PROGRAM – Uniform Motion

Apparatus

Computer
VPython – www.vpython.org

Goal

The purpose of this activity is to learn how to use VPython to model uniform motion (i.e. motion with a constant velocity).

Introduction

General structure of a program

In general, every program that models the motion of physical objects has two main parts:

1. **Before the loop**: The first part of the program tells the computer to:
   (a) Create 3D objects.
   (b) Give them initial positions and velocities.
   (c) Define numerical values for constants we might need.
2. **The `while` loop**: The second part of the program, the loop, contains the lines that the computer reads to tell it how to update the positions of the objects over and over again, making them move on the screen.

To learn how to model the motion of an object, we will write a program to model the motion of a ball moving with a constant velocity.

Procedure

Before you begin, it will be useful to look back at your notes or a previous program to see how you created a sphere and box.

1. Open a new window in VIDLE.
2. Enter the following two statements in the IDLE editor window.

   ```
   from visual import *
   ```

3. Save this file with a new name like `ball-uniform-motion.py`.
4. Add the line below to create a track that is at the origin and has a length of 3 m, a width of 0.1 m, and a height of 0.05 m. Note that the y-position is -0.075 m (below zero) so that we can place a ball at $y = 0$ such that it appears to be on top of the track..

   ```
   track=box(pos=vector(0,-0.075,0), size=(3,0.05,0.1), color=color.white)
   ```

5. Create a ball (i.e. sphere) at the position $(-1.4, 0, 0)$ m. Choose its radius to be an appropriate size so that the ball appears to be on top of the track.

Figure 1: A ball on a track.

6. Run your program. The ball should appear to be on the top of the track and should be on the left side of the track as shown in Figure 1.

   Now, we will define the velocity of the ball to be to the right with a speed of 0.3 m/s. A unit vector that points to the right is (1,0,0). So, the velocity of the ball can be written on paper as:

$$\begin{aligned} \vec{\mathrm{v}} &= |\vec{\mathrm{v}}|\,\hat{v} \\ &= 0.3 * (1,0,0) \end{aligned}$$

   Next we will see how to write this in VPython.

7. Just as the position of the ball is referenced as `ball.pos` , let's define the ball's velocity as `ball.v` which indicates that v is a property of the object named `ball` . To do all of this, type this line at the end of your program.

```
ball.v=0.3*vector(1,0,0)
```

   This statement creates a property of the ball `ball.v` that is a vector quantity with a magnitude 0.3 that points to the right.

8. Whenever you want to refer to the velocity of the ball, you must refer to `ball.v` . For example, type the following at the end of your program.

```
print(ball.v)
```

9. When you run the program, it will print the velocity of the ball as a 3-D vector as shown below:

```
<0.3, 0, 0>
```

Define values for constants we might need

To make an object move, we will update its position every Δt seconds. In general, Δt should be small enough such that the displacement of the object is small. The size of Δt also affects the speed at which your program runs. If it is exceedingly small, then the computer has to do lots of calculations just to make your object move across your screen. This will slow down the computer.

10. For now, let's use 1 hundredth of a second as the time interval, Δt. At the end of your program, define a variable dt for the time interval.

```
dt=0.01
```

11. Also, let's define the total time t for the clock. The clock starts out at $t = 0$, so type the following line.

```
t=0
```

    That completes the first part of the program which tells the computer to:

    (a) Create the 3D objects.
    (b) Give the ball an initial position and velocity.
    (c) Define variable names for the clock reading t and the time interval dt.

Create a "while" loop to continuously calculate the position of the object.

We will now create a `while` loop. Each time the program runs through this loop, it will do two things:

(a) Calculate the displacement of the ball and add it to the ball's previous position in order to find its new position. This is known as the "position update".

(b) Calculate the total time by incrementing t by an amount dt through each iteration of the loop.

(c) Repeat.

12. For now, let's run the animation for 10.0 s. On a new line, begin the `while` statement. This tells the computer to repeat these instructions as long as $t < 10.0$ s.

```
while t < 10.0:
```

Make sure that the `while` statement ends with `:` because Python uses this to identify the beginning of a loop.

To understand what a while loop does, let's update and then print the clock reading.

13. After the `while` statement, add the following line. Note that it must be indented.

```
    t=t+dt
```

After adding this line, your `while` loop will look like:

```
while t < 10.0:
    t=t+dt
```

Note that this line takes the clock reading t, adds the time step dt, and then assigns the result to the clock reading. Thus, through each pass of the loop, the program updates the clock reading.

