。

```
#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 )
```

如果你现在就运行上段脚本，你会发现虚拟人的手正好穿过这个球。我们可以对这个连接设置一个偏移量好让这个球（连接的目标）在正确的位置上，这个偏移量应该是在 y 轴偏移。我们还要对连接设置一个数据选项，这样的话我们只对位置数据进行操作并且球可以旋转了（在我们连接球之前它已经是在旋转的状态了）。

```
#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 )
```

现在我们将女性虚拟人和桶连接在一起，让她看起来好像是在桶上面走。如果你试着只运行下面代码的第一行，这个场景看起来有些奇怪。所以我们要设置连接数据选项和一个虚拟人对桶的偏移量。

```
#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] )
```

运行上面的代码看看效果如何。接下来我们会利用一些外部数据来移动桶的位置。我们简单的就用鼠标的位置数据。“swapPos”命令可以调换连接数据，在这里鼠标的 x 和 y 值就被调换成桶的 x 和 z 值。这里我们设定 link 的 y 轴的值为 0.5，这样鼠标的数据变化就不会影响桶的 y 轴的值。

```
#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] )
```

最后，我们用连接将虚拟人的视角作为渲染虚拟环境的视角。首先抓取虚拟人头部的骨骼然后连接到主视角上。

```
#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 )
```

练习

1. 运行 coordinate system demo 点击“scramble”选项。结果我们看到一个随机大小、位置和朝向转换的父和子节点。现在尝试将节点移回到原来的状态。

2. 添加六次“art/barrel.ive”模型，将他们摆放成下图的样子。

3. 在 Transformation and the scene graph 这个例子中试着让蜜蜂绕着鱼转而不是人转。

4. 在 Transformation and the scene graph 这个例子中将主视角连接到虚拟人的视角。

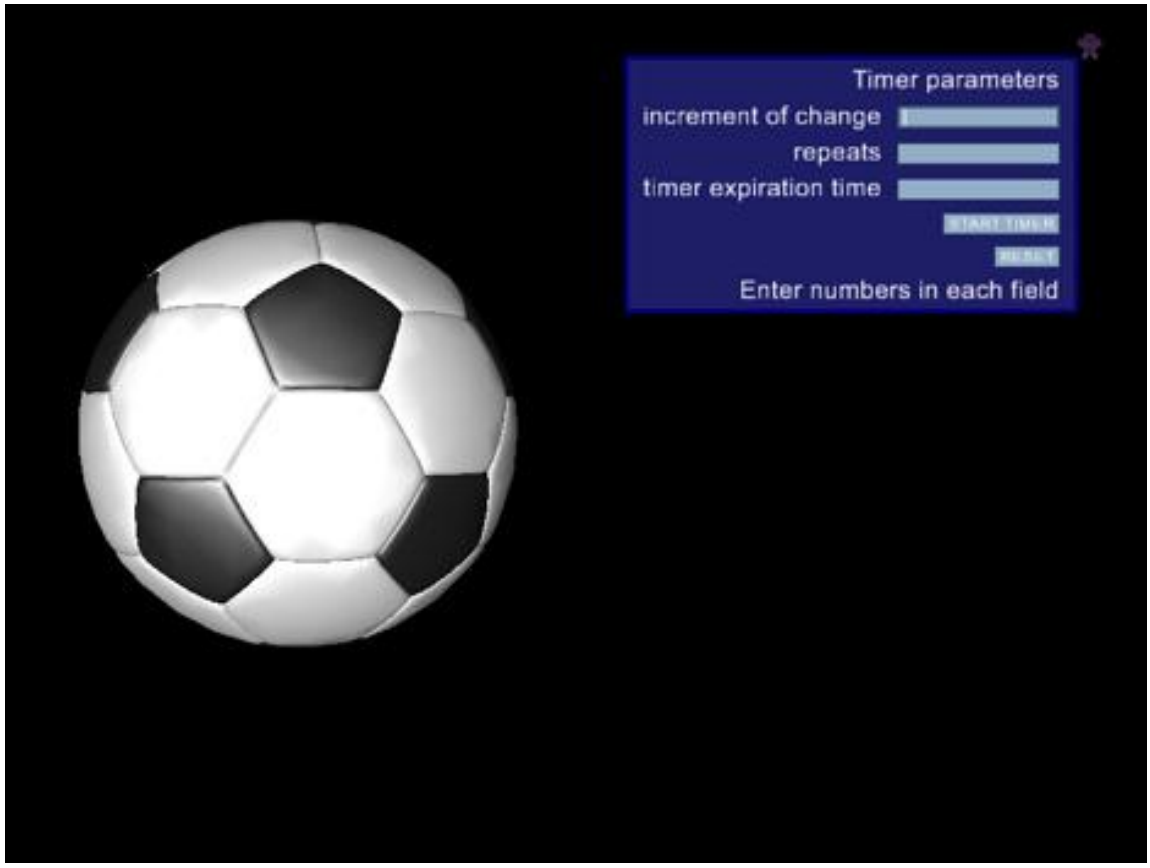
模型运动

上面的部分我们讲到的三维模型转换会改变模型的大小、位置和朝向。当然在真实世界中有些转换不会瞬间完成。一辆车从一个地方开到另外一个地方，这个过程肯定会经过一定的时间，车不会瞬间移动。植物长高的过程也是缓慢的，他们不会突然变大。为了模拟这些渐变的过程，我们需要将这些转变加上一个时间参数。这部分我们会涉及如何用一些命令实现这些功能。我们也会介绍如何触发这些运动，让整个过程变得动态和交互。

运用计时器

模拟运动的其中一个方法就是用计时器。如果你已经读过场景图这个部分，你可能已经对计时器有一些了解了。计时器可以在特定的时间内被触发，然后在时间终止前调用一个函数。如果

该函数和转换有关，那么你可以通过不断的触发计时器调用函数来实现让模型运动的功能。例如，你想让一个篮球向前滚动一米，一个技巧就是先设置一个函数让球滚动的幅度很小并且计时器被触发的间隔时间很短，这样每一次计时器被触发时调用函数球则向前滚动一点，计时器不停的被触发看起来就像是球缓慢的不断的向前滚动一样。



运行 timer demo。这个例子的目的就是让球连贯的滚动。屏幕中的菜单中可以设置每一次球滚动的幅度和滚动多少次也就是计时器被触发多少次，还有计时器计时时间。设置好后点击开始计时器，你就会看到你设置的结果。试着设置球滚动幅度，直到让它连贯滚动为止。

实例：运用计时器

在这个例子中，我们模拟一只蜘蛛织网的过程。在这个例子中注意我们是如何设置计时器以及蜘蛛运动轨迹的。

首先，我们先设置视角位置、背景颜色和添加一只蜘蛛。我们不会解释例子中的每一行，因为一些部分已经讲到了或在以后会涉及。例子中会有设置蜘蛛运动状态的语句 state，其实它不会真的让蜘蛛的位置发生变化，只是模拟蜘蛛的腿动而已。

```
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)
```

现在我们设定给一个用来模拟蜘蛛运动轨迹的计时器。我们先定义一个函数，然后利用 ontimer 命令不停的调用这个函数。每 0.01 秒计时器启动。在“move_spider”这个函数中，我们用“increment”这个变量来增加运动的步伐。这个变量实际上用来定义蜘蛛在当下的位置和朝向，这里会用到一些三角函数的知识，但是不要担心，我们只是用到而已，这个例子的重点是掌握计时器的使用。

```
#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 )
```

运行上面的代码看看结果如何。

在下面的这段代码中我们要添加一个蜘蛛网，这样可以更清楚的知道蜘蛛是怎么向前运动织网的。我们用到的网是一个“on-the-fly”对象。该对象可被动态的创建，当我们要在运动的同时创建对象时就可以用到此方法。这个例子中 On-the-fly 对象是由一系列由线连成的点组成。在下面的“lay_web”函数中我们先设置点，蜘蛛可以这些点为基础向前移动，下一个点就通过蜘蛛的身体连接到前一个点。在“ontimer”这个命令中我们不断的调用这个函数。

```
#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)
```

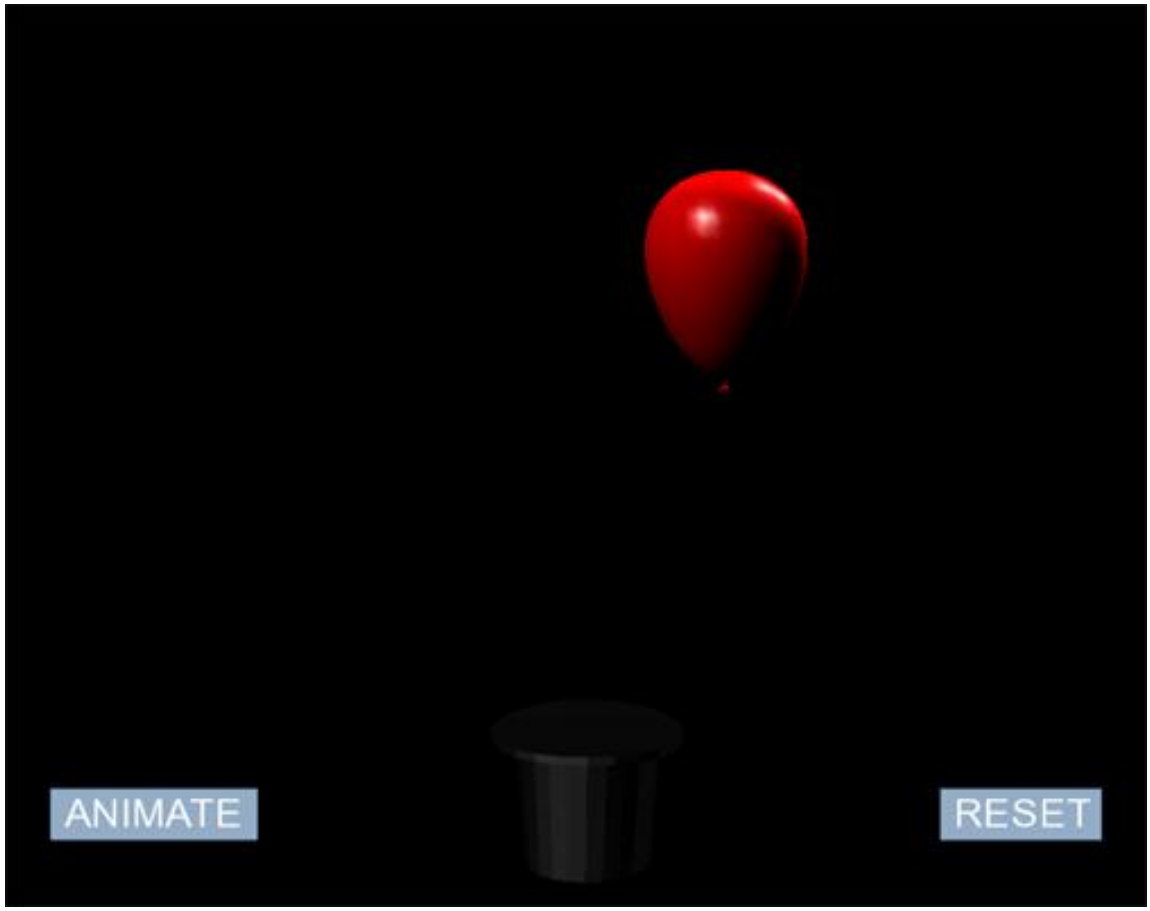
运行上段代码看看结果如何。

试着改变“increment”的值（例如，“increment+=.01”）。一个很小的数值变化就会给蜘蛛运动的结果带来很大变化。再试着改变计时器的计时时间，也就是 ontimer 的第一个参数的值。看看蜘蛛运动的效果有什么变化。

