Physics for Video Games

Notes and VPython Programs Only

Aaron Titus

High Point University

Spring 2013

http://physics.highpoint.edu/∼atitus/

Contents

1 Coordinates

Cartesian Coordinate System

To specify the location of an object, we use a coordinate system. The one shown in Figure 1.1 is a two-dimensional (2-D) Cartesian coordinate system with the $+x$ direction defined to the right and the $+y$ direction defined to be upward, toward the top of the page.

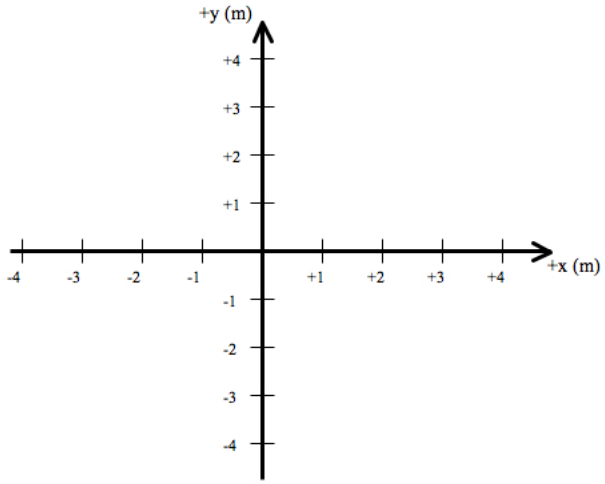


Figure 1.1: A 2-D Cartesian coordinate system.

A three-dimensional (3-D) coordinate system with the $+z$ axis defined to be outward toward you, perpendicular to the page, is shown in Figure 1.2.

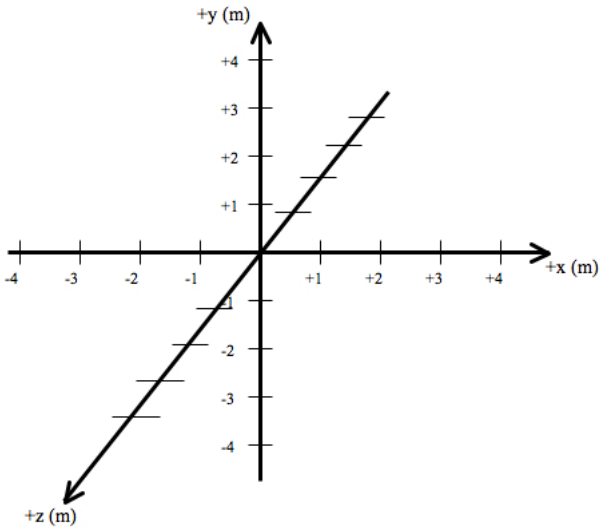


Figure 1.2: A 3-D Cartesian coordinate system.

A coordinate system is defined by:

1. an origin
2. a scale (determined by the tick marks, numbers, and units)
3. an orientation; the direction of the +x, +y, and +z directions, respectively.

The orientation is described verbally by saying something like "The +x direction is toward the right of the origin." Or, "The +y direction is upward toward the top of the page."

Coordinates

A point on a Cartesian coordinate system is designated by a pair of numbers in 2-D and a triplet of numbers in 3-D. On a 2-D coordinate system, the pair of numbers represents the (x, y) coordinates of the point.

On the coordinate system in Figure 1.3, the red dot is at the location $(+1, +3)$ meters. And the blue dot is at the location $(+4, -4)$ meters. Thus, we say that the x-position of the red dot is +1 m, and the y-position of the red dot is +3 m. Likewise, the x-position of the blue dot is +4 m, and the y-position of the blue dot is -4 m.

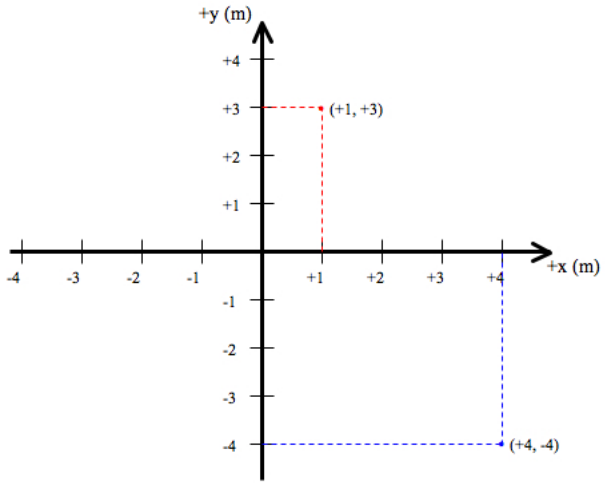


Figure 1.3: Coordinates of certain points on a coordinate system.

The sign of the coordinate tells us the side of the origin that the point is on. Thus, $x = +1$ means that the point is on the right-side of the origin. If $x = -1$ then the point is on the left-side of the origin.

Likewise, $y = +3$ means that the point is above the origin, and $y = -4$ means that the point is below the origin.

Question: State in words the location of a point with a positive z-coordinate.
Answer: Using the coordinate system in Figure 1.2, a positive value of z means that the point is in front of the page (which we assume to be the x-y plane), toward you. For example if you are reading this page, then your eyes have positive z coordinates.

Computer Convention

Programming languages define the origin of the monitor to be at the top left corner. The $+x$ axis is to the right, and the $+y$ axis is downward, as shown in Figure 1.4.

The units, in computer graphics, are pixels. If the resolution of a monitor is is 1440×900, it means that the monitor displays 1440 pixels horizontally and 900 pixels vertically. Since the origin is in the top left corner, the coordinates of a single pixel on a monitor are always positive.

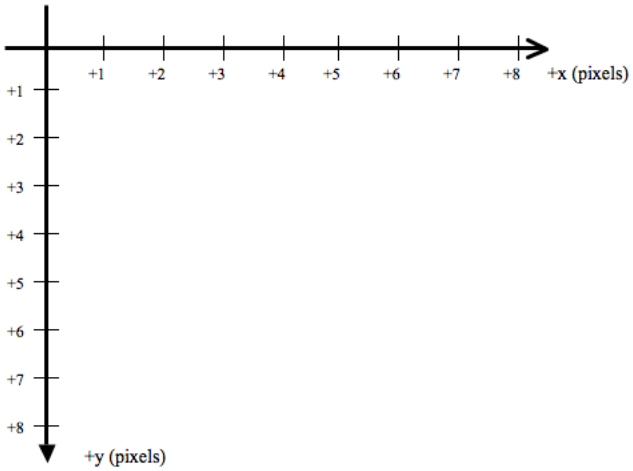


Figure 1.4: Convention for pixel coordinates on a computer monitor.

Example

Question: What are the coordinates of point A in Figure 1.5?
Answer: For point A, $x = +2$ m and $y = +3$ m. Thus, it's coordinates are $(+2, +3)$ m.

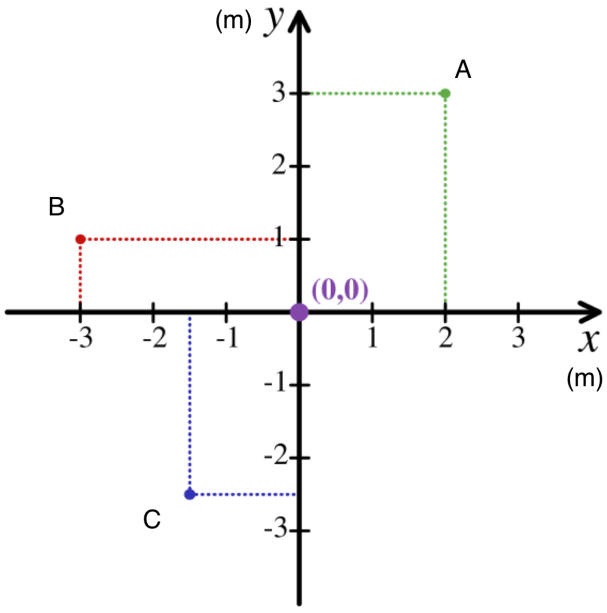


Figure 1.5: Three points on a Cartesian coordinate system.

Homework

1. What are the coordinates of points B and C in Figure 1.5?

2. A puck travels across the monitor in a computer game as shown in Figure 1.6. The top left corner of the image is the origin. Each line on the grid represents 10 pixels. The puck moves from the top, right side of the monitor to the bottom, left side of the monitor. What are the coordinates in pixels of each image of the puck?

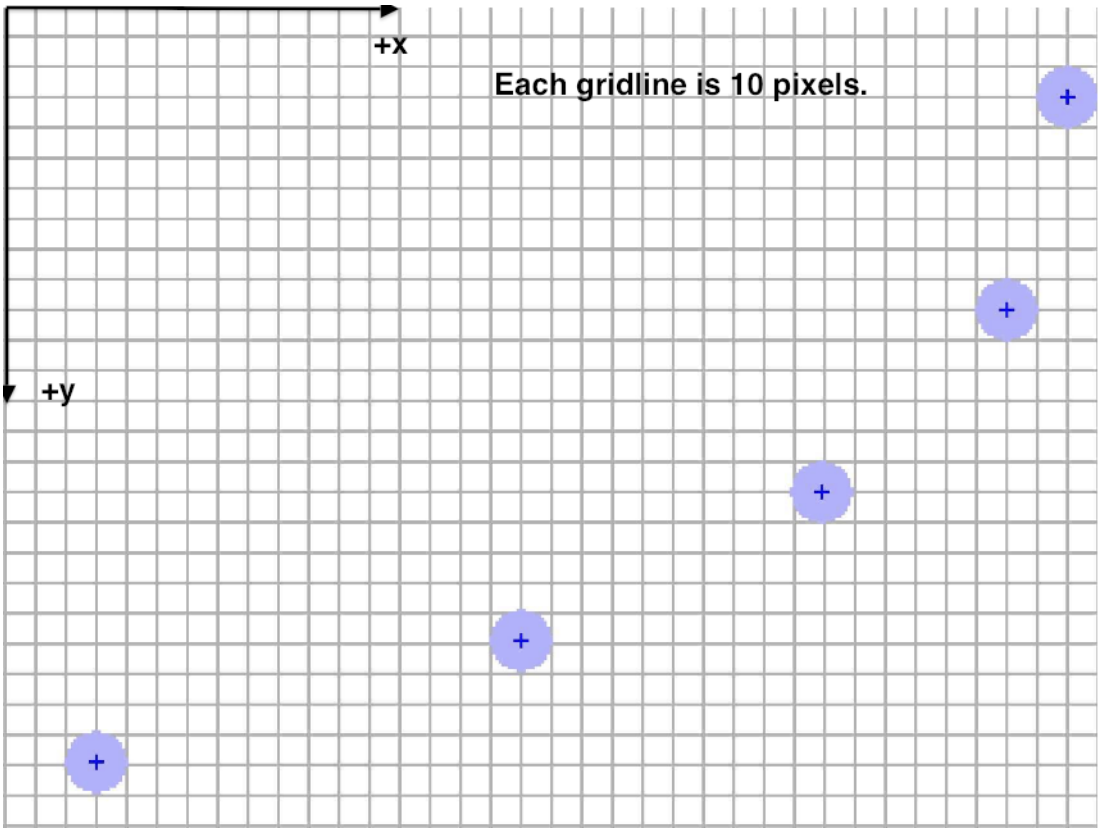


Figure 1.6: A puck on a computer monitor.

2 PROGRAM: Introduction to VPython

Apparatus

Computer
VPython – www.vpython.org

Goal

The purpose of this experiment is to write your first program in a language called Python. We will use a package Visual that gives Python the capability of doing vector algebra and creating 3D objects in a 3D scene. The use of Python with the Visual package is called VPython. The capability of 3D graphics with vector mathematics makes it a great tool for simulating physics phenomena. In this lab, you will learn:

- how to use VIDLE, the integrated development editor for VPython.
- how to structure a simple computer program in VPython.
- how to create 3D objects such as spheres and arrows.

Setup

If not already installed, be sure to install Python and Visual on your computer. It is available for Mac, Linux, and Windows. Be sure to follow the download instructions at http://www.vpython.org.