14. Print the clock reading by typing the following line at the end of the while loop (again, make sure it's indented) and run your program.

```
    print(t)
```

After adding the `print` statement, your `while` loop will look like:

```
while t < 10.0:
    t=t+dt
    print(t)
```

15. Save and run the program. View the clock readings printed in the console window. After closing your simulation window, you can still view the printed times in the console.

16. You can make it run indefinitely (i.e. without stopping) by saying "while true" and in Python the number 1 is the same as "true" so you can change the `while` statement to read:

```
while 1:
```

Change the `while` statement in your code to be `while 1:` .

Your program should look like:

```
while 1:
    t=t+dt
    print(t)
```

17. Save and run the program. Now, it will print clock readings continually until you close the simulation window.

Stop and reflect on what is going on in this `while` loop. Your understanding of this code is essential for being able to write games.

Just as we updated the clock using `t=t+dt` , we also want to update the object's position. Physics tells us that the object's new position is given by:

$$\text{new position coordinates} \quad = \quad \text{current position coordinates} + \text{velocity} \times \text{time interval}$$

This is called the *position update equation*. It says, "take the current position of the object, add its displacement, and this gives you the new position of the object." In VPython the "=" sign is an *assignment* operator. It takes the result on the right side of the = sign and assigns its value to the variable on the left.

Now we will update the ball's position after each time step `dt` .

18. Inside the `while` loop *before you update the clock*, update the position of the ball by typing:

```
ball.pos=ball.pos+ball.v*dt
```

After typing this line, your `while` loop should look like:

```
while 1:
    ball.pos=ball.pos+ball.v*dt
    t=t+dt
    print(t)
```

19. Change the print statement to print both the clock reading and the position of the ball. Separate the variables by commas as shown:

```
print(t, ball.pos)
```

20. Run your program. You will see the ball move across the screen to the right. Because we have an infinite loop, it'll continue to move to the right. After the ball travels past the edge of the track, the camera will zoom backward to keep all of the objects in the scene.

21. Printing the values of the time and the ball's position slows down the computer. Comment out your print statement by typing the # sign in front of the `print` statement (as in `#print)` .

22. Run your program again and note how fast the ball appears to move.

The computer is calculating the ball's position faster than we can watch it. The t variable in our program is not real-time. Thus we must add a rate statement to slow down the animation.

23. Type the following line just after the `while` statement.

```
rate(100)
```

Your `while` loop should now look like:

```
while 1:
    rate(100)
    ball.pos=ball.pos+ball.v*dt
    t=t+dt
#   print(t)
```

24. Run your program again. You will notice that the animation is much slower. In fact, it will depend on the speed of your computer.

25. Adjust the `rate` statement and try values of 10 or 200, for example. How does increasing or decreasing the argument of the rate function affect the animation?

    The `rate(100)` statement specifies that the while loop will not be executed more than 100 times per second, even if your computer is capable of many more than 100 loops per second. (The way it works is that each time around the loop VPython checks to see whether 1/100 second of real time has elapsed since the previous loop. If not, VPython waits until that much time has gone by. This ensures that there are no more than 100 loops performed in one second.)

Analysis

C Do all of the following.

1. Start with your program from this activity and save it as a different name.
2. Simulate the motion of a ball that starts on the right and travels to the left with a speed of 0.5 m/s. The ball's initial position should be $(1.5, 0, 0)$ m. The `while` loop should run while $t < 5$ s. Print the time and position of the ball.

B Do everything for **C** and the following.

1. Create two balls: Ball A starts on the left side at $(-1.5, 0, 0)$ m and Ball B starts on the right side at $(1.5, 0, 0)$ m. Name them `ballA` and `ballB` in your program.
2. Ball A travels to the right with a speed of 0.3 m/s and Ball B travels to the left with a speed of 0.5 m/s. Define each of their velocities as `ballA.v` and `ballB.v` , respectively.
3. Set the `while` loop to run while $t < 5$ s.
4. Print the clock reading `t` and the position of each ball up to $t = 5$ s.
5. At what clock reading t do they pass through each other?

A Do everything for **B** with the following modifications and additions.

1. Change the width of the track to be 3 m so that the track then appears as a table top.
2. Create three balls that all start at $(x = -1.5, y = 0)$; however, stagger their z-positions so that one travels down the middle of the table, one travels down one edge of the table, and the other travels down the other edge of the table. Name them `ballA` , `ballB` , and, `ballC` , respectively, and give them different colors.
3. Give them x-velocities of (A) 0.25 m/s, (B) 0.5 m/s, and (C) 0.75 m/s.
4. At what time does Ball C reach the end of the table?
5. What are the positions of all three balls when Ball C reaches the end of the table?

PROGRAM – Lists, Loops, and Ifs

Apparatus

Computer
VPython – www.vpython.org

Goal

The purpose of this activity is to learn how to use lists, `for` loops, and `if` statements in VPython.