动作队列

计时器是模拟流畅的动画的基础，三维模拟引擎包含有内置的模拟动画的方法，这些方法包含计时器功能。在 Vizard 中，这些方法包含 action 库中的函数（例如,vizact.py）和物理引擎。现在我们就介绍 action 库，物理引擎会在“物理建模”中介绍。

Action 库允许你为一个节点建立一系列的动作。当一个节点有一个动作队列时，每个动作会依次的按照队列里的顺序执行。在模拟动画中队列是一个非常有用的办法用来排列和执行动画的每一步，而不需要你一步一步的去监控这一步有没有结束下一步该执行哪个动作。队列也可以被动态的改变，你可以清除队列的动作或者停止队列的执行任务。



运行 animation demo，看看这些动作是如何被执行的。当程序打开后，你可以看到屏幕上的菜单，点击上方的菜单的“actions”选项，选择动作。然后在“sequence style”选项中选择动作的状态。如果你让这些动作顺序进行，那么程序会依次的执行这些动作。如果你让这些动作并行执行，那么这些动作会结合在一起而同时影响物体的状态。

模型中的第三个选项可以让你在动作序列中添加一个等待的动作。等待动作会被排列在队列的指定位置然后等待特殊事件发生。当事件发生后，队列继续执行。这个例子用到了两个等待命令。一个是等待空格键按下，另外一个是等待两秒。

另外一个动作用来支配多个物体就是信号功能。Action 库允许你在一个队列中添加信号传输和信号等待的命令。当一个节点遇到了信号传输动作，它就会发送一个全局信号。任何节点接到这个信号后就会执行下面的动作。例如，在“signal others”选项下选择“signal other

objects”。现在当你点击 animate，你的第一个物体就会执行关于它自己的所有动作。做完之后，它会发送给其他节点一个信号。

实例：给物体添加动作

这个例子中用了很多动作命令来为读者展示 action 库是如何工作的。另外介绍如何给多个物体添加动作。

首先，在这个场景中添加一些三维物体和设置光。在添加气球时，我们用 Python 中的 random 模块来给气球设置一个随机的颜色。我们还会给每个气球设置反射光，我们会在光的部分详细讲到。下面我们给整个环境设置一个地图，实际上就是一个覆盖全场景的材质图。之后我们用 addAudio 命令添加一些声音文件。我们还会给环境添加一些光，在光部分会详细介绍。最后我们添加一个视点并设置它的位置和朝向，可以使我们清楚的看到整个场景。

```
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])
```

现在，我们开始给程序添加一些动作让程序更有意思。我们用这些动作来模拟气球的充气过程。首先添加一些常数，他们在动作中会用到，

```
#Set constants for actions.
INFLATED = [2,2,2]
DEFLATED = [.2,.2,.2]
```

```
BREATH_LENGTH = 3
DEFLATE_LENGTH = .1
```

现在我们用 vizact 库来建立一些动作。第一部分的动作来模拟气球充气的过程，从而让气球看起来越来越大而且变得更透明，同时会播放声音。我们把这些动作放到一起变成一个动作，这里用到“vizact.parallel”命令。剩下的部分是让气球充完气后飞走，然后泄气最后落下。

```
#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) )
```

下段代码将这一系列从充气到飞走到泄气再到落下的动作连接到一起。我们用“vizact.sequence”因为我们想让这些动作依次的发生。

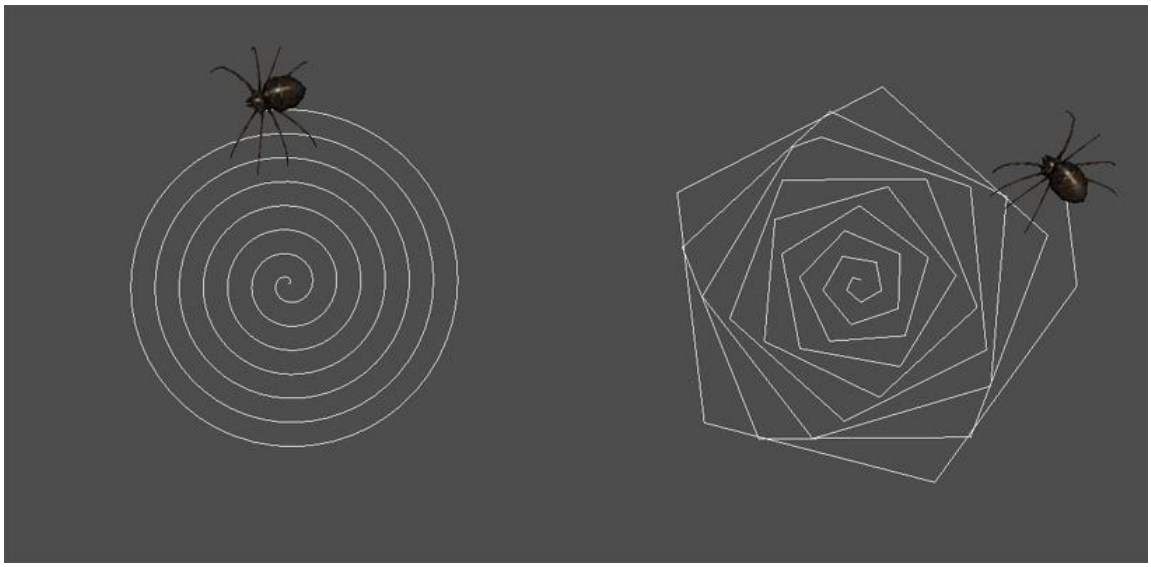
```
#Create the lifecycle of the balloon with a sequence.
life_cycle = vizact.sequence( [random_wait, inflate, float_away,
deflate, fall], 1 )
```

最后，我们给每一个气球都添加一个键盘触发这一系列动作的命令。我们用到“vizact.onkeydown”命令，空格键一被按下后就调用“balloon.addAction”命令并执行“life_cycle”。

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

练习

1. 修改“Using timers”代码，让蜘蛛快速织出非常平滑的网，再让它缓慢的织出一段段直线的网。



2. 添加四个不一样颜色的气球然后设置不同的按键来触发每一个气球的充气过程（添加一个红色气球，当你按‘r’键时触发它充气，按‘y’键来触发黄色气球充气，等等）

3. 修改“Using actions”代码来让环境中的虚拟人和气球一起飞走。

光

为了让一个三维模型可见，你需要光。想要掌握这部分最简单的办法就是掌握现实中的光如何工作，因为虚拟世界的光和真实世界的原理一样的。首先光源会向周围发射光。每一种光源都会发射不同方向、颜色和密度的光。当这些光到达一个物体时，他们就会从物体表面反射回周围环境中。而物体本身的特性，例如颜色和发光度，就会和光本身的特性交互从而决定反射光的特性。这些反射光还会到达其他的物体表面。最后只有一些光会到达人的眼睛，从而使你看到这些物体。

数字模拟技术试图将这些光的特性反映出来。在这部分，我们我们添加一些光来创造一系列的效果。我们也会介绍物体表面的属性如何影响光的效果。

设置光的一个重要概念就是**局部（local）**和**全局（global）**照明。局部照明主要处理单个物体和影响它的光源。全局照明处理影响所有物体的光源。所以全局照明模拟的是从一个物体反射到另外一个物体的光。全局变量的光会模拟真实环境，这个过程会计算物体之间的光线反射，计算过程比较复杂和花费时间。这里有一些技巧可以使用，下面我们详细介绍。

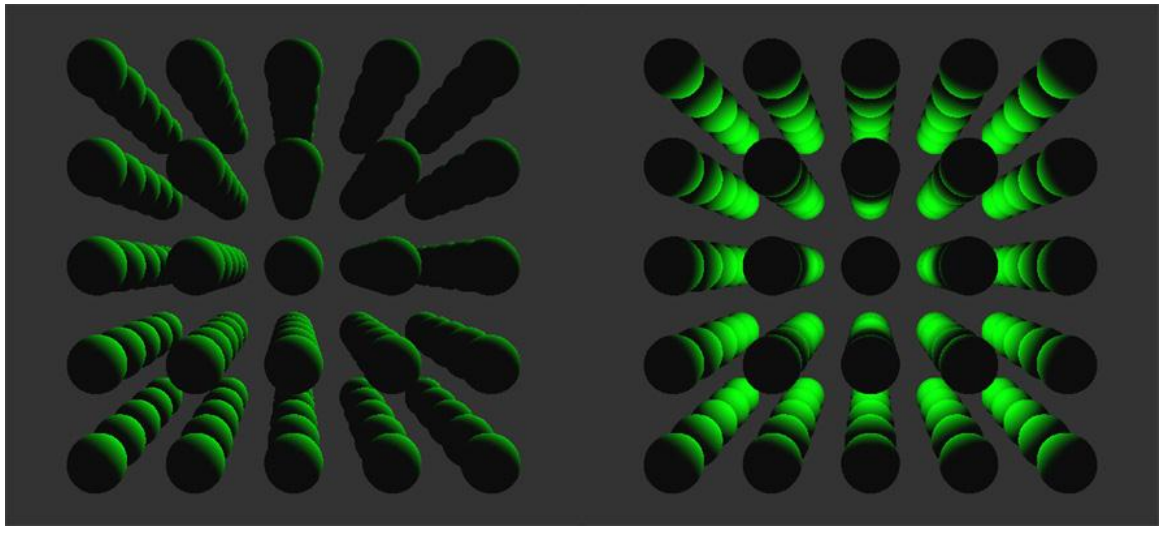
注意：当你在 Vizard 中创建一个世界的时候，程序会自动的添加一个“头灯”点亮场景。它位于视角的位置，所以你会看到相应的场景。在下面这个部分，我们移动头灯的位置并介绍添加光的其他选项。