Procedure

Opening the editor VIDLE

1. After installation of both Python and Visual, find the application "IDLE" or maybe "VIDLE" and open it. This is a text editor where you will write and run your programs.

Starting a program: Setup statements

2. Enter the following two statements in the IDLE editor window.

```
from __future__ import division
from visual import *
```

   Every VPython program begins with these setup statements.

   The first statement (`from space underscore underscore future underscore underscore space import space *`) tells the Python language to treat 1/2 as 0.5. Without the first statement, the Python programming language does integer division with truncation and 1/2 is zero!

   The second statement tells the program to use the package "visual" which will give Python the capability to handle 3D graphics and vector arithmetic.

3. Save the program. In the VIDLE editor, from the `File` menu, select `Save`. Browse to a location where you can save the file, and give it the name "vectors.py". YOU MUST TYPE the ".py" file extension because VIDLE will NOT automatically add it. Without the ".py" file extension VIDLE won't display statements in a different font color.

Creating an object

4. Now tell VPython to make a sphere. On the next line, type:

   ```
   sphere()
   ```

   This statement tells the computer to create a sphere object.

5. Run the program by pressing F5 on the keyboard. Two new windows appear in addition to the editing window. One of them is the 3-D graphics window, which now contains a sphere.

   ### The 3-D graphics scene

   By default the sphere is at the center of the scene, and the "camera" (that is, your point of view) is looking directly at the center.

6. Hold down both mouse buttons and move the mouse forward and backward to make the camera move closer or farther away from the center of the scene. (On a Mac, hold down the option key while moving the mouse forward and backward.)

7. Hold down the right mouse button alone and move the mouse to make the camera "revolve" around the scene, while always looking at the center. (On a Mac, in order to rotate the view, hold down the Command key while you click and drag the mouse.)

   When you first run the program, the coordinate system has the positive x direction to the right, the positive y direction pointing up, toward the top edge of the monitor, and the positive z direction coming out of the screen toward you. You can then rotate the camera view to make these axes point in other directions.

   ### The Python Shell window is important – Error messages appear here

   IMPORTANT: Arrange the windows on your screen so the Shell window is always visible. DO NOT CLOSE THE SHELL WINDOW. KILL the program by closing only the graphic display window.

   The second new window that opened when you ran the program is the Python Shell window. If you include lines in the program that tell the computer to print text, the text will appear in this window.

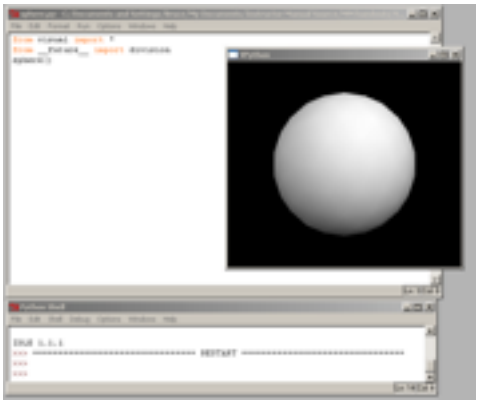


Figure 2.1: Arrange the IDLE windows so that you can see the Shell.

8. Use the mouse to make the Python Shell window smaller, and move it to the lower part of the screen as shown below. Keep it open when you are writing and running programs so you can easily spot error messages, which appear in this window.

9. Make your edit window small enough that you can see both the edit window and the Python Shell window at all times. Do not expand the edit window to fill the whole screen. You will lose important information if you do!

10. To kill the program, close only the graphics window. Do not close the Python Shell window.

Error messages: Making and fixing an error

To see an example of an error message, let's try making a spelling mistake.

11. Change the third statement of the program to the following:

    `phere()`

12. Run the program.

    There is no function or object in VPython called phere(). As a result, you get an error message in red text in the Python Shell window. The message gives the filename, the line where the error occurred, and a description of the error. For example:

```
Traceback (most recent call last):
  File "/Users/atitus/Documents/courses/PHY221/presentations/chapter-01/lab/
vpython/vectors.py", line 4, in <module>
    phere()
NameError: name 'phere' is not defined
```

    Read error messages from the bottom up. The bottom line contains the information about the location of the error.

13. Correct the error in the program. Whenever your program fails to run properly, look for a red error message in the Python Shell window.

    Even if you don't understand the error message, it is important to be able to see it, in order to find out that there is an error in your code. This helps you distinguish between a typing or coding mistake, and a program that runs correctly but does something other than what you intended.

Changing "attributes" (position, size, color, shape, etc.) of an object

Now let's give the sphere a different position in space and a radius.

14. Change the last line of the program to the following:

    sphere(pos=(-5,2,3), radius=0.40, color=color.red)

15. Run the program. Experiment with other changes to pos, radius, and color, running the program each time you change something.

16. Find answers to the following questions:

QUESTIONS TO ANSWER ABOUT SPHERES: What does changing the pos attribute of a sphere do? What does changing the radius attribute of a sphere do? What does changing the color attribute of a sphere do? What colors can you use? (Note: you can try color=(1,0.5,0) for example. The numbers stand for RGB (Red, Green, Blue) and can have values between 0 and 1. Can you make an purple sphere? Note that colors such as cyan, yellow, and magenta are defined, but not all possible colors are defined. Choose random numbers between 0 and 1 for the (Red, Green, Blue) and see what you get.

Autoscaling and units

VPython automatically zooms the camera in or out so that all objects appear in the window. Because of this autoscaling, the numbers for the pos and radius can be in any consistent set of units, like meters, centimeters, inches, etc. For example, this could represent a sphere with a radius 0.20 m at the position (2,4,0) m. In this course we will often use SI units in our programs ("Systeme International", the system of units based on meters, kilograms, and seconds).

Creating a box object

Another object that we will often create is a box. A box is defined by its position, axis, length, width, and height as shown in Figure 2.2.

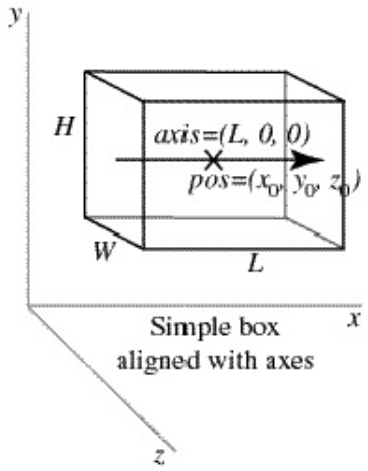


Figure 2.2: Attributes of a box.

17. Type the following on a new line, then run the program:

```
box(pos=(0,0,0), length=2, height=1, width=0.5, color=color.orange)
```

18. Change the length to 4 and rerun the program.

19. Now change its height and rerun the program.

20. Similarly change its width and position.

QUESTIONS TO ANSWER ABOUT BOXES:
Which dimension (length, width, or height) should be changed to make a box that is longer along the y-axis?

What point does the position refer to: the center of the box, one of its corners, the center of one of its faces, or some other point?

Comment lines (lines ignored by the computer)

Comment lines start with a # (pound sign). A comment line can be a note to yourself, such as:

```
# units are meters
```

Or a comment can be used to remove a line of code temporarily, without erasing it.

21. Put a # at the beginning of the line creating the box, as shown below.

```
#box(pos=(0,0,0), length=2, height=1, width=0.5, color=color.orange)
```

22. Run the program. What did you observe?

23. Uncomment that line by deleting the # and run the program again. The box now appears.

Naming objects; Using object names and attributes

We will draw a tennis court and will change the position of a tennis ball.

24. Clean up your program so it contains only the following objects:

    A green box that represents a tennis court. Make it 78 ft long, 36 ft wide, and 4 ft tall. Place its center at the origin.

    An orange sphere (representing a tennis ball) at location $\langle -28, 5, 8 \rangle$ ft, with radius 1 ft. Of course a tennis ball is much smaller than this in real life, but we have to make it big enough to see it clearly in the scene. Sometimes we use unphysical sizes just to make the scene pretty.

    (Remember, that you don't type the units into your program. But rather, you should use a consistent set of units and know what they are.)

25. Run your program and verify that it looks as expected. Use your mouse to rotate the scene so that you can see the ball relative to the court.

26. Change the initial position of the tennis ball to $\langle 0, 6, 0 \rangle$ ft.

27. Run the program.

28. Sometimes we want to change the position of the ball after we've defined it. Thus, give a name to the sphere by changing the `sphere` statement in the program to the following:

```
tennisball = sphere(pos=(0,6,0), radius=1, color=color.orange)
```

    We've now given a name to the sphere. We can use this name later in the program to refer to the sphere. Furthermore, we can specifically refer to the attributes of the sphere by writing, for example, `tennisball.pos` to refer to the tennis ball's position attribute, or `tennisball.color` to refer to the tennis ball's color attribute. To see how this works, do the following exercise.

29. Start a new line at the end of your program and type:

```
print(tennisball.pos)
```

    If you are using an older version of VPython, you may have to print the position using the syntax:

```
print "tennisball.pos"
```

30. Run the program.

31. Look at the text output window. The printed vector should be the same as the tennis ball's position.

32. Add a new line to the end of your program and type:

```
tennisball.pos=vector(32,7,-12)
```

    When running the program, the ball is first drawn at the original position but is then drawn at the last position. Note that whenever you set the position of the tennis ball to a new value in your program, the tennis ball will be drawn at that position. You may have notice a very quick flash after you first run your program, showing the ball drawn at the first position and then redrawn at the new position.

33. Add a new line to the end of your program and type:

```
print(tennisball.pos)
```

(Or just copy and paste your previous print statement.)

34. Run your program. It now draws the ball, prints its position, redraws the ball at a new position, and prints its position again. As a result, you should see the following two lines printed:

```
<0, 6, 0>
<32, 7, -12>
```

Of course, this happens faster than your eye can see it which is why printing the values is so useful.

Analysis

We are going to begin writing a program that will later become the game Missile Command, and we will use what we learned in this tutorial.

C Do all of the following.

1. Create a new blank file and name it *missile-command.py*.
2. Create a green box that is at the location (0,-950,0), has a length=2000, height=100, and width=50 units.
3. Create a red sphere at (-500,950,0) with a radius of 50.
4. Create a magenta box at (750,-850,0) with a length=100, height=100, and width=50 units.

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

1. Name the sphere `source1` .
2. Name the box `city1` .

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

1. Create a total of 3 red spheres that are equally spaced across the top of the window. Remember that the green box has a length of 2000 which determines the width of the window. Name the spheres from left to right: `source1` , `source2` , and `source3` .
2. Create a total of 6 magenta cities that are equally spaced across the top of the green box. Name them `city1` through `city6` , from left to right.

3 Vectors

Definition of a Vector

To put it simply, a vector is an arrow. The arrow has a tail and a head, each of which is at a point on a coordinate system. Thus, the arrow can be defined by the coordinates of its head and the coordinates of its tail. In Figure 3.1, the head of the vector is at $(+1, +4)$ m, and the tail is at $(-4, -2)$ m.

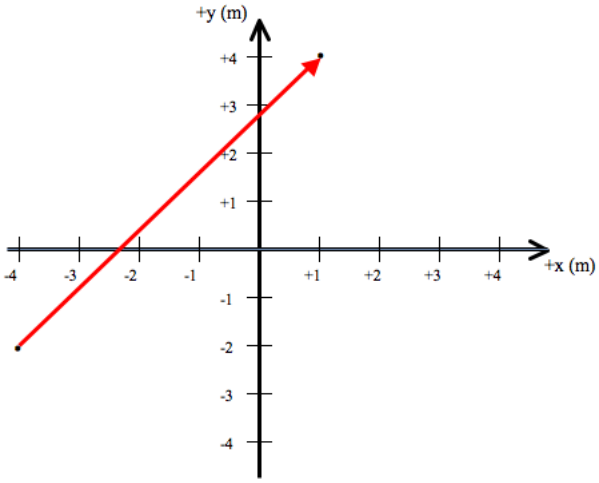


Figure 3.1: A vector is simply an arrow.

Vector Components

You can define a vector another way – by specifying the coordinates of the tail and then specifying how many units over and upward (or downward) you have to count in order to get to the head of the vector.