Introduction

Lists and For Loops

When writing a game, you will typically have multiple objects moving on a screen at one time. As a result, it is convenient to store the objects in a list. Then, you can loop through the list and for each object in the list, update the position of the object.

Procedure

Before you begin, it will be useful to look back at your notes or a previous program to see how you create objects such as spheres and boxes and how you make objects move. These instructions do not repeat the VPython code that you learned in previous activities. Have those chapters and programs available for reference as you do this activity.

1. Open a new window in VIDLE.
2. Enter the following statement in the IDLE editor window.

   ```
   from visual import *
   ```

3. Save this file with a new name like `move-objects.py`.

The `for` loop and the `range()` list

4. Type the `for` loop shown below.

   ```
   for i in range(0,10,1):
       print(i)
   ```

5. Save and run your program. The program should print:

   ```
   >>>
   0
   1
   2
   3
   4
   5
   6
   7
   8
   9
   ```

The statement `range(0,10,1)` creates a list of numbers: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9` . The `for` loop goes through this list, one item at a time, starting with the first item. For each *iteration* through the loop, it executes the code within the loop, but the value of `i` is replaced with the item from the list. Thus the value of `i` will first have the value of 0. Then for the next iteration of the loop, it has the value 1. The loop continues until it has accomplished 10 iterations and `i` has taken on the values of 0 through 9, respectively. Note that the number 10 is not in the list.

6. Change the arguments in the `range(0,10,1)` function. Change 0 to 5, for example. Or change 1 to 2. You can even change the 1 to –1 to see what this does. Run the program each time you change one of the arguments and figure out how each argument affects the resulting list. Write your answers below.

In the function `range(0,10,1)` , how does changing each argument affect the resulting list of numbers? 0: 10: 1:

7. Delete the entire `for` loop for now, and we'll come back to it later.

Lists

When writing games, you may have a lot of moving objects. As a result, it is convenient to store your objects in a list. Then you can loop through your list and move each object or check for collisions, etc.

8. To show how this works, first create 4 balls that are all at $x = -5, z = 0$. However, give them y values that are $y = -3, y = -1, y = 1, y = 3$, respectively. Name them `ball1`, `ball2`, etc. Give them different colors and make their radius something that that looks good on the screen.

9. Run your program to verify that you have four balls at the given locations. The screen should look like Figure 2 but perhaps with a black background and different color balls.

10. Define the balls' velocity vectors such that they will all move to the right but with speeds of 0.5 m/s, 1 m/s, 1.5 m/s, and 2 m/s. Remember that to define a ball's velocity, type:

```
ball1.v=0.5*vector(1,0,0)
```

You'll have to do this for all four balls. Be sure to change the name of the object and speed. You should have four different lines which specify the velocities of the four balls.

Now we will create a list of the four balls. VPython uses the syntax: `[item1, item2, item3,...]` to create a list where item1, item2, etc. are the list items and the square brackets [] denote a list. These items can be integers, strings, or even objects like the balls in this example.

11. To create a list of the four balls, type the following line at the end of your program.

```
ballsList = [ball1, ball2, ball3, ball4]
```

Notice that the names of the items in our list are the names we gave to the four spheres. The name of our list is `ballsList` . We could have called the list any name we wanted.

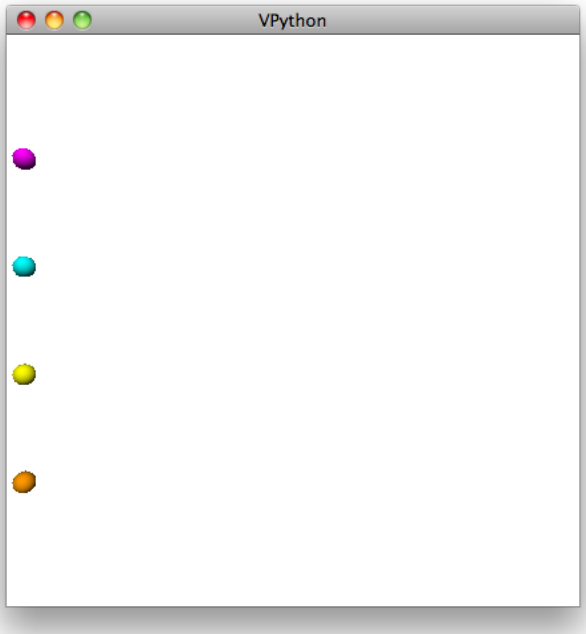

Figure 2: Four balls

Motion

We are going to make the balls move. Remember, there are three basic steps to making the objects move.

- Define variables for the clock and time step.
- Create a `while` loop.
- Update the object's position and update the clock reading.