光源

OpenGL 会允许你在一个场景中添加 8 个光源。光源可以是任何颜色和密度。这里有两个概念要区分开，一个是**平行光 directional lights** 一个是**定点光 positional lights**。平行光源来自于一个给定的方向但是却没有一个空间位置。来自于平行光源的光向同一个方向平行的发出光。这种光源会照射到在场景中的所有物体。所以，平行光源模拟的是从很远的地方来的光，就像是太阳或者月亮。

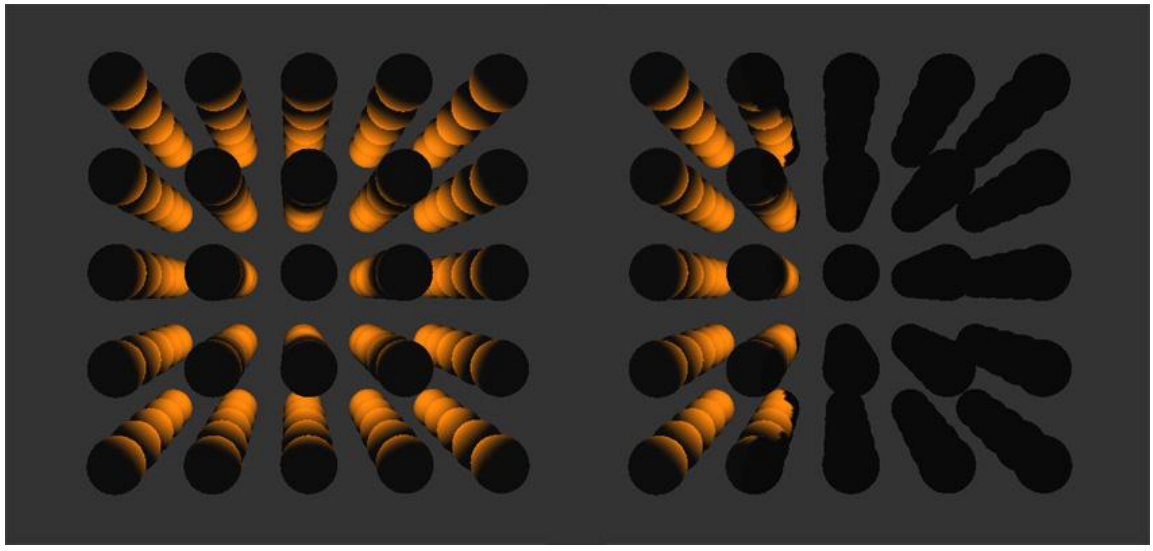
而定点光源是从一个固定位置发出。发出的光源会向四面八方散去。所以在该点上面的物体会被照亮，下面的和左面的还有右面的物体都被照亮。

为了更加清楚平行光和定点光如何工作，请运行 light source demo。当程序运行后到“positional or directional”的菜单中试着选择这两个光源。请注意这两种光照下场景中的光有何不同。（在这个例子中，光源是位于这些球体方阵的中心。）



当你放置位置光源时，这里有些选项可供选择。你可以设置为光源向四面八方发射（一个**点光源**）或者将光源的焦距缩短让它直接照射到一个特定的方向（一个**聚光灯**）。对于未知光源，你还可以改变光源随距离衰减的方式。这里有多重方法计算衰减。在 light source demo 中，找到“attenuation”菜单，你可以改变菜单中的选项从 none 到 quadratic（该选项为光衰减的标准计算方法。）

因为点光源有一个照射的范围，他们也有一些可以调整的参数包括点光源的范围、方向和光源随着距离衰减的程度。在 light demo 中的光源的方向和范围是固定的，但是你可以通过选择“spot exponent”来改变衰减范围。



光和物体表面

简单看来，光是从一个光源发射出来，例如一个灯泡、太阳。但是当我们创建虚拟世界的时候，光和物体表面之间的关系就变得很重要，因为物体表面会反射光。这里介绍四个重要的物体不同表面造成的效应：漫射光、高光、环境光和发射光。

漫射光 Diffuse light 从一个特定的方向发出来。它照亮在该方向上的物体，而不会照到不在该方向的物体（在照射的过程中，光的强度随着距离衰减）。漫射光从一个物体的表面上均匀的反射出来，所以我们改变视角看这个物体时光不会改变。这种光会给物体添加阴影而让它看起来更有三维的感觉。漫射光需要一个光源用来定义光从光源发出来的方向。(在 light source demo 中用到了漫射光

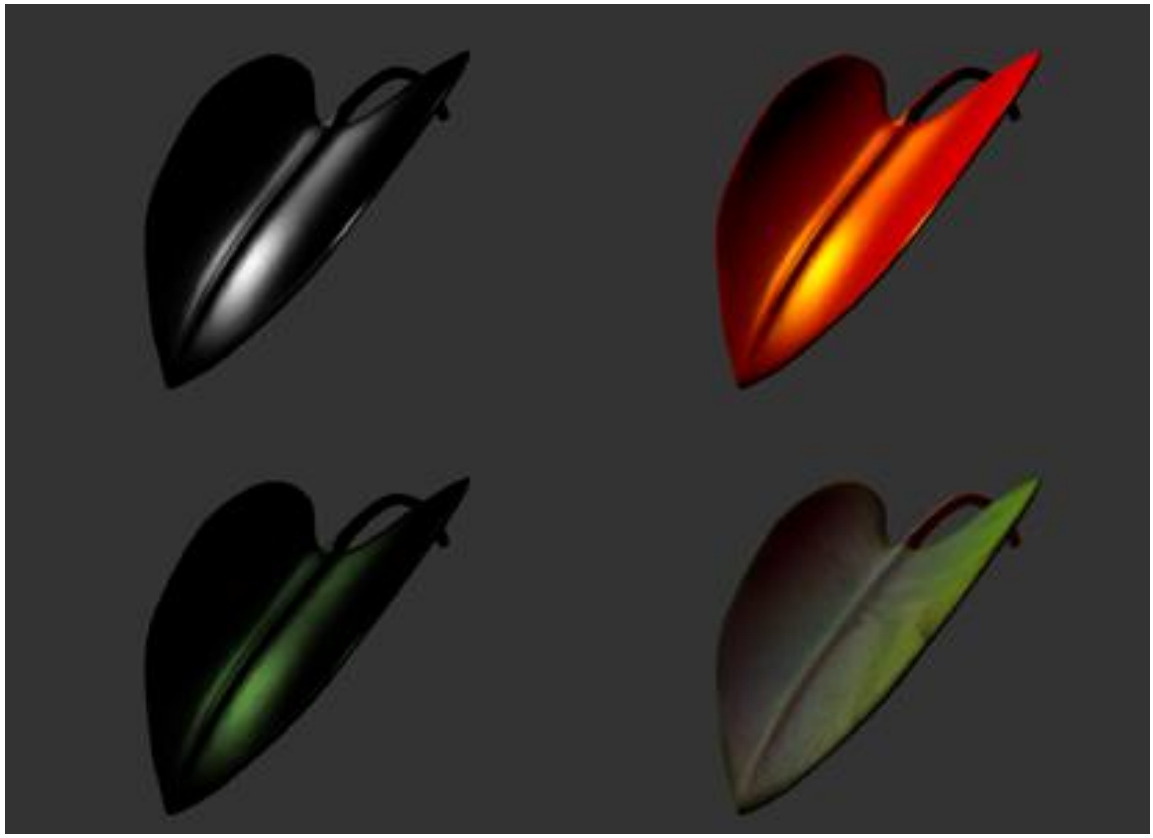
高光 Specular light 和漫射光相类似，都是从光源沿着一定的方向发出的光。但是高光并不从物体表面均匀的反射出来。运用高光可以使物体看起来闪闪发亮。

运行 light surface demo 来考察镜面反射光如何工作。在程序运行后的屏幕上找到“specular”菜单然后赋予这个反射光一种颜色。然后到“shinness”菜单下输入“128”。设置这个参数可以影响高光的锐度。现在用鼠标改变观察视角。你可以看到随着视角不同叶子表面的光也看起来不同了。如果你看不到这个光的变化，试着从“texture”菜单中去掉叶子的材质。

环境光 Ambient light 赋予物体表面一种特性，也就是物体接受来自各个方向的光照并将它反射到各个方向上去。因为环境光不是来自一个特定方向的光，它也不需要光源。

运行 light surfaces demo 来看看环境光如何工作。在“ambient”菜单下试着改变环境光的颜色。

最后一种光为**发射光 emissive light**。发射光模拟的是从一个物体发射出来的光。这种光均匀的从物体发出光。它可以模拟例如像灯泡这样的发光物体。注意，这种光依然是局部照明。所以改变发光物体的光特性让物体变得更亮时，并不会将周围物体照的更亮。



现在操作“emissive”菜单更改发射光的参数。注意这些叶子表面的光特性会影响叶子渲染出来的效果。所以物体渲染的最终结果是和光还有材质等其他特性共同作用的结果。

在本章中介绍的光类型可以为渲染环境增色，光本身的特性也类属于物体的材质特性（包括材质贴图）。三维建模软件（例如 Maya，3DS Max 等）包含大量的工具用来处理物体的材质。物体材质的基本特性再结合它的发光和反光效果共同造就物体的三维外观。