For example, in Figure 3.2, the tail of the vector is at $(-4, -2)$ m. To get to the head of the arrow, you can "walk" to the right 5 meters and then "walk" upward 6 meters. (By "walking," I mean take your pencil and count over to the right along the dashed line until you get to the $+1$ coordinate. Then, count upward along the dotted line until you get to the head of the arrow.)

The dashed line, the dotted line, and the arrow form a right-triangle. The horizontal dashed line is parallel to the x-axis and is called the *x-component* of the vector. The vertical dotted line is parallel to the y-axis and is called the *y-component* of the vector.

In the example in Figure 3.2, the x-component of the vector is $+5$ meters, and the y-component of the vector is $+6$ vectors. We will write a vector's components using parentheses as well, such as $(+5, +6)$ m. But note that coordinates (x, y) are different than vector components (horizontal component, vertical component). They have different meanings.

The $+$ sign on the x-component means that we had to count over *to the right*. The $+$ sign on the y-component means that we had to counter *upward*. If we had counted to the left, then the x-component would be negative. If we had counted downward, then the y-component would be negative.

Symbols for a vector are written with an arrow on top of them. For example, we could name this vector

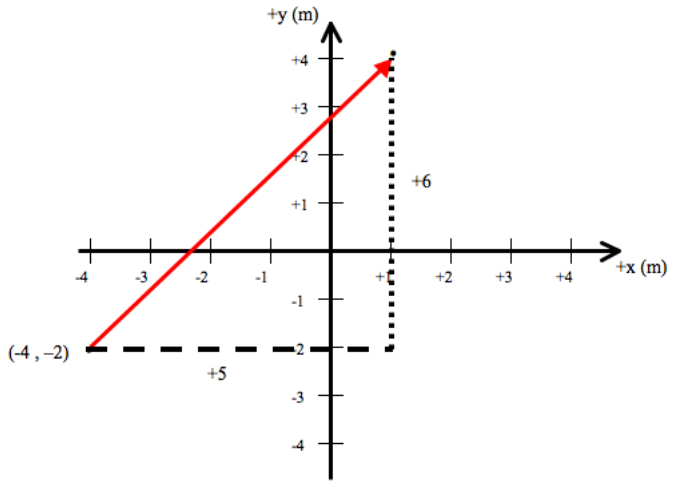


Figure 3.2: The x and y components of a vector.

$\vec{A}$. Then, $\vec{A} = (+5, +6)$ m. The x and y components are written as A_x and A_y, respectively. So, in general $\vec{A} = (A_x, A_y)$.

We can define the vector by the coordinates of its tail and its head. Alternatively, we can define the vector by the coordinates of its tail and the vector's components. These methods are identical and lead one to draw the same arrow. (See the summary in Table 3.1.)

Table 3.1: Two ways to define vector $\vec{A}$ in this example.

Method 1 to describe vector $\vec{A}$	**Method 2 to describe vector $\vec{A}$**
tail location is at: $(-4, -2)$ m head location is at: $(+1, +4)$ m	tail location is at: $(-4, -2)$ m vector components are: $(+5, +6)$ m

Head = Tail + Vector Components

If you know the tail of a vector and you know its components, then you can calculate the coordinates of the head of the vector.

$$\begin{aligned} \text{(head)} &= \text{(tail)} + \text{(vector components)} \\ \text{for example: } (+1, +4) \text{ m} &= (-4, -2) \text{ m} + (+5, +6) \text{ m} \end{aligned}$$

To add the tail and vector components, you add the x-values and y-values separately. Thus,

$$\begin{aligned} \text{x:} \quad +1 &= -4 + 5 \\ \text{y:} \quad +4 &= -2 + 6 \end{aligned}$$

which gives the location of the head as: $(+1, +4)$ m. Maybe it is easier to write it vertically, as shown below.

$$\begin{array}{rr} & (-4,-2)\ \text{m} \\ + & (+5,+6)\ \text{m} \\ \hline & (+1,+4)\ \text{m} \end{array}$$

Vector Components = Head - Tail

If you know the coordinates of the head and tail of a vector, you can calculate the vector's components by:

$$\begin{aligned} (\text{vector components}) &= (\text{head}) - (\text{tail}) \\ \text{for example: } (+5,+6)\ \text{m} &= (+1,+4)\ \text{m} - (-4,-2)\ \text{m} \end{aligned}$$

Writing it vertically looks like:

$$\begin{array}{rr} & (+1,+4)\ \text{m} \\ - & (-4,-2)\ \text{m} \\ \hline & (+5,+6)\ \text{m} \end{array}$$

Examples

Question:
The tail of vector is at (+3,-4) m. Its head is at (+1,-1) m. Does the vector point to the right or to the left? Does the vector point upward or downward?

Answer:
Find the vector components.

$$\begin{aligned} (\text{vector components}) &= (\text{head}) - (\text{tail}) \\ &= (+1,-1)\ \text{m} - (+3,-4)\ \text{m} \\ &= (-2,+3)\ \text{m} \end{aligned}$$

Because the vector's x-component is negative, the vector points to the left. Because its y-component is positive, the vector points upward. You can also answer this question by drawing the vector on a coordinate system and observing that it points to the left and upward.

Question:
Computers, by default, define the $+x$ axis to the right and the $+y$ axis downward, with the origin at the top left corner of the monitor. The tail of vector $\vec{\text{B}}$ is at (300,100) pixels, and $\vec{\text{B}} = (150, 200)$ pixels. Where is the head of $\vec{\text{B}}$?

Answer:

$$\begin{aligned} (\text{head}) &= (\text{tail}) + (\text{vector components}) \\ &= (300, 100)\ \text{pixels} + (150, 200)\ \text{pixels} \\ &= (450, 300)\ \text{pixels} \end{aligned}$$

Magnitude of a Vector

Vectors have two essential properties: (1) length and (2) direction. The length of a vector is also called its *magnitude* and is written $|\vec{A}|$.

The vector and its components make up a right triangle, as shown in Figure 3.2. The vector is the hypotenuse, and the components are the sides.

Pythagorean's theorem for a right triangle is $c^2 = a^2 + b^2$. When applied to a right triangle, Pythagorean theorem gives:

$$|\vec{A}|^2 = A_x^2 + A_y^2$$

Example

Question:
What is the length (i.e. magnitude) of $\vec{A}$ in Figure 3.2?

Answer:
$\vec{A} = (+5, +6)$ m, so $|\vec{A}|$ is

$$\begin{aligned} |\vec{A}| &= \sqrt{A_x^2 + A_y^2} \\ &= \sqrt{(5\text{ m})^2 + (6\text{ m})^2} \\ &= 7.81\text{ m} \end{aligned}$$

Notice that the hypotenuse is longer than each side of the right triangle, as expected.

Multiplying a vector by a scalar

When you multiply a vector by a scalar, you multiply each component by that scalar. If a is a scalar quantity, then

$$a\vec{r} = < ar_x, ar_y, ar_z >$$

The magnitude of this vector is thus $a|\vec{r}|$. Multiplying a vector by a scalar just *scales* the vector–this only changes the magnitude of the vector and not the direction unless the scalar is negative. Multiplying a vector by -1, "reverses" the vector. In other words, $-\vec{r}$ points in the opposite direction as $\vec{r}$.

Figure 3.3 shows vector $\vec{B}$ and the result of multiplying it by 2 and the result of multiplying it by -1.

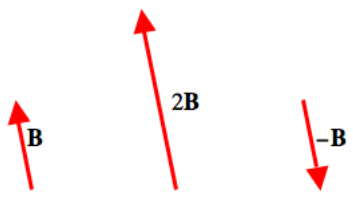


Figure 3.3: Multiplying a vector by a scalar.

Example

Question:
A missile has a velocity vector $\vec{v}_1 = (100, 20, -50)$ m/s. What is its speed (i.e. the magnitude of its velocity)? If another missile has moving twice as fast, what is its velocity vector?

Answer:
$\vec{v}_1 = (100, 20, -50)$ m/s, so $|\vec{v}|$ is

$$\begin{aligned} |\vec{v}_1| &= \sqrt{v_x^2 + v_y^2 + v_z^2} \\ &= \sqrt{(100)^2 + (20)^2 + (50)^2} \text{ m/s} \\ &= 114 \text{ m/s} \end{aligned}$$

The second missile is traveling twice as fast. Its speed is $2|\vec{v}_1| = 227$ m/s. (Note that 114 m/s was a rounded quantity.) Its velocity vector is $2\vec{v}_1$:

$$\begin{aligned} \vec{v}_2 = 2\vec{v}_1 &= 2(100, 20, -50) \text{ m/s} \\ &= (200, 40, -100) \text{ m/s} \end{aligned}$$

Direction of a Vector

The direction of a vector is specified by a unit vector. A *unit vector* is a vector with a magnitude of 1. But if we know a vector, how do we find its associated unit vector (i.e. its direction)? A unit vector in the direction of $\vec{r}$ is calculated by

$$\hat{r} = \frac{\vec{r}}{|\vec{r}|}$$

$$\hat{r} = \left\langle \frac{r_x}{|\vec{r}|}, \frac{r_z}{|\vec{r}|}, \frac{r_z}{|\vec{r}|} \right\rangle$$

Note that a unit vector is written $\hat{r}$ with a "hat" on top of the variable. It is pronounced "r-hat."

A few specially defined unit vectors are $\hat{i}$, $\hat{j}$, and $\hat{k}$ which point along the x, y and z axes, respectively. They are written as

$$\hat{i} =< 1, 0, 0 >$$

$$\hat{j} =< 0, 1, 0 >$$

$$\hat{k} =< 0, 0, 1 >$$

Suppose that a vector points in the $-x$ direction, then its unit vector is $< -1, 0, 0 >$. Likewise $< 0, -1, 0 >$ points in the –y direction, and $< 0, 0, -1 >$ points in the –z direction.

Example

Question:
A missile has a velocity vector $\vec{v}_1 = (100, 20, -50)$ m/s. What is its direction? (Note: the direction of a vector is specified by its unit vector.)

Answer:
$\vec{v} = (100, 20, -50)$ m/s, so $\hat{v}$ is

$$\begin{aligned} \hat{v} &= \frac{\vec{v}}{|\vec{v}|} \\ &= \frac{(100, 20, -50) \text{ m/s}}{114 \text{ m/s}} \\ &= (0.880, 0.176, -0.440) \end{aligned}$$

Note that there are no units because they cancel out. A unit vector only gives us direction, and nothing else.

Question:
A missile is traveling with a speed of 50 m/s in the -y direction. What is its velocity vector?

Answer:
$\vec{v} = |\vec{v}|\hat{v}$.

$$\begin{aligned} |\vec{v}| &= |\vec{v}|\hat{v} \\ &= 50(0, -1, 0) \\ &= (0, -50, 0)\text{m/s} \end{aligned}$$

Position

The coordinates of an object on a coordinate system are really a vector, with the tail of the vector at the origin. In Figure 3.4, an object at the coordinates (2,3) has a position vector $\vec{r} = (2, 3)$ m (if the units are assumed to be meters).

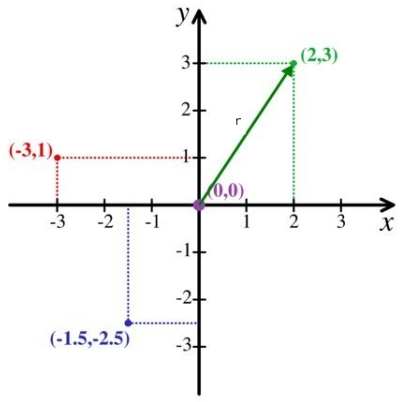


Figure 3.4: A position of an object is a vector from the origin to the coordinates of the object.

Vectors in VPython

One of the reasons that we are using the `visual` package and Python is that it knows how to add and subtract vectors and it knows how to multiply a vector and a scalar. Let's experiment with this now.

1. First, create a new VPython program with the standard import statements at the beginning. (See your previous program as an example.)

   Multiplying by a scalar; Scaling an arrow's axis

   Since position of a sphere is a vector, we can perform scalar multiplication on it.