12. Define variables for the clock and for the time step.

```
t=0
dt=0.01
```

13. Create an infinite `while` loop and use a `rate()` statement to slow down the animation.

```
while 1:
    rate(100)
```

14. We are now ready to update the position of each ball. However instead of updating each ball individually, we will use a `for` loop and our list of balls. Type the following loop to update the position of each ball. Note that it should be indented.

```
for thisball in ballsList:
    thisball.pos=thisball.pos+thisball.v*dt
```

This loop will iterate through the list of balls. It begins with `ball1` and assigns the value of `thisball` to `ball1`. Then, it updates the position of `ball1` using its velocity. On the next iteration, it uses `ball2`. After iterating through all objects in the list, it completes the loop. And at this point it has updated the position of each ball.

15. Now update the clock. Your while loop should ultimately look like the following:

```
while 1:
    rate(100)
    for thisball in ballsList:
        thisball.pos=thisball.pos+thisball.v*dt
    t=t+dt
```

Note that the line `t=t+dt` is indented beneath the `while` statement but is not indented beneath the `for` loop. As a result, the clock is updated upon each iteration in the while loop, not the for loop. The for loop merely iterates through the balls in the ballsList.

Using a `for` loop in this manner saves you from having to write a separate line for each ball. Imagine that if you had something like 20 or 50 balls, this would save you a lot of time writing code to update the position of each ball.

16. Run your program. You should see the four balls move to the right with different speeds.

17. When a ball reaches the right side of the window, the camera will automatically zoom out so that the scene remains in view. In game, we wouldn't want this. Therefore, let's set the size of our window and tell the camera not to zoom. Near the beginning of your program, after the `import` statement, add the following lines:

```
scene.range=5
scene.autoscale=False
```

The range attribute of `scene` sets the right edge of the window at $x = +5$ and the left edge at $x = -5$. The autoscale attribute determines whether the camera automatically zooms to keep the objects in the scene. We set autoscale to false in order to turn it off. Set it to true if you want to turn on autoscaling.

18. Run your program.

IF statements

We are going to keep the balls in the window. As a result, our code must check to see if a ball has left the window. If it has, then reverse the velocity. When you need to check *if* something has happened, then you need an `if` statement.

Let's check the x-position of the ball. If it exceeds the edge of our window, then we will reverse the velocity. If the x-position of a ball is greater than $x = 5$ or is less than $x = -5$, then multiply its velocity by -1. Though we can write this with a single `if` statement, it might make more sense to you if we use the *if-else* statement. The general syntax is:

```
if condition1 :
    indentedStatementBlockForTrueCondition1
elif condition2 :
    indentedStatementBlockForFirstTrueCondition2
elif condition3 :
    indentedStatementBlockForFirstTrueCondition3
elif condition4 :
    indentedStatementBlockForFirstTrueCondition4
else:
    indentedStatementBlockForEachConditionFalse
```

The keyword "elif" is short for "else if". There can be zero or more `elif` parts, and the `else` part is optional.

19. After updating the velocity of each ball inside the `for` loop, add the following `if-elif` statement:

```
        if thisball.pos.x>5:
            thisball.v=-1*thisball.v
        elif thisball.pos.x<-5:
            thisball.v=-1*thisball.v
```

Note that it should be indented inside the `for` loop because you need to check each ball in the list.

After inserting your code, your `while` loop should look like:

```
while 1:
    rate(100)
    for thisball in ballsList:
        thisball.pos=thisball.pos+thisball.v*dt
        if thisball.pos.x>5:
            thisball.v=-1*thisball.v
        elif thisball.pos.x<-5:
            thisball.v=-1*thisball.v
    t=t+dt
```

20. Run your program. You should see each ball reverse direction after reaching the left or right edge of the scene.

Analysis

C Do all of the following.

1. Start with your program from this activity and save it as a different name.
2. When a ball bounces off the right side of the scene, change its color to yellow.
3. When a ball bounces off the left side of the scene, change its color to magenta.

B Do everything for **C** and the following.

1. Create 10 balls that move horizontally and bounce back and forth within the scene. Make the scene 10 units wide and give the balls initial positions of $x = -10$, and $z = 0$, but with y positions that are equally spaced from $y = 0$ to $y = 9$. Give them different initial velocities. Make their radii and colors such that they can be easily seen but do not overlap.

A Do everything for **B** with the following modifications and additions.