实例：为场景添加光

在本例中，我们会为环境添加各种光。

首先，我们先打开一个空白脚本文件，然后为场景添加一些三维物体，再放置观察视点以看到全景。

```
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 为观察视点准备了一个默认的头灯。运行上段代码你就可以借助这个头灯观察到渲染出的环境。下段代码关掉了头灯，添加下段代码后再运行程序，看看渲染场景有什么变化。下段代码中我们用 viewpoint 库中的“getHeadLight”命令抓取了主要视点的头灯。抓取后我们就可以用 node3d：light 命令来关闭头灯。

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

现在我们添加一个平行光来营造月亮照射的效果。注意下面设置位置的参数。第四位参数我们设置为 0，也就是设置光为平行光。1 为定点光。前三位参数表示光从[x, y, z]点发出来。我们设置 y 轴为 1，xz 轴为 0。光从正上方发出来。

```
#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 )
```

注意这个平行光如何照射它周围物体的。再试着改变光颜色和密度的参数，看看结果如何。试过后，我们将月光关闭：

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

接下来我们为物体添加发光特性，让物体成为一个光源。我们为一个灯笼添加发射光。

```
#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)
```

现在我们运行这段代码看看渲染的结果如何。试着改变发射光的特性，看看结果如何。下面我们再关闭发射光：

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

我们为整个场景添加一个光源，把它和火把连接。模拟火把发光的效果。

```
#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 ) )
```

练习

1. 在“lighting a scene”中移除所有的光，在虚拟人的上方添加一个光源将虚拟人照亮。

2. 在虚拟世界中添加一个球体（"art/white_ball.wrl"）。用<node3d>.color, <node3d>.specular, <node3d>.shininess, <node3d>.ambient, and <node3d>.emissive 命令来变化光的漫射、高光、环境光和发射光的参数，让这个球体看起来像一个保龄球或是一个太阳或是一颗樱桃。

建模物理

在前面的模型运动章中我们讲到了可以让模型动起来的方法。但是如果你想要模拟真实世界中的物理事件，你需要用到一些新的方法。例如让一个虚拟的球在地上弹动后并滚到远处。

在这一章我们讲到的物理引擎会帮助我们模拟真实世界中的物理事件。**物理引擎**应用广泛，从计算和处理大量的复杂物理事件到预测物物运动关系。它可以将模拟的过程变得简单，虽然不一定精确模拟，但是效果逼真。在这部分我们介绍这些可以时时模拟的物理引擎，学习如何使用它检测碰撞和模拟碰撞时相互作用的力。

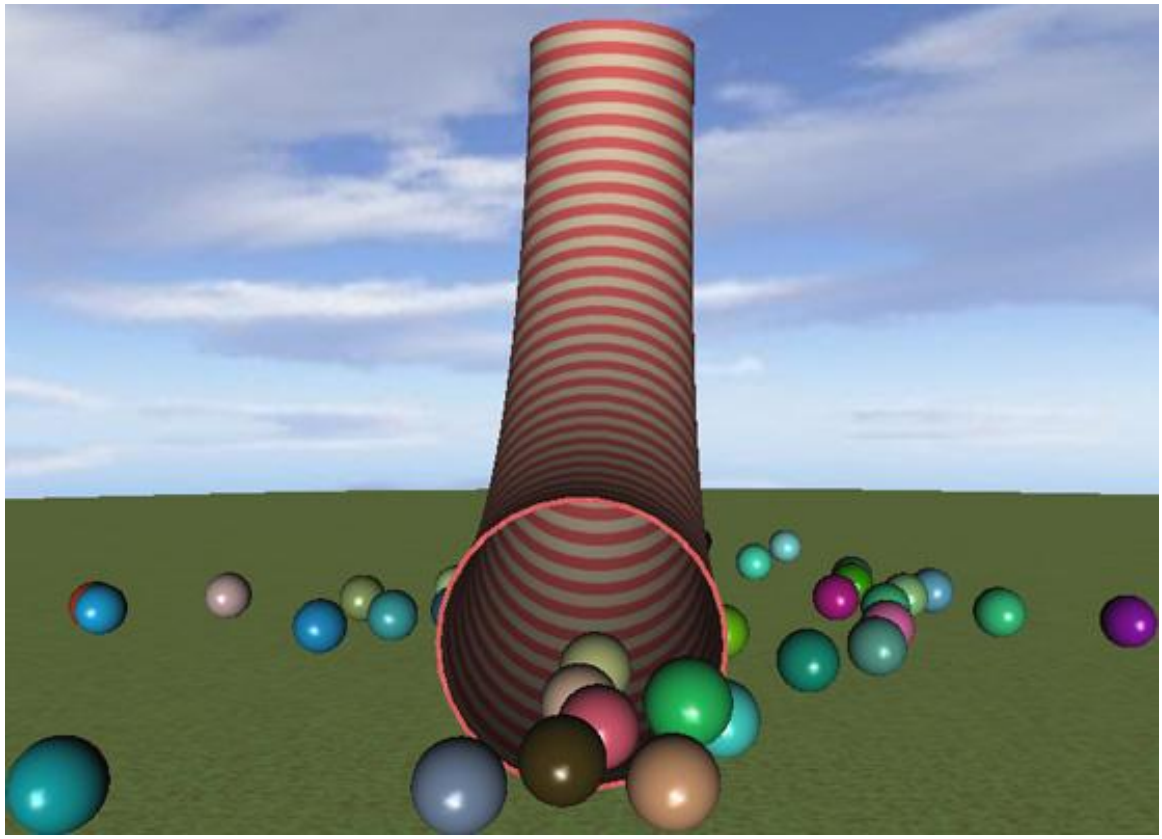
碰撞区域

碰撞检测是物理引擎的一个重要功能。为了让碰撞检测时时进行，物理引擎用到了**碰撞区域**。碰撞区域就是物体周围的区域，这个区域定义了物体碰撞他物或是被碰撞的边缘。在理想世界中，碰撞区域就是物体轮廓。区域形状越复杂，碰撞检测就会越敏感，因为计算机在时时的计算两个预要碰撞的物体的区域是否交叉。但是经常我们不会用精确的物体轮廓。

碰撞区域通常用来定义一个物体在什么情况下会碰撞到另外的物体。当两个物体的轮廓有交叉时，碰撞就被检测到。另外，碰撞区域的形状也会决定物体碰撞后的运动轨迹。就像是一个球会沿着一个面平滑的滑动，而一个盒子就不会。所以选择正确的区域对准确预测碰撞后的轨迹是非常重要的。

运行 physics demo，看一看碰撞轮廓如何工作。当你第一次运行程序时，你会看到一堆立方体从天上掉下来，每个立方体都定义了自己的碰撞轮廓，他们掉到同时定义了轮廓的桶上。这些立方体也同时会相互碰撞。“到 target object”菜单下改变桶的碰撞轮廓，看看哪种轮廓会最好的模拟碰撞过程。

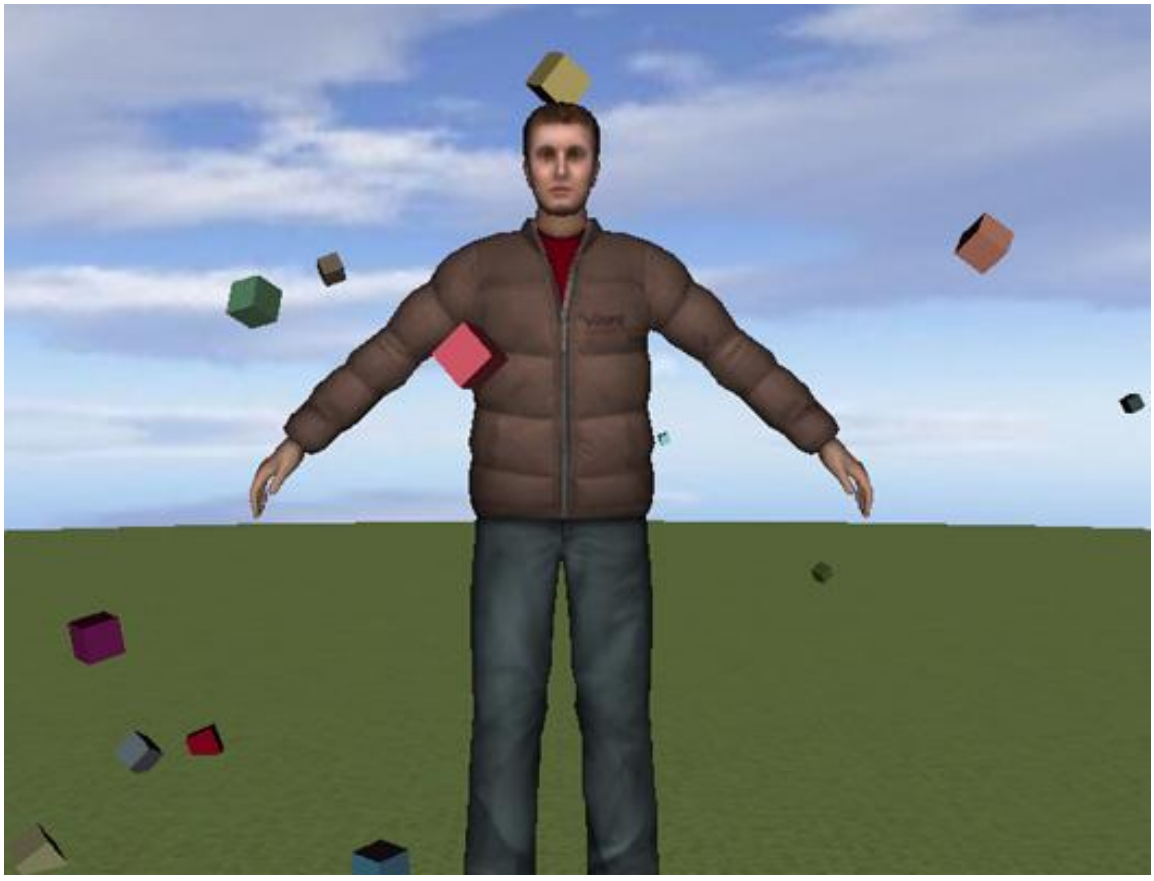
注意：你可能已经知道了碰撞网这种碰撞轮廓是最能反应碰撞过程的。事实上，它是唯一的模式可以允许从上面掉下来的物体通过管状模型。这是因为它是基于碰撞网本身的。使用碰撞网的缺点是, 不能应用动态物理(力)。所以，碰撞网最好应用于别的物体从碰撞网物体上反弹的情况，但碰撞网物体不受影响。



力

除了碰撞检测，物理引擎还可以用来计算作用力。这个功能有时候被称为物理模拟的**动态效果**。物理引擎模拟的作用力最基本的就是重力。当重力模拟打开的时候，物体就会以 9.8 m/s^2的重力加速度下落（当然你也可以自己设定这个数值）。可以在 physics demo 中看到，力不是单独作用的。将物体之间的碰撞检测和作用力的计算过程结合就可以使物理引擎来模拟物体之间作用的动态过程。

除了碰撞轮廓本身的形状，碰撞轮廓还有其他的特征用来决定力是如何作用的。比如它的坚硬度、摩擦系数、弹性和密度，这些特性都会影响物体碰撞后的运动轨迹和速度。再次运行 physics demo，试着在“target model options”菜单中的“dropping objects options”中改变其中的参数。



除了重力，物理引擎还可以模拟很多其他的力。被模拟的力也有很多参数可以调整：力作用的事件，力的方向，力作用在物体的哪个轴，力作用在物体的哪个部位（是在物体的重心还是在别的地方）。这些基本的参数要结合起来控制例如弹簧、发动机、推进器以及其他可以改变物体运动速度或转矩的作用力。物理引擎可以允许你建立一个接合处，这样物体可以绕着这个接合处旋转,例如拉开门，门就可以绕着合页打开。

实例：物理引擎

在这个例子中，我们会用到碰撞检测来模拟一个飞刀场景。我们先添加一个马戏团的场景，在一个游戏中一个虚拟人被绑在一个旋转的圆板上，他的周围还有一些气球，用户用飞刀来戳破这些气球。

在程序的第一部分，我们先设置场景和添加灯光。我们还需要关闭鼠标导航功能好让鼠标来完成其他的功能。再放置视角到适当的位置。最后添加模型。