2. Place the tennis ball at the position $\langle 2, 0, 0 \rangle$ m.

3. Modify the position of the tennis ball by changing the statement to the following, and note what happens.

```
tennisball = sphere(pos=2*vector(2,0,0), radius=0.2, color=color.green)
```

QUESTIONS TO ANSWER ABOUT SCALING ARROWS: To move the tennis ball three times further on the x-axis, what scalar would you multiply its position by? To move the tennis ball to the other side of the origin by multiplying by a scalar factor, what factor should you use?

Vector addition and subtraction

The *relative position* of an object is the object's position relative to a location other than the origin. If point P is a given location in space, then the position of an object relative to P is

$$\vec{r}_{\text{relative to P}} = \vec{r} - \vec{r}_{\text{P}}$$

VPython can subtract vectors. So now, you can create a second object, a baseball, and draw an arrow from the tennis ball to the baseball.

4. Place the tennis ball at the position $\langle 2, 0, 0 \rangle$ m.

5. Create a sphere at the position $\langle -1, 4, 0 \rangle$ m and name the sphere `baseball` .

6. Create an arrow that represents the position of the baseball. It should point from the origin to `baseball.pos` , so its `pos` is $(0, 0, 0)$ and its axis is `baseball.pos` . To do this, type the line below.

```
arrow(pos=vector(0,0,0), axis=baseball.pos, color=color.white)
```

7. Create an arrow that represents the position of the baseball relative to the tennis ball. Note that its tail is at the position of the tennis ball, and its head is at the position of the baseball. The axis of the arrow represents the relative position vector and is calculated by:

$$\vec{r}_{\text{baseball relative to tennisball}} = \vec{r}_{\text{baseball}} - \vec{r}_{\text{tennis ball}}$$

In symbolic notation in VPython, write, this is calculated as `baseball.pos-tennisball.pos` . So, write:

```
1 arrow(pos=tennisball.pos, axis=baseball.pos-tennisball.pos, color=color.
      white)
```

Note that `pos` represents the tail of the arrow. The vector is the `axis` of the arrow which is really the position of the head relative to the position of the tail.

Homework

1. The source of a missile is at (100, 20, 0) pixels. A city is at (600, 400, 0) pixels. What is the vector that points from the source (tail) to the city (head)?

2. The vector that points from a source of a missile at (500, 20, 0) pixels to a city that is at (200, 500, 0) pixels has the components (-300, 480, 0) pixels. What is the magnitude of this vector and what is its direction?

3. In the game *Frogger*, a log is at (10,5,0) m. It moves with a speed of 1 m/s in the $-x$ direction. What is its velocity vector?

4. You have learned how to create spheres and arrows in VPython. In this activity, you will practice what you've learned by creating a new program.

   Start a new file by going to the `File` menu and selecting `New window`. Again, the first two lines of your program will be:

   ```
   from __future__ import division
   from visual import *
   ```

   The program you will write makes a model of the Sun and various planets. The distances are given in scientific notation. In VPython, to write numbers in scientific notation, use the letter "e" to represent the phrase "times ten to the." For example, the number 6.4×10^7 is written as 6.4e7 in a VPython program.

   Create a model of Sun and three of the inner planets: Mercury, Venus, and Earth. The distances from Sun to each of the planets are given by the following:

   Mercury: 5.8×10^{10} m from the sun
   Venus: 1.1×10^{11} m from the sun
   Earth: 1.5×10^{11} m from the sun

   The inner planets all orbit Sun in roughly the same plane, so place them in the x-y plane. Place Sun at the origin, place Mercury at $< d_i, 0, 0 >$, place Venus at $< -d_i, 0, 0 >$, and place Earth at $< 0, d_i, 0 >$, where d_i represents the distance from Sun to the particular planet i.

   If you use the real radii of the Sun and the planets in your model, they will be too small for you to see. So use these values:

   Radius of Sun: 7.0×10^9 m
   Radius of Mercury: 2.4×10^9 m
   Radius of Venus: 6.0×10^9 m
   Radius of Earth: 6.4×10^9 m

   The radius of Sun in this program is ten times larger than the real radius, while the radii of the planets in this program are 1000 times larger than the real radii.

   Finally make two arrows:

   (a) Create an arrow that points from Earth to Mercury. Do not use any numbers to specify the position and axis of the arrow.

   (b) Imagine that a space probe is on its way to Venus, and that it is currently halfway between Earth and Venus. Make a relative position vector that points from the Earth to the current position of the probe. Do not use any numbers to specify the position and axis of the arrow.

4 Uniform Motion

Displacement

The displacement of an object is its change in position. It is defined as:

$$\begin{aligned} \text{displacement} &= \text{final position coordinates} - \text{initial position coordinates} \\ \Delta\vec{r} &= \vec{r}_f - \vec{r}_i \end{aligned}$$

(Remember that the coordinates themselves are a vector so we are really subtracting vector quantities in this equation.) Suppose that in an animation, a car moves to the right as shown in Figure 4.1. Its final position coordinates are (+2,+3) m. Its initial position coordinates are (-4,+3) m. Thus, the displacement of the car is: displacement = $(2, 3)$ m $-$ $(-4, 3)$ m = $(+6, 0)$ m

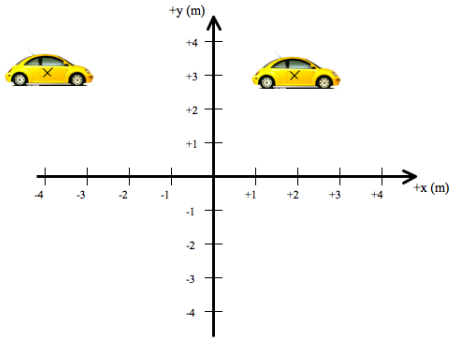


Figure 4.1: A car moves to the right.

An object's displacement is a vector whose tail is at the object's initial position and head is at the object's final position. The displacement of the car is the vector shown in Figure 4.2.

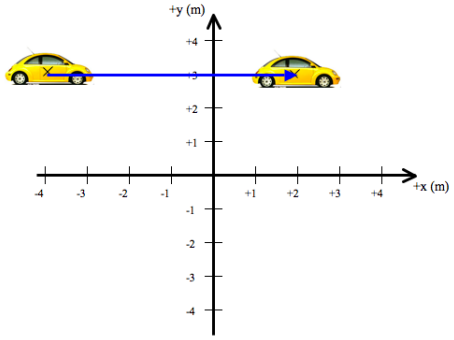
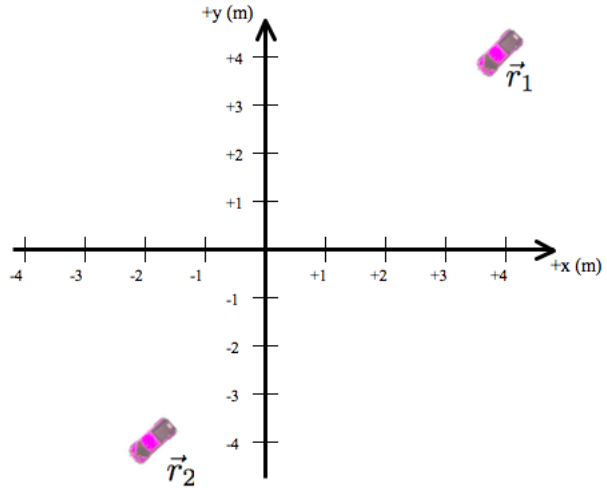


Figure 4.2: The car's displacement vector.

Since the displacement is $(+6, 0)$ m and has zero y-component, then it must be a vector that is parallel to the x-axis and points 6 m to the right. Thus, the mathematical result agrees with the picture in Figure 4.2.

Example

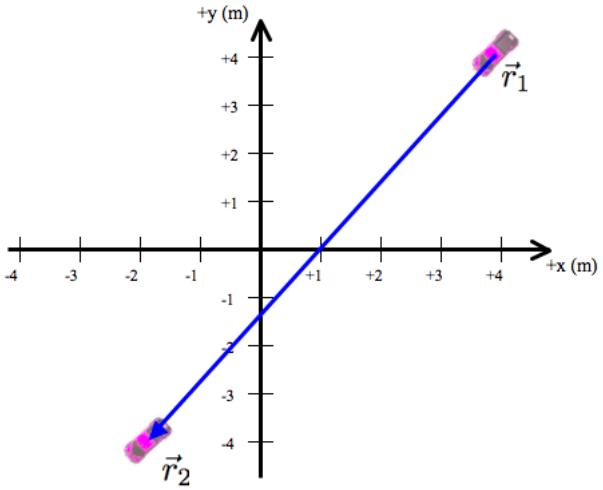
Question:
The top view of a pickup moving from $\vec{r}_1$ to $\vec{r}_2$ is shown below. Sketch and calculate the truck's displacement.



Answer:
The displacement of the pickup is:

$$\begin{aligned} \text{displacement} &= \vec{r}_2 - \vec{r}_1 \\ &= (-2, -4)\ \text{m} - (+4, +4)\ \text{m} \\ &= (-6, -8)\ \text{m} \end{aligned}$$

To sketch the displacement vector, draw an arrow from the initial position of the pickup to the final position of the pickup, as shown below.



Updating the position of an object

When creating games or animation, you move objects on the screen. To move an object to some "final" coordinates, you take its initial coordinates and add a displacement.

$$\text{final position coordinates} \quad = \quad \text{initial position coordinates} + \text{displacement}$$

Suppose that a toy pickup is at (+4,-2) m and you displace it (-3,0) m. Then, its new position is

$$\begin{aligned} \text{final position coordinates} &= (+4, -2)\ \text{m} + (-3, 0)\ \text{m} \\ &= (+1, -2)\ \text{m} \end{aligned}$$

The position of the pickup at $\vec{r}_2$ is shown in Figure 4.3.

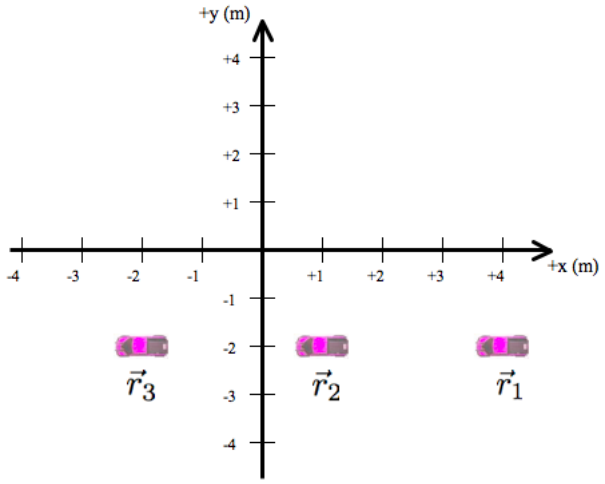


Figure 4.3: A pickup is displaced (-3,0) m to the left.

Example

Question:
Suppose that a toy pickup is at (+4,-2) m and you displace it (-3,0) m. Let's call this *step 1*. After two *additional* steps, what will be the position of the pickup?

Answer:
There is a total of three "steps". After the first step, the pickup is at $(+4, -2)\ \text{m} + (-3, 0)\ \text{m} = (+1, -2)\ \text{m}$. After the second step, the pickup is at $(+1, -2)\ \text{m} + (-3, 0)\ \text{m} = (-2, -2)\ \text{m}$, as shown in Figure 4.3. After the third step, the pickup is at $(-2, -2)\ \text{m} + (-3, 0)\ \text{m} = (-5, -2)\ \text{m}$.
It is a good idea to extend the axis in Figure 4.3 and sketch the location of the pickup at $\vec{r}_4$.

Uniform motion

Each step of the pickup shown in Figure 4.3 takes place in a time interval Δt. If you have a running clock to measure time, then the *clock reading* when the pickup is at $\vec{r}_1$ is t_1. The clock reading when the pickup is at $\vec{r}_2$ is t_2. The *time interval* Δt (or time elapsed) is defined as the difference in the clock readings:

$$\Delta t = t_2 - t_1$$

Suppose that we use the same time interval between successive displacements. If the object's displacement in equal time intervals is the same, then the result is called *uniform motion*. It's fairly easy to identify uniform motion because successive pictures of the object in equal time intervals are equally spaced apart, as shown in Figure 4.3 for the pickup.