1. Start with your program in part (B) with 10 balls that start at the same positions as in part B ($x = -10$ and $z = 0$, with y positions that are equally spaced from $y = 0$ to $y = 9$).
2. Set the initial velocity of each ball to be identical. Give them the same speed, but set their velocities to be in the $-y$ direction.
3. When a ball reaches the bottom of the scene ($y = -10$), change its velocity to be in the $+x$ direction. When a ball reaches the right side of the scene change its velocity to be in the $+y$ direction. When a ball reaches the top of the scene, change its velocity to be in the $-x$ direction. Finally, when it reaches the left side of the scene, change its velocity to be in the $-y$ direction. In this way, make the balls move around the edge of the scene.

4. Run your program. You might find that the balls do not move as you expect. The reason is that if you update a ball's position and it just barely goes out of the scene, then you need to move the ball back within the scene. For example, in the python code below, if the ball's position is updated and it goes past the right edge of the scene at $x = 10$, then the line within the IF statement moves the ball one step backward, back into the scene again. In other words, it reverses the position update statement. (Note the negative sign.)

```
thisball.pos=thisball.pos+thisball.v*dt
if thisball.pos.x>10:
    thisball.pos=thisball.pos-thisball.v*dt
```

   You need to make sure that in each `if` or `elif` statement, you move the ball back to its previous position.

PROGRAM – Keyboard Interactions

Apparatus

Computer
VPython – www.vpython.org

Goal

The purpose of this activity is to incorporate keyboard and mouse interactions into a VPython program.

Procedure

Using the keyboard to set the velocity of an object

1. Open the program from *PROGRAM–Lists, Loops, and Ifs* of the four balls bouncing back and forth within the scene. We will use this program as our starting point. If you did not do this exercise, then the code for the program is shown below.

```
from visual import *

scene.range=5
scene.autoscale=False

ball1=sphere(pos=(-5,3,0), radius=0.2, color=color.magenta)
ball2=sphere(pos=(-5,1,0), radius=0.2, color=color.cyan)
ball3=sphere(pos=(-5,-1,0), radius=0.2, color=color.yellow)
ball4=sphere(pos=(-5,-3,0), radius=0.2, color=color.orange)

ball1.v=0.5*vector(1,0,0)
ball2.v=1*vector(1,0,0)
ball3.v=1.5*vector(1,0,0)
ball4.v=2*vector(1,0,0)

ballsList = [ball1, ball2, ball3, ball4]

t=0
dt=0.01

while 1:
    rate(100)
    for thisball in ballsList:
        thisball.pos=thisball.pos+thisball.v*dt
        if thisball.pos.x>5:
            thisball.v=-1*thisball.v
        elif thisball.pos.x<-5:
            thisball.v=-1*thisball.v
    t=t+dt
```

2. Above your `while` loop, create a box that is at the position $(-4.5, -4.5, 0)$. Name it `shooter` and make its width, length, and height appropriate units so that it looks like it is sitting on the bottom of the window.

3. Run your program and verify that the box is of correct dimensions and is in the left corner of the screen without appearing off screen.

4. Define the velocity of the box to be to the right with a speed of 2 m/s. Name it `shooter.v`.

   Now we want to use the keyboard to make the box move. For right now, we are going to use the following strategy:

   - Look to see if a key is pressed.
   - Check to see which key is pressed.
   - If the right-arrow is pressed, set the velocity of the shooter to be to the right.
   - If the left-arrow is pressed, set the velocity of the shooter to be to the left.
   - If any other key is pressed, set the velocity of the shooter to be zero.
   - Move the box.

5. At the end of your `while` loop, before you update the clock, type the following `if` statement.

```
if scene.kb.keys:
        k = scene.kb.getkey()
        if k == "right":
            shooter.v=2*vector(1,0,0)
        elif k == "left":
            shooter.v=2*vector(-1,0,0)
        else:
            shooter.v=vector(0,0,0)
shooter.pos = shooter.pos + shooter.v*dt
```

6. Run your program and press the right arrow key, left arrow key, or any other key in order to see how it works.

7. Now study the `if` statement and understand what each line does:

```
if scene.kb.keys:
```

   The list `scene.kb.keys` is a list of keys that have been pressed on the keyboard. The `if` statement checks to see whether this list exists because it only exists if at least one key has been pressed. Every time you press a key, the keystroke is appended to the end of this list.

```
k = scene.kb.getkey()
```

   The function `scene.kb.getkey()` will get the last key that was pressed and will remove it from the end of the list. In this case, this keystroke is assigned to the variable k. The following `if-elif-else` statement checks the value of k to see whether it was the left arrow key or right arrow key. For the left arrow key, the velocity is defined to the left. For the right arrow key, the velocity is defined to the right. For any other key (`else`), the velocity is set to zero.