```
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])
```

下面，我们将旋转的木盘和物体连接起来，木盘作为源而虚拟人作为目标。我们先对虚拟人做一个预先的移动让他位于木盘正确的位置。因为人和木盘连接起来了，所以木盘的任何运动都会连接给人，木盘转的时候人也跟着转。

```
#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) )
```

我们抓取虚拟人的骨骼，然后把他身体的一些部位放到正确的位置（这部分我们在虚拟人的部分会详细讲到）。

```
#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 )
```

现在我们添加气球，调整它们的外观，将它们连接到木盘上。

```
#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 )
```

我们给整个场景添加一些动作。首先为虚拟人添加一个声音文件，当飞刀刺到他时他会叫。再给气球添加一些动作，刀将气球刺破的动作。

```
#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] )
```

这里我们定义物体的碰撞轮廓。首先我们给木盘、气球和虚拟人添加一个碰撞网，给地面添加一个碰撞平面。因为我们还会给刀添加一个重力作用，所以这里给刀赋予一个碰撞盒。再为刀开启一个碰撞提醒的功能，这样 Vizard 会一直监视刀的状态直到碰撞事件发生。

```
#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  )
```

在下段代码中我们要定义一个函数用来处理刀刺物体的过程。这个函数有一个 3D node 参数。我们在函数中还关闭了刀的所有物理力，这样刀就不会受任何外力影响。然后将到刀和这个 3D node 连接起来。

```
#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 )
```

这里我们定义一个向目标物扔刀的函数。这个函数会接受来自鼠标在屏幕上的位置信息，然后创建一个从屏幕位置到虚拟世界的向量。如果刀和其他物体有连接，那么取消这个连接，同时清除刀现在的速度。然后设置刀在该向量上的速度再把刀放到目标位置上。当鼠标事件发生时这个函数被触发。

```
#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)
```

当有碰撞事件发生时下面这个函数被调用。因为之前我们为刀添加了一个碰撞提醒，所以当刀碰撞到其他物体时这个函数被调用。它的参数接受来自碰撞事件传来的数据。e.obj2 的值即为刀碰撞到的物体。我们也会用这个值来决定之后的动作。如果刀碰到人，那么刀就会留在人身上，而虚拟人也有相应的动作。如果刀碰到木板，刀留在木板上。如果刀碰到气球，气球就会被扎破。

```
#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)
```

当然我们也要设置气球破了后还会回到原来的状态。下面的函数就处理这个过程，简单的让气球回到原来的大小并让它恢复物理能力。

```
#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 )
```

最后，我们在这个场景中启动物理引擎 viz.phys.enable()。注意物理引擎不会自己启动除非你调用这个启动命令。

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

练习

1. 练习写一段程序实现鼠标点击后一个球从几米高的地方掉下。添加一个地面，赋予这个地面一个碰撞平面。在为这个球添加一个碰撞轮廓。定义一个函数可以让球在落到地面后弹起。

2. 在上面的这段代码中添加一个碰撞事件。当一个球碰到别的物体时球的颜色发生变化。

虚拟人

虚拟人（avatar）属于特殊的一类三维模型。这些虚拟人就是虚拟世界中的人，他们可以走动和做各种动作，运动身体的各个部位。

注意：从技术角度讲,虚拟人是由我们真人时时控制的在虚拟世界中的人的替身。而由电脑控制的模型只能称之为被控制的**客体（agent）**。两者具有不同概念。所以在这本书中我们称这种在虚拟世界中数字化的人为虚拟人。

虚拟人的结构

虚拟人的结构和其他三维物体的结构有所不同。特别的是，虚拟人具有骨骼结构。它的**骨骼结构**是层级结构就像是之前我们讲过的场景图结构。这种骨骼结构的单元是单个骨头。每一个骨头继承了来自上级骨头的转换。但在实际的虚拟人的运动模拟中，各个骨头是不可见的。骨头的运动只是控制覆盖在虚拟人身体的**网**结构的运动。这个网**连接**在骨骼上。这些连接可以是简单的，也可以非常复杂如一块网连接虚拟人身体一块区域的多个骨头或不同区域的多个骨头。当骨头运动时，连接在骨头上的网就被带动起来从而造成虚拟人的身体运动。

运行 avatar demo 来操作虚拟人的骨头运动。在“Pick a bone”这个菜单下，找到一个选择身体左右或中心骨头的下拉菜单。选择一个。在屏幕右上菜单中有一个滑动条可以旋转骨头。试着将虚拟人的手放置到他的身体背部。注意当你旋转例如名为“Bip01 Spine1”的骨头时，虚拟人的该部位的网结构就会被拉伸或是扭曲。

（当你关闭掉这个程序时，程序就会将虚拟人的姿势保存到一个名为“avatar_bones.txt”的文件里。用 debaser 这个模块中的 pose_avatar 命令可以调用这个姿势。运行“links example.py”这个脚本。）

虚拟人运动模拟

虚拟人的运动模拟比较复杂。最基本的运动也比较难例如行走的动作。不同的骨头要以不同的速度运动，任何一个小的差错都会导致身体的不正确的扭曲。

对于时时交互的模拟，操纵骨头的运动来实现模拟人的运动比较方便。但对于复杂的模拟还是要通过运动模拟软件来实现。模拟软件可以帮助你创建动作序列，你可以按照你想要的动作顺序来组织这个序列。

虚拟人的动作模拟一定要和现实的运动情况相符。你不能让虚拟人做现实中人做不到的动作。现在的动作模拟软件可以通过解决给定位置下的**反向动力学（inverse kinematics）**从而帮助你实现现实的动作模拟。反向运动解决的是为了达成所需要的姿势而设置的关节可活动的角度（例如胳膊上的关节）。关节可活动的模式对于人体来讲只限于很小的范围。所以模拟软件可以帮助你计算一个关节的运动带动其他部位运动的正确的模式。

动作模拟的两种方法为关键帧和动作捕捉。关键帧方法就是在一定的时间区域内将虚拟人摆放在特定的姿势上。通过反向动力学运算的正确运动模式，模拟软件可以推算每个关键帧之间的身体运动状态，这样模拟后的效果就会变得很平滑。

目前的模拟软件可以将关键帧的模拟处理的非常的好，而虚拟人的动作模拟最实际的使用动作捕捉方法。通过动作捕捉，真实人的动作可以被捕捉到从而连接到虚拟人上，从而让虚拟人做人的动作。因为这种方法捕捉的是真实人的运动模式所以效果会最接近真实情况。

实例：虚拟人动作模拟

在这个例子中我们用各种方法模拟一个男性和一个女性虚拟人的动作。

首先我们在虚拟世界中添加模型，然后放置观察视角到合适的位置。

```
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)
```

现在我们给虚拟人一个动作模拟。这个动作会一直持续直到我们调用另外一个动作。

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

下面这段代码我们让男性的头部一直朝向女性虚拟人。首先用"getBone"命令抓取男性的头部骨头。我们再锁住头骨让他暂时先不运动。（如果你锁住一个骨头，该骨头不会有任何的运动）。在下面的函数中，我们先要获取男女虚拟人的位置，然后找到他们之间的夹角。然后我们用这个夹角来设置男性头部的朝向。最后我们用一个计时器命令调用这个函数。

```
#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 )
```

下段代码我们用一些按键来触发一些运动。首先，用'd'键来触发混合功能，我们将虚拟人的第五个运动模式即跳舞混合到他当前的运动中（我们设置的是第一个运动模式），这种混合运动持续 5 秒钟。用‘s’键解除第一个运动模式。

```
#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 )
```

我们现在模拟女性虚拟人行走的运动。我们会用到 vizact 库中的 walkto 命令。这个命令接受一个位置参数。然后这个命令会让虚拟人向该位置行走从而模拟走的运动。我们会多次用到该命令使这个虚拟人来回走动。

```
#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.FOREVER )
female.addAction( walking_sequence )
```

现在我们定义一些物理性性状和一个函数，当有碰撞事件发生时调用该函数。关于物理命令我们在建模物理部分由详细说明。我们在这部分着重通过“onCollideBegin”函数理解虚拟人动作模拟。当有碰撞发生时我们执行第十六个运动模式。该模式让虚拟人向后倒去。之后我们设置“freeze=True”好让虚拟人在倒下后不做任何运动。如果不设置该参数，那么虚拟人在倒下这个动作结束后回到一个 idle 模拟状态。在这个函数中我们还会让女性虚拟人拍手。拍手后她回到 idle 模拟状态。

```
#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)
```

我们在给程序添加一些交互功能。当按空格键时一个球飞向男性虚拟人（这部分的命令我们在建模物理中讲到过）。