For example, consider a simulation of a cart moving to the right on a track shown in Figure 4.4. The dots represent the location of the center of the cart at a certain clock reading. Since the clock readings were made at equal time steps and since successive positions of the cart are equally spaced, the cart's motion is described as uniform motion.

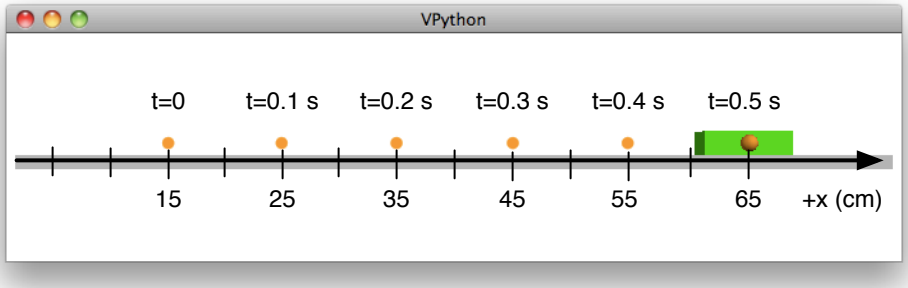


Figure 4.4: A cart on a track moves to the right with uniform motion.

Let's record the x-positions of the cart and the clock readings in a data table.

Table 4.1: x-positions of the cart in Figure 4.4.

t (s)	x (cm)
0	15
0.1	25
0.2	35
0.3	45
0.4	55
0.5	65

A graph of x as a function of time is shown in Figure 4.5. You will notice that a best-fit "curve" through the data is a straight line. The "rise" is 10 cm for each "run" of 0.1 s. This gives a constant slope for the curve. The slope is

$$\begin{aligned} slope &= \frac{rise}{run} \\ &= \frac{10 \text{ cm}}{0.1 \text{ s}} \\ &= 100 \text{ cm/s} \end{aligned}$$

The slope means that the x-position of the cart increases 10 cm for each 0.1 s of time elapsed, or in other words, 100 cm in each 1 second elapsed. This is called the *x-velocity* of the cart.

Note that the graph of x vs. t doesn't tell us anything about the y-motion of the cart. In this case, the cart is moving horizontally, so its *y-velocity* is zero. As a result, we can write the velocity of the cart as $\vec{\mathrm{v}} = (100, 0)$ cm/s. But if we didn't know its y-velocity and all we had was the data for graph for $x(t)$, then we wouldn't know anything about the y-motion of the cart.

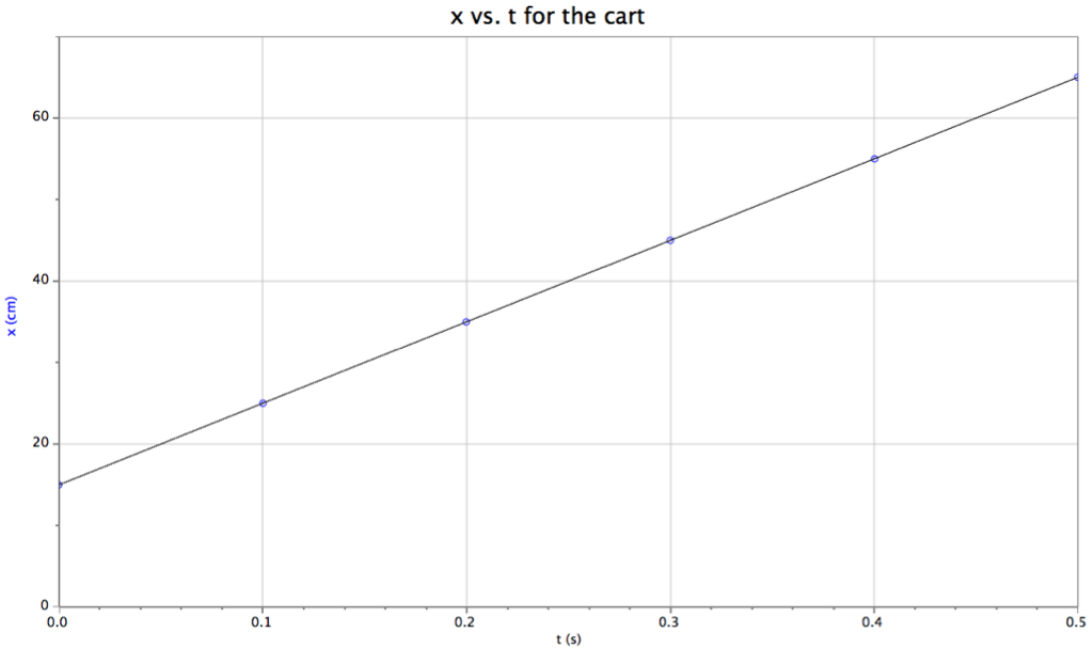
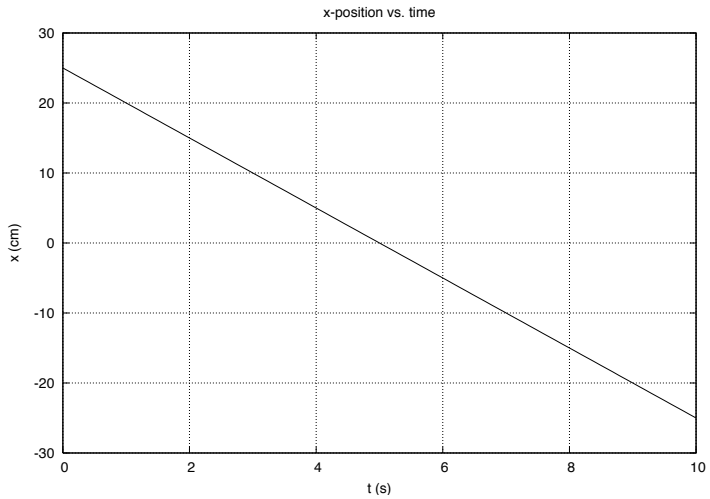


Figure 4.5: A graph of x-position as a function of time for the cart.

Example

Question:
A graph for the x-position as a function of time for a cart on a track is shown below.



(a) What is the x-velocity of the cart?

(b) In which direction is the cart traveling?

(c) What can you say about the y-motion of the cart?

Answer:
(a) The x-velocity is the slope of the graph. Choose two points on the line. For example, you might choose (1 s, 20 cm) and (4 s, 5 cm). The slop is

$$\begin{aligned} slope &= \frac{rise}{run} \\ &= \frac{(5-20)\text{ cm}}{(4-1)\text{ s}} \\ &= \frac{-15\text{ cm}}{3\text{ s}} \\ &= -5\text{ cm/s} \end{aligned}$$

(b) The x-velocity of the cart is negative. If we define the $+x$ axis on our coordinate system to the right, then the cart is moving to the left.
(c) This graph only tells us the x-velocity. We have no idea what the y-velocity is. Maybe the cart is traveling down a ramp, to the left. Or maybe it's traveling up a ramp, to the left. Or maybe it's traveling on a horizontal ramp. We just don't know.

Velocity

The velocity of an object is its displacement per second. It is calculated by

$$\begin{aligned} \text{velocity} &= \frac{\text{displacement}}{\text{time interval}} \\ \vec{\text{v}} &= \frac{\Delta\vec{\text{r}}}{\Delta t} \end{aligned}$$

The cart in Figure 4.4 has a displacement of (+10,0) cm between each dot, and the time interval between dots is 0.1 s. Thus, the cart's velocity is:

$$\begin{aligned} \vec{\text{v}} &= \frac{(+10, 0)\ \text{cm}}{0.1\ \text{s}} \\ &= (100, 0)\ \text{cm/s} \end{aligned}$$

Predicting the future position of an object

If you know the velocity of an object, you can calculate what its displacement will be in any given time interval. The object's displacement in a time interval Δt is

$$\text{displacement} = \text{velocity} \times \text{time interval}$$

You can predict its future position after a time interval Δt by adding its displacement to its initial position.

$$\begin{aligned} \text{final position coordinates} &= \text{initial position coordinates} + \text{velocity} \times \text{time interval} \\ \vec{\text{r}}_{\text{f}} &= \vec{\text{r}}_{\text{i}} + \vec{\text{v}}\Delta t \end{aligned}$$

For the cart in this example, we can predict its position at any future clock reading, assuming that it continues moving with the same velocity. Since $x_i = 15$ cm at $t = 0$, then at $t = 0.6$ s,

$$\begin{aligned} x_{\text{f}} &= x_{\text{i}} + v_{\text{x}}\Delta t \\ &= 15\ \text{cm} + (100\ \text{cm/s})(0.6\ \text{s}) \\ &= 75\ \text{cm} \end{aligned}$$

We could have used $x_i = 65$ cm at $t = 0.5$, then at $t = 0.6$ s

$$\begin{aligned} x_{\text{f}} &= x_{\text{i}} + v_{\text{x}}\Delta t \\ &= 65\ \text{cm} + (100\ \text{cm/s})(0.1\ \text{s}) \\ &= 75\ \text{cm} \end{aligned}$$

The result is the same, as long as you use the time interval Δt since the object was at the initial x-position x_i.

Making things move

In animation, you make things move one step at a time. If each step occurs in a time interval Δt, then the new position of the object after each step is:

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

Suppose that the time interval is $\Delta t = 0.5$ s and the object begins at $\vec{r}_i = (4, -2)$ m and has a velocity of $(-1, 2)$ m/s. After one time step, the object will be at:

$$\begin{aligned} \text{new position coordinates} &= (4, -2) \text{ m} + \big((-1, 2) \text{ m/s}\big)(0.5 \text{ s}) \\ &= (3.5, -1) \text{ m} \end{aligned}$$

After the next time step, the object will be at:

$$\begin{aligned} \text{new position coordinates} &= (3.5, -1) \text{ m} + \big((-1, 2) \text{ m/s}\big)(0.5 \text{ s}) \\ &= (3, 0) \text{ m} \end{aligned}$$

Using this method, you can continue to move the object at 0.5 s time steps. Its positions, starting at $(4, -2)$ m at 0.5 s time steps are shown in Table 4.2. The object is drawn at each time step in Figure 4.6.

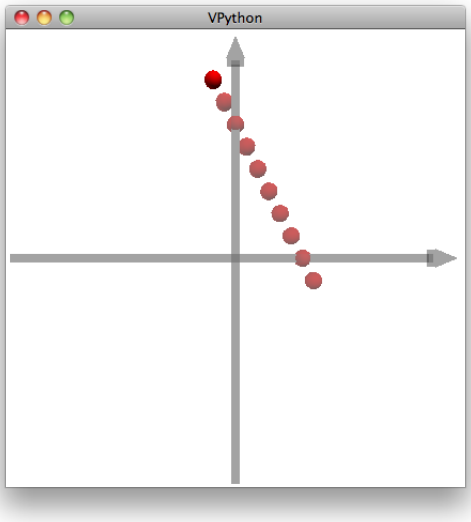


Figure 4.6: Updated positions of the object from Table 4.2.