Using the keyboard to create a moving object

We are now going to use the keyboard to launch bullets from our shooter. We need another list where we can store the bullets. Before the `while` loop, create an empty list called `bulletsList`.

```
bulletsList=[]
```

8. In your `if` statement where you check for keyboard events, add the following `elif` statement.

```
        elif k=="␣":
            bullet=sphere(pos=shooter.pos, radius=0.1, color=color.
                white)
            bullet.v=3*vector(0,1,0)
            bulletsList.append(bullet)
```

Study this section of code and know what each line does. If you press the spacebar, a white sphere is created at the position of the shooter. Its name is assigned to be `bullet`. Then, its velocity is set to be in the $+y$ direction with a speed of 3 m/s. Finally, and this is really important, the bullet is added (i.e. appended) to the end of the `bulletsList`. This is so that we can later update the positions of all of the bullets in this list.

9. In your `while` statement before you update the clock, add a `for` loop that updates the positions of the bullets.

```
for thisbullet in bulletsList:
    thisbullet.pos=thisbullet.pos+thisbullet.v*dt
```

Your final `while` loop should look like this:

```
while 1:
    rate(100)
    for thisball in ballsList:
        thisball.pos=thisball.pos+thisball.v*dt
        if thisball.pos.x>5:
            thisball.v=-1*thisball.v
        elif thisball.pos.x<-5:
            thisball.v=-1*thisball.v

    if scene.kb.keys:
            k = scene.kb.getkey()
            if k == "right":
                shooter.v=2*vector(1,0,0)
            elif k == "left":
                shooter.v=2*vector(-1,0,0)
            elif k==" ":
                bullet=sphere(pos=shooter.pos, radius=0.1, color=color.
                    white)
                bullet.v=3*vector(0,1,0)
                bulletsList.append(bullet)
            else:
                shooter.v=vector(0,0,0)
    shooter.pos = shooter.pos + shooter.v*dt

    for thisbullet in bulletsList:
        thisbullet.pos=thisbullet.pos+thisbullet.v*dt

    t=t+dt
```

You should study this code and know what each line means.

10. Run your program.

Analysis

C Do all of the following.

1. Assign variables to the speed of the shooter and the speed of the bullet.
2. Replace all instances of "2" for the shooter with the variable for the speed of the shooter.
3. Replace all instances of "3" with the variable for the speed of the bullet of the bullet.

B Do everything for **C** and the following.

1. Fire the bullets from the center of the top plane of the box instead of its center.
2. Add a counter called `shots` and set shots=0 before your `while` loop. Update the value of `shots` and print the value of `shots` every time a bullet is fired.
3. Check to see if the up arrow key is pressed or the down arrow key is pressed. If one of these keys is pressed, set the velocity of the shooter to be up or down, respectively.

A Do everything for **B** with the following modifications and additions.

1. Add additional keystrokes that will fire a bullet to the left, to the right, or downward.
2. Suppose that the shooter only has 10 bullets. When the shooter reaches a maximum of 10 bullets, hitting the spacebar will no longer file a bullet.
3. Create a keystroke that will replenish the shooter, meaning that after hitting this keystroke, you can fire 10 more bullets.

PROGRAM – Collision Detection

Apparatus

Computer
VPython – www.vpython.org

Goal

The purpose of this activity is to detect collisions between moving objects. You will learn to create a function, and you will learn about boolean variables that are either `True` or `False` .

Introduction

The idea of collision detection is a fairly simple one: *check to see if two objects overlap.* If their boundaries overlap, then the objects have collided.

Distance between spheres

Suppose that two spheres have radii R_1 and R_2, respectively. Define the center-to-center distance between the two spheres as d. As shown in Figure 3:

if $d > (R_1 + R_2)$ the spheres do not overlap.

if $d < (R_1 + R_2)$ the spheres overlap.

if $d = (R_1 + R_2)$ the spheres exactly touch. Note that this will never happen in a computer game because calculations of the positions of the spheres result in 16-digit numbers (or more) that will never be exactly the same.

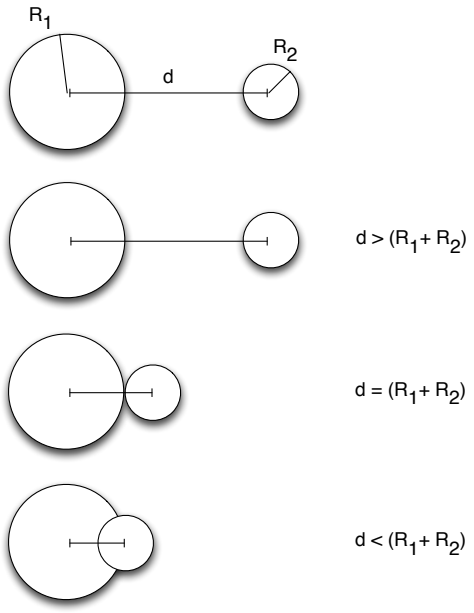

Figure 3: Condition for whether two spheres collide.