```
#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 )
```

最后，我们要定义一个函数用来重新设置虚拟人的运动状态。用“stopAction”命令来解除第十六个运动模式，再用“state”命令来让虚拟人一直处于第一个运动模式的循环。我们还可以为女性虚拟人添加一个走路的运动状态。最后一行启动物理引擎。

```
#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()
```

`

练习

1. 练习用 avatar demo 来让虚拟人处于挥舞的位置。写下你想要操纵的骨头名称和骨头的欧拉角。现在添加一段命令来添加虚拟人和设置角度（用<bone>.setEuler 命令）再用 viz.AVATAR_LOCAL 命令来设置骨头的旋转坐标系。

2. 在练习 1 中的程序中添加一个计时器可以让虚拟人的手来回挥舞。

3. 写一段代码让虚拟人拍手（这个动作模拟包含在“vcc_male.cfg”和“vcc_female.cfg”文件中），添加两个键盘触发功能，按‘c’键时在原来动作中混合一个欢呼的动作，按‘s’键时去掉欢呼的动作。

硬件

输入设备

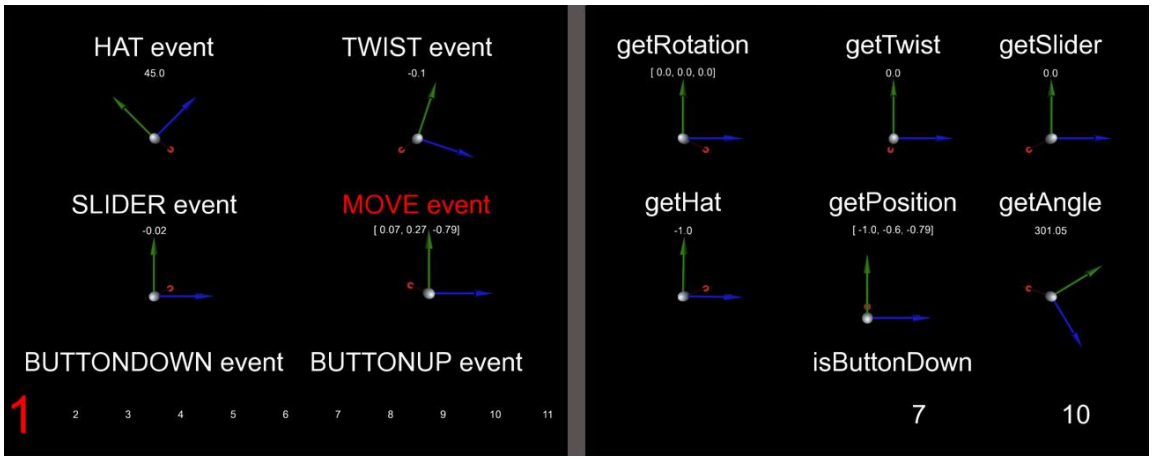
硬件将用户和虚拟世界连接起来。输入设备允许我们在虚拟世界中导航、和虚拟物体互动，而输出设备可以让我们看见、听见和感知到虚拟世界。我们在本书中用到的主要的输入设备即为键盘和鼠标。其实还有好多其他的输入设备，比如运动追踪设备。在本章中我们会了解如何使用简单的输入输出设备比如说有喜感来和虚拟世界互动和导航。

本书中主要用到的输出设备为显示器。当然还有很多其他的显示设备。在“显示设备”章节中，我们会详细介绍。

运用输入和输出

一般来讲，连接硬件设备和虚拟现实软件的时候你需要一个插件以便获得从设备传来的数据。Vizard 支持大部分输入和输出设备（查看 Vizard 帮助文档中的硬件部分，可以知道 Vizard 都支持哪些设备）。Vizard 有一个模块叫 vizjoy 可以连接游戏手柄和游戏杆这类 USB 口的设备。我们在这章会详细介绍这个模块。同时你需要准备一个游戏杆或者手柄来运行这章的例子。

运行 joystick demo 看看游戏杆是如何运行的。在屏幕上方的菜单中选择“Joystick capabilities”。这个菜单中列出了 vizjoy 这个模块都可以使用你的游戏杆的哪些功能，例如游戏杆指向的轴，游戏杆有多少个键，是否有力反馈。我们一会会讲到这些功能。



这里有两种方法来获得输入设备传来的数据—**事件（events）**和**取样（sampling）**。当你为一个事件注册了一个回调，实际上就告诉程序等到一个事件发生后调用一个函数。所以这个函数只有在特殊情况下才被调用（我们在编程基础的“事件”部分我们已经讲过）。

对于取样方法，程序会持续不断的获取输入数据。你可以用计时器重复的从输入设备取样。因为这种持续不断的取样特性，你可以用输入设备进行导航。

运行 joystick demo。到“Events vs sampling”菜单中选择用事件或是抽样方法来获取数据。在本例中我们用 vizjoy 事件。你会看到当 vizjoy 事件发生时程序显示的环境中的相应的文字会变红。你还会看到屏幕下方列有数字，这些数字代表相应的游戏杆上的按键。

许多游戏杆有力反馈功能，如果你使用的游戏杆有此功能，到“Force feedback”菜单下选择此功能。

实例：用取样和事件方法获取硬件输入设备的数据

在这个例子中，我们会用 vizjoy 这个模块来添加一个游戏杆然后用游戏杆来控制环境中的一个模型。在这个例子中，我们还会用到 pool 模块，这个模块可以在 demos and examples 这个文件夹中找到。程序开始我们要先导入 viz、math 和 pool 模块，然后启动 pool_object 这个对象。我们还会启动物理引擎，再将观察视角放到正确的位置。

```
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)
```

我们用 vizjoy 添加一个游戏杆。接下来发起一个回调功能来监控游戏杆触发的事件（程序中我们会用到帽子事件，帽子是游戏杆和手柄上的方向控制器）当帽子被触动时，joyhat 这个函数就会被调用，“e”这个参数值被传递。“e.hat”的值就是帽子也就是方向控制器的角度，joyhat 函数用这个角度来旋转对象。运行下段代码。

```
#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)
```

下面我们再添加另外一个回调，它用来监控游戏杆的按键事件。现在当游戏杆的键被按下时，joybutton 这个函数就被调用。同时“e”这个参数就被调用，这个参数传递的是键值也就是哪个键被按下（“e.button”）。试着运行下段代码然后按游戏杆上的键来架竿击球。每个游戏杆的键设置都有所不同，所以你可能需要根据你自己的游戏杆来设置不同按键的功能。

```
#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)
```

实例：游戏杆导航

在下面这个例子中，我们继续编写这个游戏，但不同的是我们要用游戏杆来控制视角。首先我们创建一个游戏杆导航的模块。这个模块很简单里面包含一个函数来运行一个计时器。这个函数从游戏杆获取位置和朝向的数据，然后根据这些数据来设置观察视角。

```
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)
```

保存这个模块为 joystick_nagivator.py，但先别关闭这个文件，也许你还会根据你自己使用的游戏杆的功能更改此模块。现在打开一个新的空白脚本，调用这个模块来继续编写这个游戏。先导入你刚刚创建的 joystick_nagivator.py 模块，运行此脚本。看看这个模块工作。如果发现有问题，你可以修改模块中的语句，确保游戏杆可以正确行使导航功能。

```
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
```

现在我们将主球和视角连接起来，这样我们可以通过视角来看到主球和其他球的关系从而击球。视角的位置和朝向都会被应用到连接的目标上。

```
#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)
```

追踪设备

在虚拟现实系统中追踪设备是一个重要的组成部分。通过追踪用户头部的位置和朝向，我们可以让用户在虚拟世界中行走时看到根据他们的视角渲染出来的世界。追踪设备还可以用来追踪身体的运动，**运动捕捉（motion capture）**设备就具有专门捕捉身体运动的功能。将捕捉到的运动映射到**虚拟人（avatar）**的身体，这样虚拟人就会像现实中的人一样运动。

通过 Vizard 我们可以从多种设备中读取数据并将这些数据应用到虚拟世界中，就像是上面的例子中我们将视角和主球连接起来。

运行 AR demo 看看 **6DOF** 追踪（6 个自由度的追踪—位置的信息有 3 个自由度，朝向的信息有 3 个自由度）如何工作。这个例子是一个**增强现实 augmented reality(AR)**的例子，增强现实就是将虚拟现实和真实世界重叠在一起来呈现。这个例子会用到一个**标记物 marker** 来追踪位置和朝向。这些数据会应用到虚拟人身上，虚拟人会随着这个标记物的运动来移动自己的身体位置和朝向，就好像虚拟人站在标记物上。在这个例子中，你需要一个摄像头。打印出 pattSample1.pdf 文件，该文件在 Teacher in a book demos and examples/art 文件夹下，把打印出来的文件放在固定和平展的地方（如果打印出的图形不平整的话，追踪器是追踪不到的）。

练习

1. 回到“建模物理”那一章的 physics example 的例子。将鼠标触发事件换成游戏杆触发事件。

2. 创建一个类用鼠标来控制一个物体的位置和朝向。

输出设备

我们在本书中用的最多的输出设备为显示器。当然虚拟现实硬件不仅限于二维显示器。**头盔 head-mounted displays (HMDs)**和 **CAVE** 系统会给用户带来沉浸式的视觉体验。对于头盔来说，可以通过分别输出左右眼的图像来为用户呈现立体的视觉。在位置朝向追踪器的帮助下，用户还可以在虚拟世界里自由行走，系统根据用户的位置来渲染相应的三维图像。CAVE 系统通过将三维影像投射到环幕上为用户带来沉浸其中的感觉。CAVE 也可以结合追踪器，用户改变视角时系统渲染出相应的图像。

输出设备不仅仅是视频输出，它也包括制造三维立体声的音响系统。大多数传统立体声系统就可以营造出有深度和方向感的声音。最近一些年，模拟触觉的触觉设备也被整合到虚拟现实系统中，触觉设备可以让用户在触摸虚拟物体的时候有触摸真实物体的感觉。还有一些模拟不同气味的设备。

一些硬件的使用需要特殊的插件，但在 Vizard 库中包含驱动大部分硬件的模块。你可以查看 Vizard 帮助文档中的硬件部分，看看 Vizard 可以驱动哪些硬件。

实例：立体视觉、立体声和触觉输出

在这个例子中，我们要在上面的击球游戏中添加更多的输出方式。好现在我们从游戏杆导航的那部分代码开始。