Table 4.2: Updated positions at time steps of 0.5 s for a velocity of $(-1, 2)$ m/s.

coordinates (m)	t (s)
0	(4.0 , -2.0)
0.5	(3.5 , -1.0)
1.0	(3.0 , 0.0)
1.5	(2.5 , 1.0)
2.0	(2.0 , 2.0)
2.5	(1.5 , 3.0)
3.0	(1.0 , 4.0)
3.5	(0.5 , 5.0)
4.0	(0.0 , 6.0)
4.5	(-0.5 , 7.0)
5.0	(-1.0 , 8.0)

Example

Question:
In a game, a missile starts at the location $(500, 500, 0)$ m at $t = 0$ and travels with a constant velocity of $(-43, -25, 0)$ m/s. Using a time step of 2.0 s, find the next 10 positions of the missile. Make a data table showing the missile's position at each clock reading.
Answer:
After the first time step, the new position of the missile is

$$\begin{aligned} \text{new position coordinates} &= \text{current position coordinates} + \text{velocity} \times \text{time interval} \\ \vec{r}_f &= \vec{r}_i + \vec{v}\Delta t \\ &= (500, 500, 0)\ \text{m} + \big((-43, -25, 0)\ \text{m/s}\big)(2\ \text{s}) \\ &= (414, 450, 0)\ \text{m} \end{aligned}$$

and the clock reading is $t = 0 + 2\ \text{s} = 2\ \text{s}$.
After the second time step, the new position of the missile is

$$\begin{aligned} \text{new position coordinates} &= \text{current position coordinates} + \text{velocity} \times \text{time interval} \\ &= (414, 450, 0)\ \text{m} + \big((-43, -25, 0)\ \text{m/s}\big)(2\ \text{s}) \\ &= (328, 400, 0)\ \text{m} \end{aligned}$$

and the clock reading is $t = 2 + 2\ \text{s} = 4\ \text{s}$. Continue to do this for a total of 10 time steps. A data table with the results is shown below.

t (s)	coordinates (m)
0	(500.0 , 500.0)
2	(414.0 , 450.0)
4	(328.0 , 400.0)
6	(242.0 , 350.0)
8	(156.0 , 300.0)
10	(70.0 , 250.0)
12	(-16.0 , 200.0)
14	(-102.0 , 150.0)
16	(-188.0 , 100.0)
18	(-274.0 , 50.0)
20	(-360.0 , 0.0)

Homework

1. In a certain video game that you create, an object travels diagonally from the right side to the left side. Suppose that you use real-world coordinates (not pixels) in your program, with the $+x$ axis defined to the right and the $+y$ axis defined upward. At $t = 0$, an object is at $(100, 200, 0)$ m and has a velocity of $(-10, 0, 0)$ m/s.

   (a) What is the object's displacement during a time interval of 0.1 s?

   (b) After one time step of $\Delta t = 0.1$ s, what is the object's new position?

   (c) What is the object's position at $t = 5$ s? (Note that you can use the original position at $t = 0$ and use a time interval of 5 s to find its new position.)

   (d) What is the object's position at $t = 10$ s? What about $t = 15$ s?

2. An object moves along the y-axis with uniform motion; its y-position as a function of time is shown in Figure 4.7.

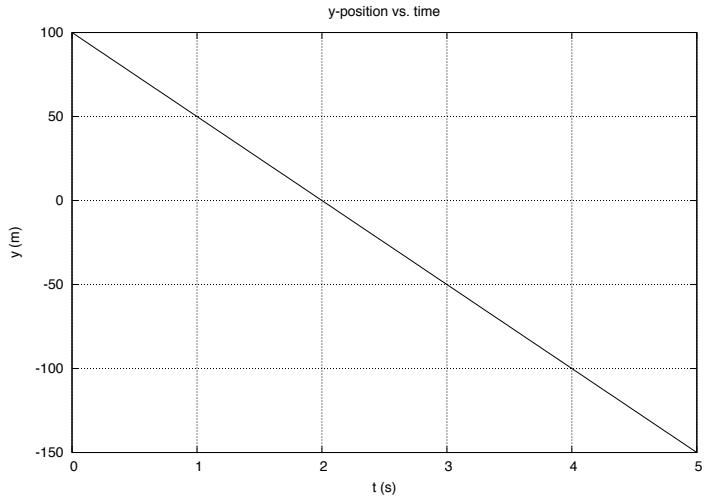


Figure 4.7: y(t) graph for an object

   (a) What is the y-velocity of the object?

   (b) What is its y-position at $t = 3$ s?

   (c) What is its initial y-position (i.e. $t = 0$)?

3. In writing games, you will frequently make an object move at constant velocity. Suppose you are writing a *Galaga* game where your spaceship moves back and forth horizontally at a constant velocity. You use real-world coordinates and a real-world coordinate system, and you choose a speed of 10 m/s for the spaceship.

   (a) If the spaceship is moving to the right at the given speed, what is its velocity (written as a vector)?

   (b) If you use time steps of $\Delta t = 0.05$ s in your animation and if the spaceship is at the position $(-50, 0, 0)$ m, what will be its position during the next 10 time steps? Show each calculation of each step.

5 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.

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 5.1.

Figure 5.1: A ball on a track.

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?

6 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 6.1 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.

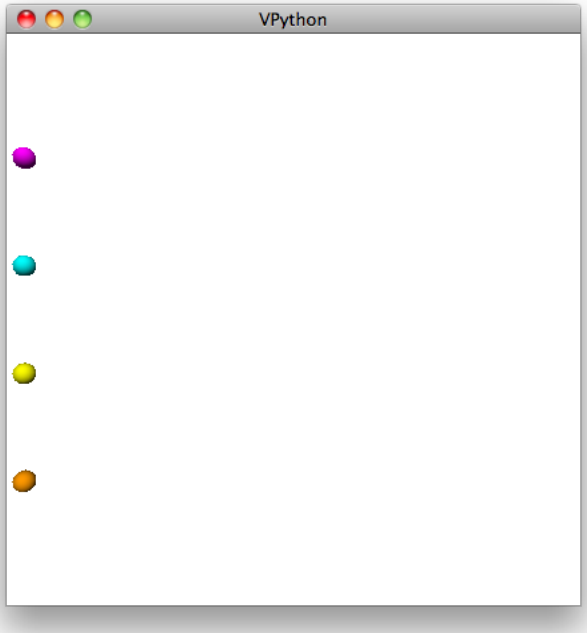


Figure 6.1: Four balls

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.

7 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.

8 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 8.1:

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.

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 8.2.

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.

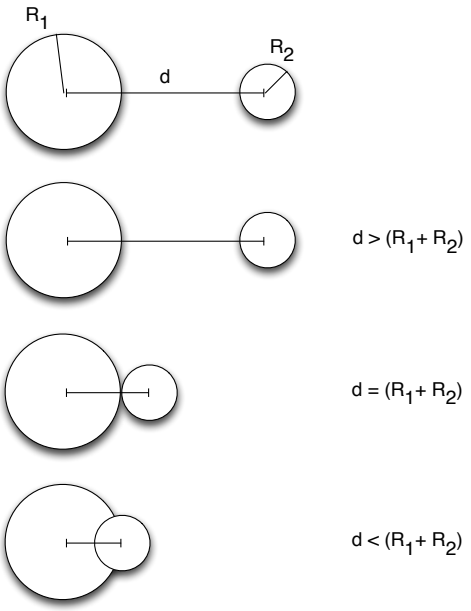


Figure 8.1: Condition for whether two spheres collide.

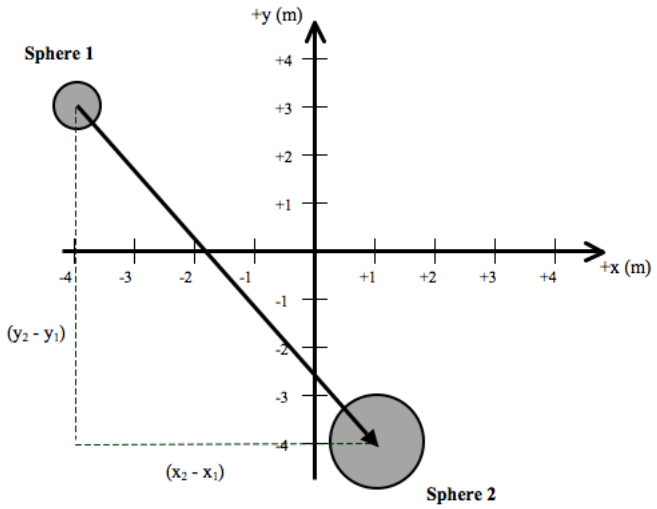


Figure 8.2: Distance between two spheres.

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.

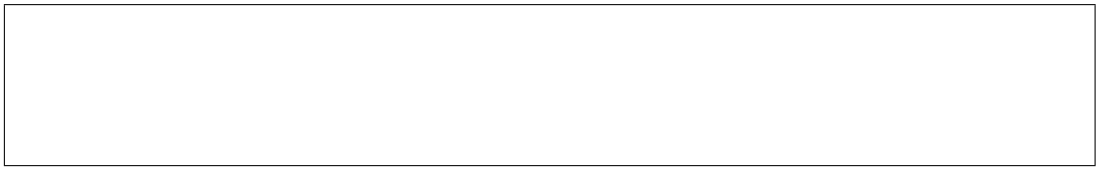
B Do everything for **C** and the following.

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.

9 Galilean Relativity

Introduction

In a Mythbusters episode called *Vector Vengeance*, the crew shoots a soccer ball out the back of a pickup truck. However, they chose the muzzle speed to be exactly the same speed as the truck, with the muzzle velocity opposite the truck's velocity. (A single frame is shown in Figure 9.1.)

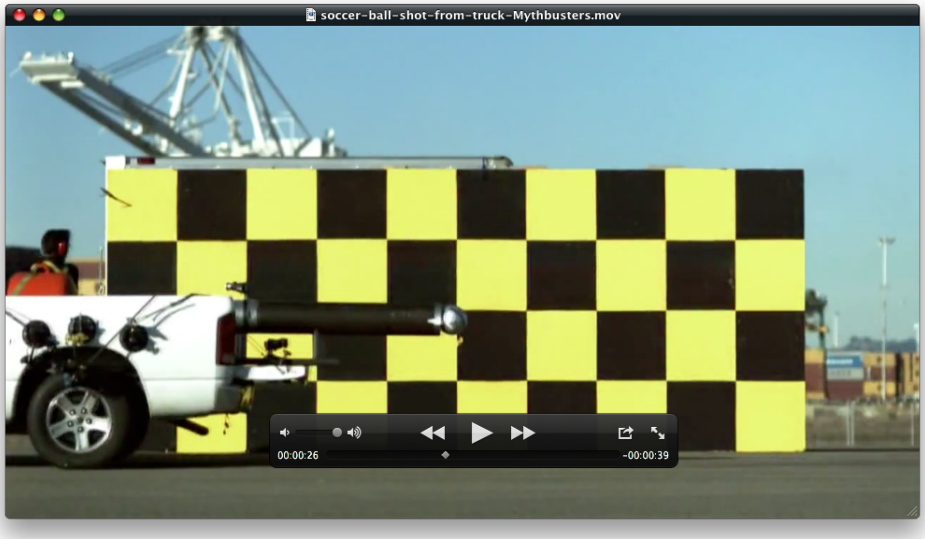


Figure 9.1: A soccer ball shot out the back of a pickup truck.

When the ball exits the barrel, what will be its path as viewed by a person on the ground?

If the crew increases the muzzle speed of the ball, what will be its path as viewed by a person on the ground?

If the crew decreases the muzzle speed of the ball, what will be its path as viewed by a person on the ground?

Relative Velocity

There are three velocities to think about in the Mythbusters video:

1. The muzzle velocity of the ball $\vec{v'}$ is the velocity of the ball as measured by a person who is sitting at rest with respect to the gun. We will call this the *Other* frame because it is not the frame of reference of you who is presumably holding the camera or standing next to it.

2. The velocity of the ball with respect to the ground $\vec{v}$ is the velocity of the ball as measured by a person who is at rest with respect to the ground. This is *you* and is called the *Home* frame.

3. The velocity of the *frame* itself $\vec{\beta}$ is the velocity of the truck as measured by a person on the ground.

These three velocities are related by:

$\vec{v'}$: velocity of an object measured by an observer in the *Other* frame
$\vec{v}$: velocity of an object measured by an observer in the *Home* frame
$\vec{\beta}$: velocity of the *Other* frame as measured in the *Home* frame

$$\vec{v'} = \vec{v} - \vec{\beta} \qquad \text{Galilean Transformation Equation}$$

Note that this is a vector equation, so it must hold true for the x, y, and z components respectively.

$$\begin{aligned} v'_x &= v_x - \beta_x \\ v'_y &= v_y - \beta_y \\ v'_z &= v_z - \beta_z \end{aligned}$$

A very important point to realize is that *your* velocity in *your* reference frame is always zero. Observers are not moving in their own reference frames. Thus, the velocity of the *Home* frame is always zero by definition.