If the spheres are at coordinates (x_1, y_1, z_1) and (x_2, y_2, z_2), then the distance between the spheres is:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

This is the magnitude of a vector that points from one sphere to the other sphere, as shown in Figure 4.

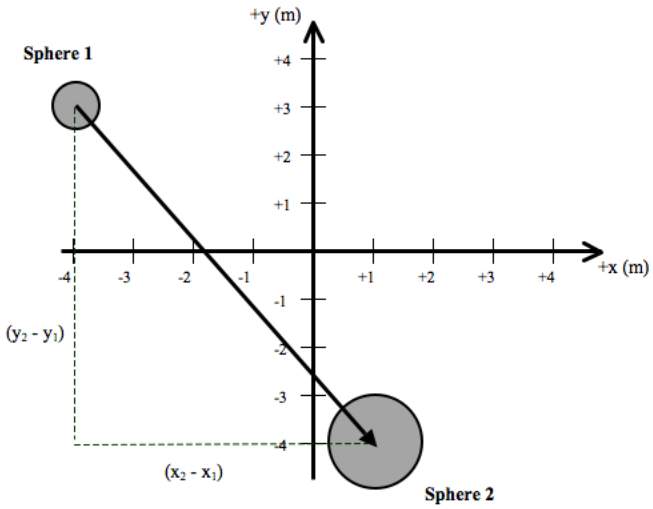

Figure 4: Distance between two spheres.

Because we only want the magnitude of the vector from one sphere to the other, it does not matter which sphere you call Sphere 1. Thus, you can just as easily calculate the distance using:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

Because you square the vector's components, the sum of the squares of the components will always be positive.

Exercises

Ball1 is at $(-3, 2, 0)$ m and has a radius of 0.05 m. Ball2 is at $(1, -5, 0)$ m and has a radius of 0.1 m. What is the distance between them?
Ball1 is at $(1, 2, 0)$ m and has a radius of 0.05 m. Ball2 is at $(1.08, 1.88, 0)$ m and has a radius of 0.1 m. What is the distance between them? At this instant, have the balls collided?

Procedure

Starting program

1. Begin with the program that you wrote in *Chapter 9 PROGRAM – Keyboard Interactions*. It should have a shooter (that moves horizontally and shoots missiles) and four balls that move horizontally and bounce back and forth within the window.

   If you do not have that program, type the one shown below.

```
from visual import *

scene.range=5
scene.autoscale=False

ball1=sphere(pos=(-5,3,0), radius=0.2, color=color.magenta)
ball2=sphere(pos=(-5,1,0), radius=0.2, color=color.cyan)
ball3=sphere(pos=(-5,-1,0), radius=0.2, color=color.yellow)
ball4=sphere(pos=(-5,-3,0), radius=0.2, color=color.orange)

ball1.v=0.5*vector(1,0,0)
ball2.v=1*vector(1,0,0)
ball3.v=1.5*vector(1,0,0)
ball4.v=2*vector(1,0,0)

ballsList = [ball1, ball2, ball3, ball4]

shooter=box(pos=(-4.5,-4.5,0), width=1, height=1, length=1, color=color.
    red)
shooter.v=2*vector(1,0,0)

bulletsList=[]

t=0
dt=0.01

while 1:
    rate(100)
    for thisball in ballsList:
        thisball.pos=thisball.pos+thisball.v*dt
        if thisball.pos.x>5:
            thisball.v=-1*thisball.v
        elif thisball.pos.x<-5:
            thisball.v=-1*thisball.v

    if scene.kb.keys:
            k = scene.kb.getkey()
            if k == "right":
                shooter.v=2*vector(1,0,0)
            elif k == "left":
                shooter.v=2*vector(-1,0,0)
            elif k==" ":
                bullet=sphere(pos=shooter.pos, radius=0.1, color=color.
                    white)
                bullet.v=3*vector(0,1,0)
                bulletsList.append(bullet)
```

```
        else:
            shooter.v=vector(0,0,0)
    shooter.pos = shooter.pos + shooter.v*dt

    for thisbullet in bulletsList:
        thisbullet.pos=thisbullet.pos+thisbullet.v*dt

    t=t+dt
```

Defining a function

When you have to do a repetitive task, like check whether each missile collides with a ball, it is convenient to define a function. This section will teach you how to write a function, and then we will write a custom function to check for a collision between two spheres.

A function has a *signature* and a *block*. In the signature, you begin with `def` and an *optional parameter list*. In the block, you type the code that will be executed when the function is called.