```
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()
```

现在我们给虚拟世界添加一些声音。一般来讲，添加声音有两种方法。一种方法就是简单的添加一个声音文件。但是这样添加用户只是听到声音而不会产生立体声效果，也不可能定位声音的方向。定位声音源可以从一个特定位置发出，例如从右侧的音响，如果这个位置离你越近声音的效果会越大。首先我们添加一个声音文件为环境添加背景噪音。运行下段代码。

```
#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 )
```

下面我们添加定位声音。我们会用到碰撞事件来处理每次球撞击的事件，当撞击时会有声音出现。注意我们还会用三维声音坐标来播放定位声音。最小最大坐标值设置的是超过最小最大距

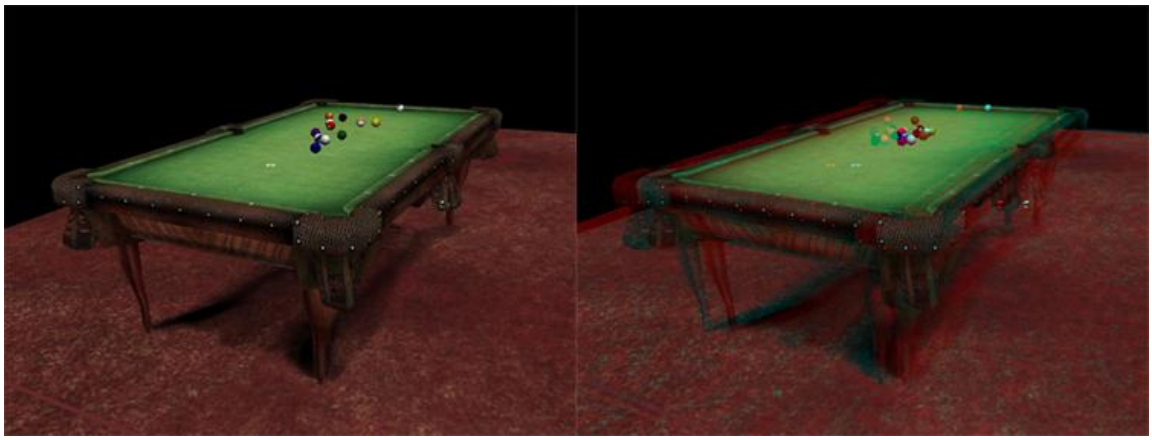
离时声音不会随声音源的远近而变化。（如果你用的不是速度很快的计算机，那么需要注释掉背景噪音那一行，声音的播放也会耗费内存所以播放背景噪音会让你的计算机负荷很大）

```
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 )
```

现在带上耳机运行上面的代码。你可以听到球碰撞的声音。

下面我们添加触觉反馈效果。这部分需要带有力反馈功能的游戏杆。我们会为碰撞事件添加一个反馈。当主球碰到附球时，碰撞任务就会给游戏杆添加一个力反馈然后这个力反馈会在几百毫秒后消失。

```
#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 )
```



最后我们添加一个立体视觉效果，需要立体眼镜(在电影院用到的红蓝眼镜)才会看到这个效果。Vizard 为头盔和立体眼镜渲染立体效果的原理和为其他设备渲染的原理不一样。这里我们用到 viz.go 语句。

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

or

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

练习

1. 回到建模中的场景图那一章，打开“Transformations and the scene graph”例子。为蜜蜂添加一个立体音。注意需要一个循环语句好让蜜蜂不断的产生嗡嗡声。

2. 如果你有头盔或是立体眼镜，试着用 viz.ipd 和 viz.fov 语句来调整瞳孔距离和视角。

网络世界

使用网络功能可以让不同的用户体验同一个虚拟世界。而所有用户可以在这里面进行交流、协同工作、打游戏等等的各种交互行为。为了共享环境，你可以设置一个**主机（master）**来操纵多个**客机（client）**的各种事件同时发生。

使用网络还可以解决各种技术问题。例如，在网络中的所有机器可以渲染同一个场景。所以你可以用多台显示器或投影屏来展示渲染的场景（例如，CAVE 的多个投影墙），每个电脑负责渲染连接到自己的显示设备，但是又与其他电脑保持同步。你可以具体参考 Vizard 文档中的 **vizcave** 和 **clustering** 语句。

还可以利用网络来为多台电脑分配不同的任务来减轻电脑运行的负担。例如一台电脑需要做大量的建模工作，你可以让另外一台电脑只负责建模过程中的简单的场景图转换。（参考下面的例子）。

数据分享

其实用网络的最关键就是让每台电脑发送和接受的数据量最小化。对于每台机器来说，他们有共同渲染的一部分场景作为共同数据，但是每台机器又有属于自己要处理的数据。例如，我们构建一个非常大的花园迷宫里面有鸣叫的小鸟和缓缓飘过的云。我们想让一组用户在这个迷宫行走，实际上是一个手推车带着用户行走，但这个迷宫和它里面的各种物体对每一个用户来说都是一样的。所以我们让每一台机器都渲染同样的场景，但是每一台电脑还需要知道所有手推车的朝向和位置。所以多台电脑之间互相传送的只是每个手推车的位置和朝向信息而已。

运行 network demo，需要知道在你使用的网络中还有哪些电脑。运行这个例子后屏幕上会要求你输入可以连接的另外一台电脑的 IP 地址。成功连接后，会看到你自己屏幕上的手推车的位置和朝向是另外一台电脑的用户的位置和朝向。

实例：网络世界

在这个例子中，我们会让三个用户玩桌球游戏。两个用户使用客机来控制主球。另外一个为主机。因为游戏中的撞球模拟会根据撞球位置有所不同，所以我们可以让主机来做建模物理，然后发送这个模拟后的数据到客机。这样客机不会因为要做这些模拟计算而影响游戏速度。所以我们让主机负责模拟的工作。

我们会为主机和客机编写不同的代码。我们从客机开始。我们会用“Joystick navigation”中的语句来载入游戏到客机（虽然在这里我们还不能实现动态物理模拟）：

```
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] )
```

因为我们要让两个客机的用户看到对方的主球，所以我们添加下面的语句。

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

现在我们添加网络。因为每个客机都要和主机通讯，所以我们为他们建立一个网络。先为主机命名，可以是本地网络名或是 IP 地址。

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

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

下面添加一个函数来发送客机的主球位置给主机。函数会不间断的执行好让主机时刻接受来自客机的数据，这样就知道客机的主球在任意时间的位置了。

```
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 )
```

最后我们建立一个网络时间用来处理从主机传来的数据。参数“e”传递这些数据。主机会传送两种数据，“balls_data”和“cue_pos”。在函数内部这个数据被传递为 e.balls_data 和 e.cue_pos。这些数据会用来模拟桌上所有球的位置。

```
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)
```

现在我们保存这个客机脚本。下面编写主机脚本，同样先载入游戏本身的代码。

```
#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)
```

下面我们为两个客机建立外连的网络。我们为两个客机命名和赋 IP 地址。

```
#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 )
```

然后将两个客机中的主球放在一个字典里。

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

用一个函数来抓取所有球的位置和朝向数据，把这些数据放在一个数组中。

```
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
```

这里通过两个变量来传递数据给两个客机，一个是 cue_pos（对方用户的主球位置）和 balls_data（ball_info 这个函数收集到的所有球的位置和朝向）。

```
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 )
```

最后，我们建立一个网络回调函数用来监控来自客机的数据，用这些数据来放置客机中的主球。