Example

Question:
A Mythbusters crew shoots a soccer ball to the right out the back of a pickup truck with a muzzle speed of 20 m/s. The truck is moving with a speed of 25 m/s to the left.
(a) In what direction is the ball moving, relative to a person on the ground, after it exits the muzzle?
(b) What is the ball's x-velocity and speed, relative to a person on the ground, after it exits the muzzle?
Answer:
The "Other" reference frame in this case is the pickup truck. The soccer ball's x-velocity relative to the muzzle is $v\prime_x = +20$ m/s. The soccer ball's x-velocity relative to the ground is the unknown $\vec{v}$. Solve the Galilean transformation equation above for the unknown.

$$\begin{aligned} v_x &= v'_x + \beta_x \\ &= 20\text{m/s} - \text{m/s} \\ &= -5\text{m/s} \end{aligned}$$

The x-velocity is $v_x = -5$ m/s which means that the ball is (a) moving to the left with a speed (b) $|v| = 5\ m/s$ when it leaves the gun.

Back to the shooter game.

In the last chapter, you finished writing a simple shooter game where you move a box right and left on a keyboard and press the spacebar to fire bullets. *But there was one major problem with our simulation. It violated physics (unless you design a special mechanism inside the box.*

What is wrong with the motion of the bullets in our simulation?

Example

Question:
A shooter is moving with a velocity of 2 m/s in the $-x$ direction when it fires a bullet in the $+y$ direction with a muzzle speed of 5 m/s. What is the velocity of the bullet for a stationary observer?

Answer:
The "Other" reference frame in this case is the shooter which has a velocity $\vec{\beta} = (-2, 0, 0)$ m/s. The bullet's muzzle velocity is $\vec{v'} = (0, 5, 0)$ m/s. The bullet's velocity in the Home frame is

$$\begin{aligned} \vec{v} &= \vec{v'} + \vec{\beta} \\ &= (0, 5, 0) \text{ m/s} + (-2, 0, 0) \text{ m/s} \\ &= (-2, 5, 0) \end{aligned}$$

Though the bullet is moving upward at a speed of 5 m/s, it still moves to the left with a velocity of -2 m/s. As a result, it will stay above the shooter as long as the shooter continues to move with a constant velocity.

Homework

1. A shooter is moving with a velocity of 3 m/s in the $+x$ direction. You want it to fire a bullet so that the bullet will move vertically (+y direction) in the Home frame with a speed of 4 m/s. What should be the velocity of the ball (in the reference frame of the shooter) so that the shooter will cause the missile to travel vertically with a speed of 4 m/s?

2. A frog is riding a log that is moving in the $+y$ direction with a speed of 3 m/s. If the frog launches itself in the $-x$ direction with a speed of 1.5 m/s, what will be the frog's velocity relative to an observer on the riverbank?

3. A person in a spaceship reports to you that a bullet was launched with a velocity of (3,-4,0) m/s. You measure the velocity of the ball and find that it is (0,-2,0) m/s. What is the velocity of the spaceship relative to you?

10 Collision with a Stationary Rigid Barrier

Introduction

Collisions, in general, are an important part of physics. At the Large Hadron Collider, physicists accelerate particles like protons and antiprotons to speeds very close to the speed of light and then collide them. The collision produces all kinds of other particles, including quarks, the fundamental particles of which protons are made.

Collisions are an important part of games as well. You have already learn how to check for collisions between spheres in VPython. But how should objects react after colliding? Our goals are to:

1. understand the physics of collisions in the real world.
2. understand how to use physics to create realistic collisions.
3. understand how to violate the laws of physics in specific ways in order to make a game that is more enjoyable to play. Or saying it another way, understand the machinery required to make objects in a game model real objects in nature. That is, know how to intelligently lie so that you can say that the game behaves in a physically correct way.

We will begin by studying collisions between balls and massive rigid barriers, like a floor or wall or bumper on a billiards table.

Coefficient of restitution

When a ball collides with a stationary, rigid barrier, it will either rebound with the same speed or it will slow down as a result of the collision. If it rebounds with the same speed, then it is an *elastic* collision. If it slows down as a result of the collision, then it is an *inelastic* collision. In nature, when a ball bounces off the floor or wall or something like that, it nearly always slows down.

The coefficient of restitution (COR) for an object colliding with a stationary, rigid barrier is defined as:

$$C_R = \frac{v_f}{v_i}$$

where v_f is the speed of the object after the collision and v_i is the speed of the object before the collision. Note that COR is always less than or equal to 1. If COR$= 1$, then it is an elastic collision. If COR< 1, then it's an inelastic collision.

Note that COR depends on the materials of both the object and the barrier. If you change the material of the object or the material of the barrier, you will affect the COR.

The COR tells you how "bouncy" a ball-floor system is, for example. If you drop a ball on the floor and it rebounds close to the same height it was dropped from, then the COR is high (meaning closer to 1). If you drop a ball on the floor and it barely rebounds at all, then the COR is low (meaning closer to zero). If you change the ball or if you change the floor from tile to wood, for example, you will measure a different COR.

1-D collision

In a one-dimensional collision of an object with a stationary rigid barrier, the direction of the object reverses and the speed after the collision will be less than or equal to the speed before the collision.

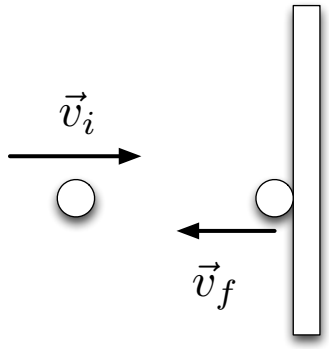


Figure 10.1: A ball collides with a rigid wall.

The velocity of the ball is a vector. Thus, you must find the speed using

$$v_i = \sqrt{v_{i,x}^2}$$

(If the ball is moving in the y or z directions, then use the appropriate component of the velocity.)

Example

Question:
A rubber ball has a velocity (3,0,0) m/s before it collides with a concrete wall and (-2,0,0) m/s after it collides with the wall. What is the COR of the ball and wall?
Answer:
The initial speed of the ball is 3 m/s. The final speed of the ball is 2 m/s. Thus, the COR is

$$\begin{aligned} C_R &= \frac{2 \text{ m/s}}{3 \text{ m/s}} \\ &= 0.67 \end{aligned}$$

Question:
If you use a different wall, perhaps one that is made of drywall nailed to wood studs, will the COR be the same or different?
Answer:
The COR depends on the material of both the ball and wall. If you change the material of the wall, you will likely get a different final speed after the collision.

2-D collision – frictionless

When an object collides with a frictionless, stationary barrier, the collision only changes the component of the velocity that is perpendicular to the surface. The component of the velocity parallel to the surface stays

the same.

In the example in Figure 10.2, the ball has an initial velocity in the +x and +y directions. The velocity of the ball is written as $\vec{v} = (v_x, v_y)$ (in two dimensions). However, it is important to rewrite it in terms of the collision where one component is parallel to the surface and one component is perpendicular to the surface that it collides with. In this example, $\vec{v} = (v_\perp, v_\parallel)$ (in two dimensions) where $v_\perp$ is in the x-direction and $v_\parallel$ is in the y-direction.

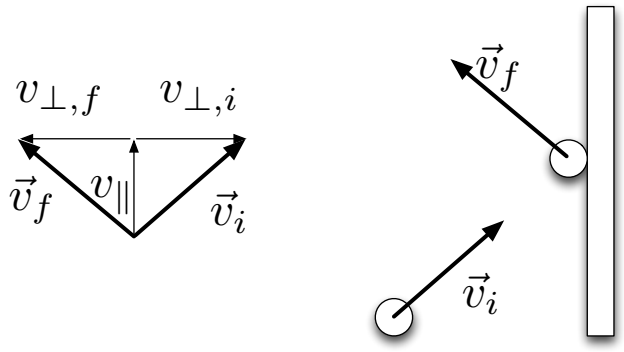


Figure 10.2: A 2-D collision with a frictionless rigid wall.

After colliding with the wall, the y-component of the ball (which is parallel to the surface of the wall) remains the same. However, the x-velocity of the ball (which is perpendicular to the wall) changes direction. Only the component of the velocity perpendicular to the surface changes due to the collision. Not only does it reverse direction, but it may also decrease in magnitude. The perpendicular component of the velocity after the collision will be

$$v_{\perp,f} = C_R v_{\perp,i}$$

If it is an elastic collision, then $v_{\perp,f}$ merely changes direction as is shown in Figure 10.2.

2-D collision – friction

Friction acts parallel to the surfaces in contact and changes the parallel component of the velocity of the object. For a collision with a stationary rigid barrier, then in this case friction always causes the $v_\parallel$ to decrease in magnitude. If the barrier is moving, then it's possible to increase $v_\parallel$.

Consider a hockey puck bouncing off of a rigid hockey stick as shown in Figure 10.3. Let's assume, for the sake of simplicity, that it is an elastic collision. In this case, the frictional force on the ball is opposite to $v_\parallel$ and thus decreases the value of $v_\parallel$.

What if the hockey stick is moving in the $+y$ direction when the puck hits the stick? Then, the direction of the frictional force depends on the velocity of the puck *relative to the stick*. If the stick is moving faster than

the puck (in the parallel direction), the frictional force is in the direction of $v_{\parallel}$ and causes it to increase. If the stick is moving slower than the puck, the frictional force is opposite to the direction of $v_{\parallel}$ and causes it to decrease. If the stick is moving with a speed equal to $v_{\parallel}$, then the frictional force is zero and $v_{\parallel}$ is constant.

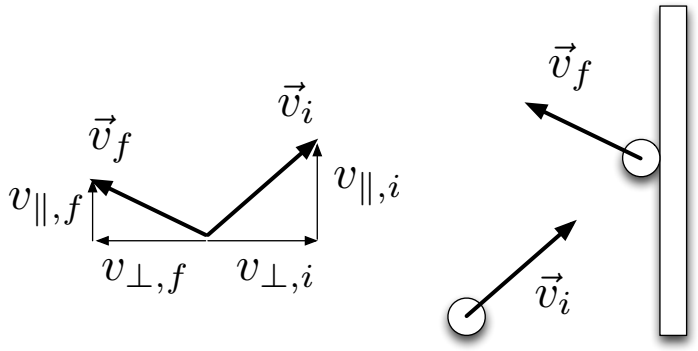


Figure 10.3: A 2-D collision with a frictionless rigid wall.

Homework

1. A golf ball is falling vertically and has a speed of 1.5 m/s just before it bounces off the floor. After the bounce, it has a speed of 0.6 m/s. What is the coefficient of restitution of the ball and floor?

2. If the golf ball in the previous question were dropped off a table instead of the floor, is the coefficient of restitution going to be the same or different?

3. A hockey puck on an air hockey table has a velocity of $(2.0,\ -1.2)$ m/s when it collides from a side wall of a hockey rink as shown in Figure 10.4.

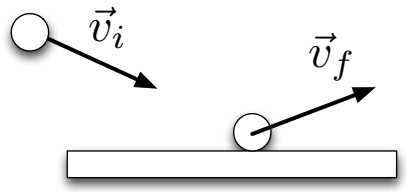


Figure 10.4: A 2-D collision with a frictionless rigid wall.

   (a) What is $v_{\parallel}$ before the collision?
   (b) What is $v_{\perp}$ before the collision?
   (c) If the wall is frictionless and if the collision is elastic, what is the velocity of the puck after the collision?
   (d) If the wall is frictionless and if the COR is 0.4, what is the velocity of the puck after the collision?

4. A pickup truck is moving to the right when a ball is tossed into the back of the truck as shown in Figure 10.5. The truck's velocity is $(3, 0)$ m/s.

   (a) Suppose that there is friction between the bed of the truck and the ball during the collision. Assume that the collision is elastic. If the ball's velocity relative to the ground before the collision is $(2.0,\ -1.2)$ m/s, in what direction is the frictional force on the ball and does $v_{\parallel}$ increase, decrease, or remain constant as a result of the collision?
   (b) Suppose that there is friction between the bed of the truck and the ball during the collision. If the ball's velocity relative to the ground before the collision is $(4.0,\ -1.2)$ m/s, in what direction is the frictional force on the ball and does $v_{\parallel}$ increase, decrease, or remain constant?
   (c) Suppose that there is friction between the bed of the truck and the ball during the collision. If the ball's velocity relative to the ground before the collision is $(3.0,\ -1.2)$ m/s, in what direction is the frictional force on the ball and does $v_{\parallel}$ increase, decrease, or remain constant?