2. To see how a function works, type the following code near the top of your program after `the import` statement.

```
def printDistance(object1, object2):
    distance=mag(object1.pos-object2.pos)
    print(distance)
```

This function accepts two parameters named `object1` and `object2`. It then calculates the distance between the objects by finding the magnitude of the difference in the positions of the objects. (Note that `mag()` is also a function. It calculates the magnitude of a vector.) Then, it prints the distance to the console.

3. At the end of the `while` loop, call your function to print the distance between a ball and the shooter by typing this line. Now each iteration through the loop, it will print the distance between the shooter and `ball1`.

```
printDistance(shooter,ball1)
```

4. Run the program. You will notice that it prints the distance between the shooter and `ball1` after each timestep.

5. Change the code to print the distance between `ball1` and `ball4` and run your program.

Note that you didn't have to reprogram the function. You just changed the parameters sent to the function. This is what makes functions such a valuable programming tool.

Many functions return a value or object. For example, the `mag()` function returns the value obtained by calculating the square root of the sum of the squares of the components of a vector. This way, you can write `distance=mag(object1.pos-object2.pos)` , and the variable `distance` will be assigned the value obtained by finding the magnitude of the given vector. To return a value, the function must have a `return` statement.

6. You can delete the `printDistance` function and the `printDistance` statement because will not use them in the rest of our program.

7. Near the top of your program, after the `import` statement, write the following function. It determines whether two spheres collide or not.

```
def collisionSpheres(sphere1, sphere2):
    dist=mag(sphere1.pos-sphere2.pos)
    if(dist<sphere1.radius+sphere2.radius):
        return True
    else:
        return False
```

Study the logic of this function. Its parameters are two spheres, so when you call the function, you have to give it to spheres. It then calculates the distance between the spheres. If this distance is less than the sum of the radii of the spheres, the function returns `True`, meaning that the spheres indeed collided. Otherwise, it returns `False`, meaning that the spheres did not collide.

This function will only work for two spheres because we are comparing the distance between them to the sum of their radii. Detecting collisions between boxes and spheres will come later.

8. Inside the `for` loop that updates the position of the bullet, add the following lines:

```
        for thisball in ballsList:
            if collisionSpheres(thisbullet, thisball):
                thisball.pos=vector(0,-10,0)
                thisball.v=vector(0,0,0)
```

After adding these lines, the bullet `for` loop will look like this:

```
    for thisbullet in bulletsList:
        thisbullet.pos=thisbullet.pos+thisbullet.v*dt
        for thisball in ballsList:
            if collisionSpheres(thisbullet, thisball):
                thisball.pos=vector(0,-10,0)
                thisball.v=vector(0,0,0)
```

For each bullet in the bulletsList, the program updates the position of the given bullet and then loops through each ball in the ballsList. For each ball, the program checks to see if the given bullet collides with the given ball. If they collide, then it sets the position of the ball to be below the scene at $y = -10$, and it sets the velocity of the ball to be zero. If they do not collide, nothing happens because there is no `else` statement.

9. Run your program. You will notice that when a bullet hits a ball, the ball disappears from the scene. Note that it is technically still there, and the computer is still calculating its position with each time step. It is simply not in the scene, and its velocity is zero.

Analysis

We now have the tools to make a game. In a future chapter you will have the freedom to create a game of your choice based on what we've learned. However, in these exercises, you will merely add functionality to this program to make it a more interesting game.

C Do all of the following.

1. If a missile exits the scene (i.e. `missile.pos.y > 5`), set its velocity to zero.
2. Create a variable called `hits` and add one to this variable every time a missile hits a sphere.
3. Print `hits` every time a missile hits a ball.

B Do everything for **C** and the following.

1. Make 10 balls that move back and forth on the screen and set their y-positions to be greater than $y = 0$ so that they are all on the top half of the screen.
2. Add a variable called `shots` and increment this variable every time a missile is fired.

A Do everything for **B** with the following modifications and additions.

1. The score should not be simply based on whether a missile hits a ball, but it should also be based on how many missiles are needed. For example, if you hit all four balls with only four missiles shot, then you should get a higher score. Also, if you hit all four balls with only four missiles shot in only 1 s, then you should get a higher score than if it required 10 s. Design a scoring system based on missiles fired, hits, and time. Write your scoring system below.

2. Program your scoring system into the code. Use a variable `points` for the total points. Use `print()` statements every time you hit a missile to print t, `shots` , `hits` , and `points` .
3. After all balls are hit, use the `break` statement to break out of the while loop and close the program.
4. After you are confident that it is working, write down your top 5 scores.

5. Ask three friends to play the game one or more times and write down the top score by each friend.