```
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)
```

练习

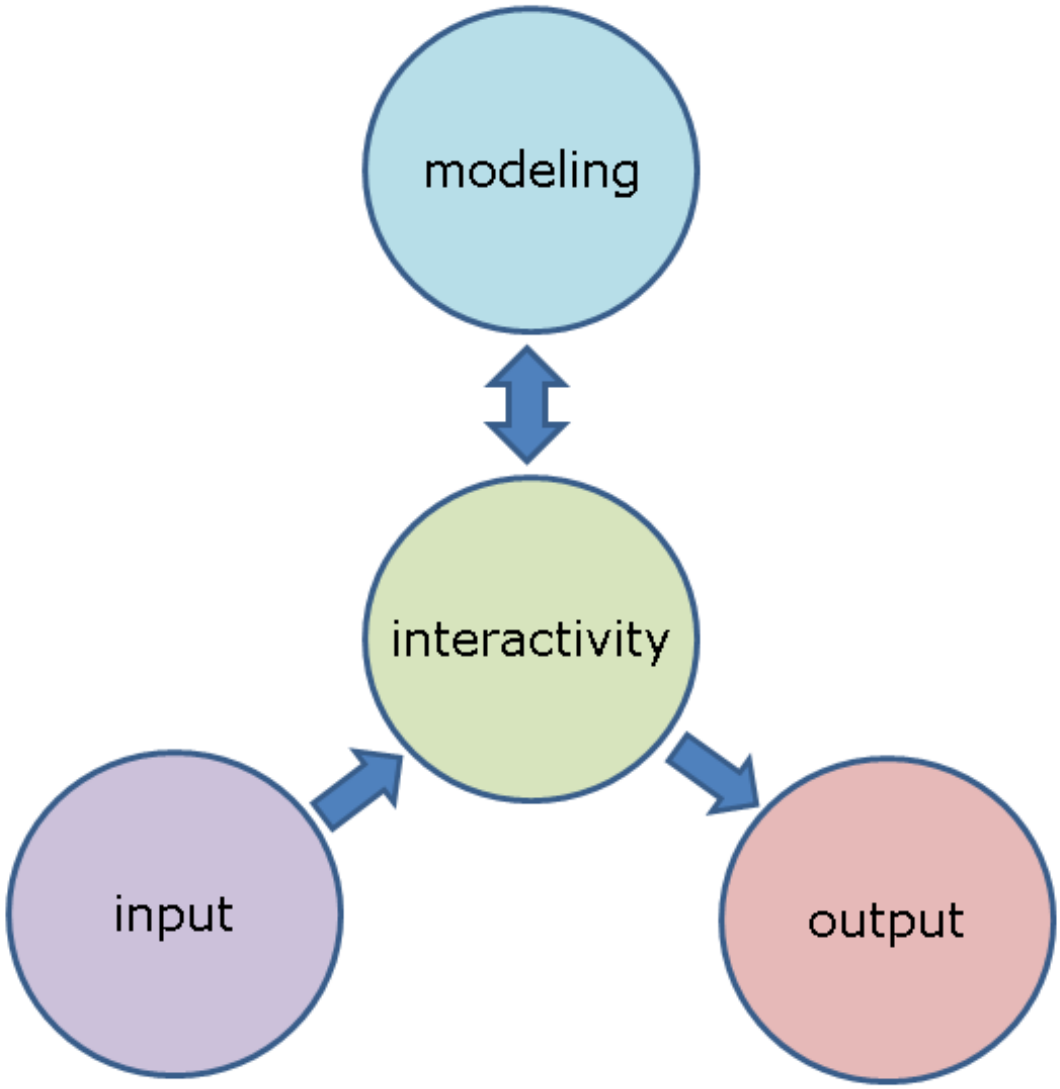
1. 练习编写一个有两个用户的网络世界。添加一个虚拟人用来代表对方的用户。传递视角的朝向和位置数据。用这个数据来模拟虚拟人的行为。

2. 更改上面的桌球游戏，让主机控制球杆而主机和客机都可以看到这个球杆。

构建你自己的虚拟世界

收集资料

虚拟世界可以包含很多内容，所以本部分主要介绍在构建虚拟世界之前的收集资料过程。当你刚开始创建一个虚拟世界时，先收集好所需的一切细节，这会帮助你组织虚拟世界的元素和结构：例如所需的三维物体，所用到的输入设备和输出设备，怎样才能将这些元素放在一起，怎样创建交互的功能等等。



在“Project Plan-EXAMPLE”这个册子中会介绍一个程序的很简单的构建计划，这个文件在 Teacher in a Book demos and examples 文件夹中。运行“program flow example.py”看看这个程序运行的结果。

构建一个虚拟世界的过程实际上分为不同的阶段。而输入、输出和建模过程在这些不同阶段都会有不同的设置。这个小册子的第一页就用一个流程图描述了这些阶段。最左边的部分描述了程序经过的不同阶段。右面的部分描述了这几个阶段中比较特殊的部分。这个流程图实际上可以转化成代码中的任务命令。流程图不仅仅是一个很好的工具而且它涉及的好坏涉及到编写代码的过程。

注意在每个阶段，流程图都描述了这个阶段是如何结束的。代码如何从一个阶段过渡到下一个阶段对于代码的运行效率很重要，它也会决定输入设备传递数据的过程。

构建计划的另外一个部分就是一个表格里面包含你需要的模型，输入输出设备，为交互功能所需的事件和采样。下面的这个例子就包含三维模型和所需的声音文件。当开始构建你自己的虚拟世界时，试着填写这个表格。

程序流程

运用任务命令来控制程序流程

在刚开始构建虚拟世界和结束一个虚拟世界时，有很多工作要做。确定每一件事情按顺序进行，有时候程序对用户动作的反应部分编写起来会很复杂。**tasks** 命令是一个有力的工具。Tasks 任务实际上就是一些函数可以随时暂停和等待一个**情况**的发生。他们处理的过程可以对整个程序的流程没有任何影响。例如我们构建一个虚拟城市里面有很多道路和交通灯。我们想让用户过马路时，他旁边的车流停止行驶让他通过。这里我们就需要一个当用户穿马路时车流暂停和用户过完马路车流继续停止的过程。但是这部分的车流停止的时候其他的车继续运行。所以这里 task 命令就很有用。

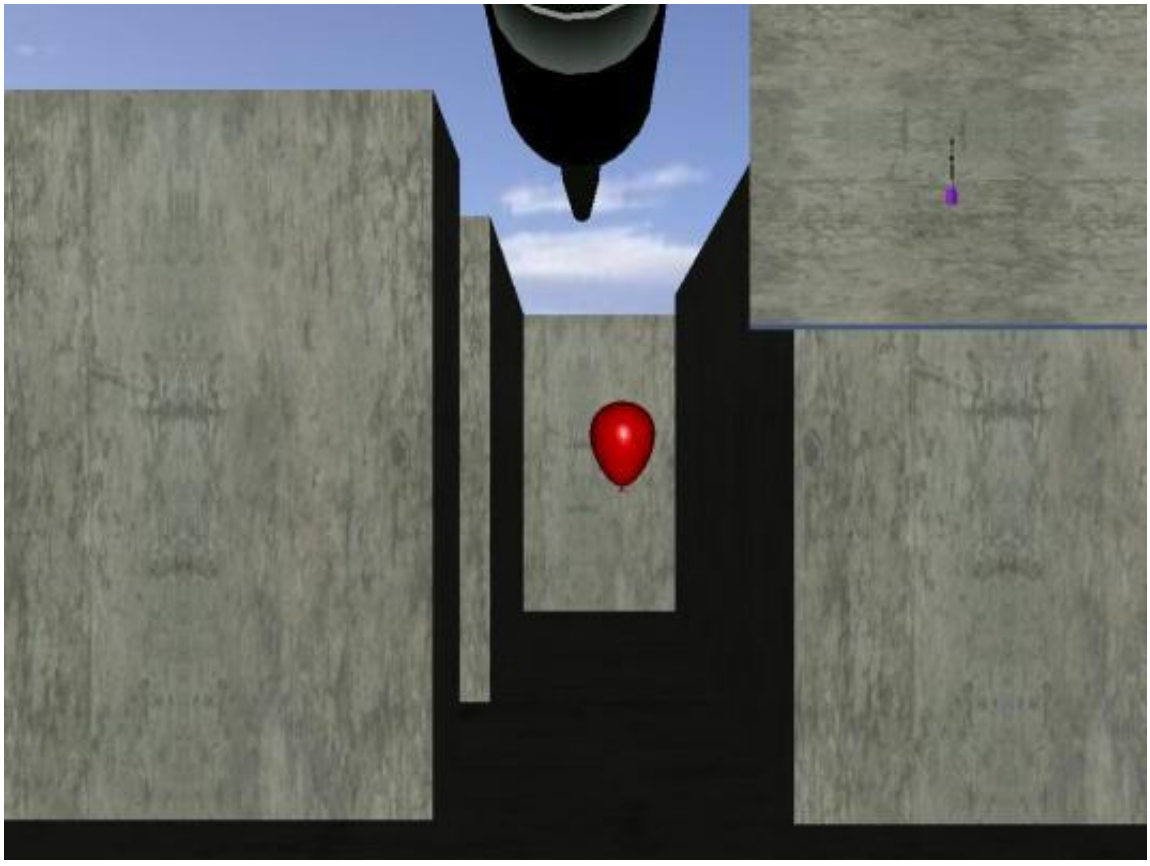
Task 也可以将你的程序分为几个部分来运行。例如你要构建一个虚拟世界让人在这个世界中学习如何焊接的任务。首先，用户需要不断的重复焊接一些材料来练习技能。接下来你想要用户焊接一些新的材料。那么你需要用 task 命令将这个程序分为两部分，首先让所有东西动起来为了让用户练习技能，直到练习结束 task 命令开始运行下一个阶段。就像我们之前所说的根据这个过程可以先画一个流程图然后用 task 命令将这个流程图转化为实际的代码。

当然 task 也可以嵌套我们称之为**分任务(subtasks)**。我们可以让分任务先执行，执行结束后调用总任务。我们可以设置在一些情况下程序暂停和**让行(yield)**其他任务。

下面这个例子中，我们来看看在“Project plan-EXAMPLE”文件中描述的流程图如何转化成一系列的任务。

实例：程序流程

打开“Project plan-EXAMPLE”中的第一个表格用来参照脚本内容。这个例子是一个游戏。当用户准备好后点击一个键程序开始运行。首先屏幕上会显示一些指导语然后游戏开始。游戏过程中，用户会在一个迷宫中行走，迷宫中的一些地方挂有气球。用户要在规定时间内将这些气球击破。游戏结束后，程序会反馈给用户成绩，还会问是否要重新开始游戏。



我们首先添加程序所需的模型：

```
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 )
```

现在我们为程序添加函数。先添加一些基本的任务，然后将这些任务整合到一个大的任务中。第一个函数为虚拟世界设置背景：

```
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( '' )
```

yield 语句让函数成为一个任务。在上段代码中，程序会等待 yield 一个‘s’按键事件的发生，当该事件发生后执行函数。为了执行这个函数任务，你还需要将其列在计划里。添加下段代码将设置背景的函数任务列为计划：

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

暂时将上面的语句用注释符标记掉，添加下面的任务。下面的任务用来呈现游戏指导语，指导与由一些文字组成，文字慢慢显现出来再消失。这里我们用一个 yield 语句添加一个动作并等待这个动作结束。再用一个 yield 语句添加一个计时器事件。

```
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 ))
```

将这个任务列入计划然后看看这个任务是如何运行的。

下面我们再写两个不同的任务用来控制游戏结束的方式。一个任务用来控制游戏进行的时间，另外一个用来监控用户得到的分数。哪个任务先结束就表示用户是否赢得游戏，如果游戏时间先结束那么用户没有在规定时间内完成游戏，如果用户在规定游戏时间之内得到一定的分数那么用户赢得游戏。注意我们让 balloon_popping_task 任务等待一个碰撞（击破气球）事件的发生，之后从这个事件中得到数据（分数）。

```
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 )
```

下面的任务就是游戏本身。这里我们先将两个任务纳入计划中，然后等待其中的任何一个任务执行完毕。viz.Data 对象包含了先执行完毕的任务中的数据，这样我们就知道用户最后是否赢得比赛。

```
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 )
```

现在我们添加另外一个任务，该任务先等待按键事件的发生，按键事件发生后任务继续执行。

```
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)
```

最后我们计划这些任务发生的顺序。这就是我们之前讲过的哪个流程图。注意其中的 while 语句会无限循环。

```
#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() )
```

练习

在 "Teacher in a book demos and tutorials" 目录下，有一个“The viztask challenge setup.py” 程序和一个同名的影音文件。看看你是否可以给这个程序添加一个任务，让程序运行的结果和影片播放的结果一样。

术语表

A

数组 array
一个有序列表。

C

注释 comments
脚本中没有得到执行的语句，语句前加以“#”符号。

常数 constant
整个脚本中值保持不变的量。

D

字典 dictionary
Python 的一种数据结构。字典包含有键和值。可以通过键来取值。

F

帧频 frame rate
每秒钟渲染的帧数。

S

字符串 strings
文字的值.

T

材质 textures
覆盖在三维模型上的图形文件。

V

变量 variable
整个脚本中值会变化的量。

索引

T

V

W

Y