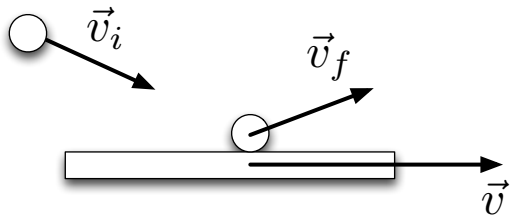


Figure 10.5: A 2-D collision with a moving rigid object.

11 GAME – Pong

Apparatus

Computer
VPython – www.vpython.org

Goal

The purpose of this activity is to study the classic arcade game Pong and use VPython to develop both a physical and an unphysical version of the game.

Procedure

Playing Pong

1. Go to `http://www.ponggame.org/` and play the classic Pong Game. Try both the keyboard and mouse to control the paddle.

Pay attention to the motion of the ball when colliding with the wall and a paddle. 1. Is the collision of the ball and a side wall elastic or inelastic? Explain your answer by referring to observations of the motion of the ball. 2. Are the side walls frictionless or not? Explain your answer by referring to observations of the motion of the ball. 3. Is the collision of the ball and a paddle elastic or inelastic? Explain your answer by referring to observations of the motion of the ball. 4. Is the paddle frictionless or not? Explain your answer by referring to observations of the motion of the ball.

Creating a "bouncing" puck in a box

We are going to simulate a puck on an air hockey table that is bouncing around the table. In this simulation, we will assume that the walls and paddle are rigid, frictionless barriers. We will also assume

that the puck and barriers make elastic collisions. Thus, the COR is 1 for all collisions. For simplicity, we will draw the puck as a ball and refer to it as a ball.

2. Begin a new program. Import the visual package.
3. Set the size of the scene. You may want to set the height and width of the window in pixels. The example below will set the range to be 10 (meters or whatever unit you wish you use), the width to be 600 pixels, and the height to be 600 pixels.

```
scene.range=20
scene.width=600
scene.height=600
```

4. Create walls at the top, bottom, and sides of the screen.
5. Run your program and verify that you have four walls around the perimeter.
6. Create a ball at the center.
7. Define the initial velocity of the ball (`ball.v=vector(5,8,0)`), the initial clock reading (`t=0`), and the time step (`dt=0.01`).
8. Create an infinite while loop.
9. Use `rate(100)` to slow down the simulation so that the motion is smooth.
10. Update the position of the ball.

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

11. Use an `if-elif` statement to check for a collision between the ball and each wall. If there is a collision, change the velocity of the ball in an appropriate way.
12. Run your program and verify that it works properly.

Creating inelastic collisions

In the real world, a ball would lose energy upon colliding with a rigid barrier. Said another way, the coefficient of restitution is always less than 1. Now we will change the last program by adding friction and a coefficient of restitution.

13. Save your program with a different name so that you don't lose the work that you just did.
14. Make the left wall a "real wall" that causes the ball to lose speed upon colliding with the wall. In other words, after colliding with the left wall, the ball's velocity would be reduced by a factor less than 1. You define a variable *COR* which you can change to be whatever value you want (between 0 and 1). A COR of 0 means that the ball would stick to the wall. A COR of 1 would be an elastic collision.

```
ball.v.x = -COR*ball.v.x
```

It helps to use a smaller COR, like 0.5 or less, to see the affect more quickly.

Describe the motion of the ball after a long time. Could we have predicted this given the fact that only one wall results in inelastic collisions?

Suppose that a wall has a spring in it that "punches" the ball during the collision, similar to the bumpers in a pinball machine. Then, you could model this wall by giving it a COR greater than 1.

15. Make one of the walls "super elastic" by giving it a COR greater than 1. (This is like the bumper in a pinball machine.)

16. Run your program and observe the effect of the collisions on the motion of the ball.

Adding friction to a collision

Friction acts parallel to the wall in order to reduce the parallel component of the velocity of the ball.

17. Save your program with a different name so that you do not lose your previous work.

18. Make all collisions elastic collisions (i.e. $COR = 1$).

    Add friction to the left wall by changing the y-component of the velocity of the ball when it collides with the wall. Exactly how friction affects the velocity of the ball is a bit complicated. Let's use a simple (albeit unphysical) model that reduces the parallel component of the velocity of the ball by a certain percentage. This is similar to the COR for the perpendicular component of the velocity.

19. When the ball collides with the left wall, change the y-component of the velocity by a factor of 20% or something like that. (A factor of 1 is no friction and a factor of 0 is maximum friction.)

```
ball.v.y=0.2*ball.v.y
```

20. Run your program.

Describe the motion of the ball after a long time. Could we have predicted this given the fact that only one wall has friction?

Making a 1-player Pong game

21. Create a new VPython program. As always, begin by importing the visual package.

22. It might be nice to set the width and height of the window in pixels. Use the code below to set the range to 20 (m or whatever units you want to imagine), the width to 600 pixels, and the height to 450 pixels. You are welcome to use a larger width and height if you wish.

```
scene.range=20
scene.width=600
scene.height=450
```

23. Create walls for the ceiling and floor. Also create a wall on the left side that has a hole in it that represents a goal. This is similar to what you see in air hockey, for example. You'll need two boxes on the left side, with a space between them for the goal.

24. Create small box as a paddle on the right side. You will eventually use your mouse to move this box up and down.

25. Create a ball and make its initial velocity something like (15,12,0) m/s.

26. Create variables for the clock and time step.

27. Create an infinite while loop. Update the position of the ball. Check for collisions with the walls and paddle and change the velocity accordingly. For now, assume elastic, frictionless collisions. Run your program and verify that everything works as expected.

28. Now we will move the paddle with the mouse. Add the following code inside your infinite while loop. Note that I used the names `paddle2`, `ceiling`, and `floor` for my objects. You will have to changes these to be the same names that you used for your objects. You do not need to retype the comments.

```
#get mouse position and move paddle2
mouse=scene.mouse.pos
#check if mouse is within walls and set y-position of paddle to
    the mouse
if(mouse.y-paddle2.height/2>floor.y+floor.height/2 and mouse.y+
    paddle2.height/2<ceiling.pos.y-ceiling.height/2):
     paddle2.pos.y=mouse.y
else:
    #place paddle at center of mouse is outside the walls
    if(mouse.y-paddle2.height/2<floor.y+floor.height/2):
        paddle2.pos=(18,0,0)
    elif(mouse.y+paddle2.height/2>ceiling.y-floor.height/2):
        paddle2.pos=(18,0,0)
```

29. Run the program and see if you can control the paddle.

30. Check to see if the ball goes past the paddle or past the left walls. If it does, break out of the loop.

Analysis

C Complete this exercise.

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

1. Make one of the left walls a super-elastic wall with $COR > 1$ and one of the left walls an inelastic wall with $COR < 1$.
2. Make half your paddle a super elastic paddle with $COR > 1$ and half your paddle an inelastic paddle with $COR < 1$. Give each half different colors.

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

1. Place your infinite loop inside another infinite loop. When the ball goes past the paddle or through the goal, increment a score, reset the ball to the middle of the scene, and pause the game and wait for a mouse click or key press. Use the function below to pause the game. Call `pause()` when you want the game to pause.

```
def pause():
    while True:
        rate(50)
        if scene.mouse.events:
            m = scene.mouse.getevent()
            if m.click == 'left': return
        elif scene.kb.keys:
            k = scene.kb.getkey()
            return
```

2. Change your program so that the ball bounces off the paddle in a similar way as the *Pong* game that you played at the beginning of this chapter. Note that the physics is incorrect unless you invent a mechanical device that would cause the ball to bounce in this way.

12 Newton's Second Law

Acceleration

The acceleration of an object is a rate of change in its velocity:

$$\begin{aligned} \text{acceleration} &= \frac{\text{later velocity} - \text{earlier velocity}}{\text{time interval}} \\ \vec{a} &= \frac{\vec{v}_f - \vec{v}_i}{\Delta t} \\ \vec{a} &= \frac{\Delta \vec{v}}{\Delta t} \end{aligned}$$

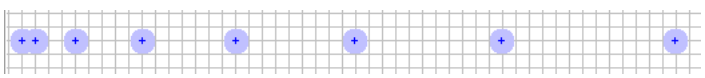
Since velocity is a vector with both magnitude and direction, an object as non-zero acceleration if the (1) magnitude of velocity changes; (2) the direction of velocity changes; or (3) both magnitude and direction of velocity changes.

Speeding up and slowing down in a straight line

If the magnitude of the velocity changes, but not its direction, then the object speeds up or slows down but continues to move in a straight line. An example is the fancart that you analyzed in a previous experiment. If the acceleration and velocity of the cart were in the same direction, then the cart sped up. If the acceleration and velocity of the cart were in opposite directions, then the cart slowed down. You can observe the fact that the cart is accelerating by viewing the marks. If the marks get further apart or closer together, then the object is accelerating.

Example

Question:
Suppose that a puck starts at the right side and moves toward the left side of the image shown below. Marks show the position of the puck at equal time steps. Using our standard definitions of $+x$, $+y$, and $+z$ directions, what is the direction of the velocity and the acceleration of the puck? Is the puck speeding up or slowing down?



Answer:
The puck's velocity is in the direction it is moving which is to the left. Thus, its velocity is in the $-x$ direction. (You can say that its x-component is negative and its y and z components are zero.)
The puck is slowing down as observed by the decreasing distance between marks of the puck. Therefore, its acceleration is in the opposite direction as its velocity, or the $+x$ direction.

Question:
Suppose that the puck in the previous question starts at the left side and moves toward the right side of the image. Marks show the position of the puck at equal time steps. Using our standard definitions of $+x$, $+y$, and $+z$ directions, what is the direction of the velocity and the acceleration of the puck? Is the puck speeding up or slowing down?
Answer:
The puck's velocity is in the direction it is moving which is to the right. Thus, its velocity is in the $+x$ direction. (You can say that its x-component is negative and its y and z components are zero.)
The distance between marks of the puck is increasing; therefore, the puck is speeding up. As a result, its acceleration is in the same direction as the velocity (the $+x$ direction).

Note that the sign of the acceleration in both of the examples above is positive. It alone does not tell you whether the object will speed up or slow down.

Changing direction with constant speed

Whenever an object travels along a curved path, it also has an acceleration. Even if it travels with a constant speed, the direction of its velocity changes; therefore, it has a non-zero acceleration. We can calculate the acceleration in the same way: $\vec{a} = \frac{\Delta\vec{v}}{\Delta t}$. But to visualize the direction of the acceleration, you should find the direction of $\Delta\vec{v}$. Follow this procedure:

1. Sketch the vectors $\vec{v}_f$ and $\vec{v}_i$.
2. Off to the side, sketch $\vec{v}_f$ and $\vec{v}_i$ so that they are drawn tail to tail. Be sure to keep their lengths and directions the same.
3. Sketch the vector $\Delta\vec{v}$ from the head of $\vec{v}_i$ to the head of $\vec{v}_f$

For example, suppose an object travels in a circle with constant speed, as shown in Figure 12.1. Its velocity at two different instances of time is indicated. What is the direction of its acceleration during this time interval?

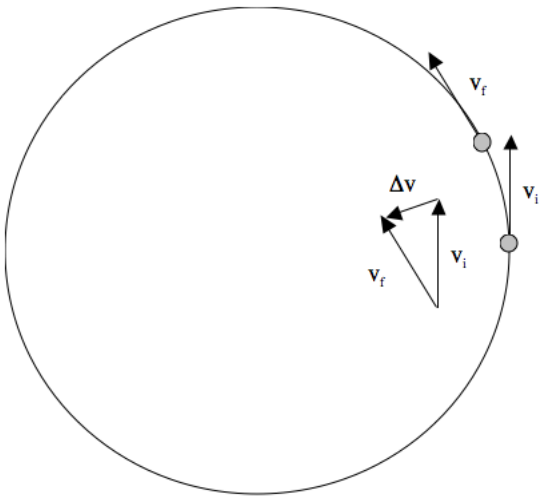
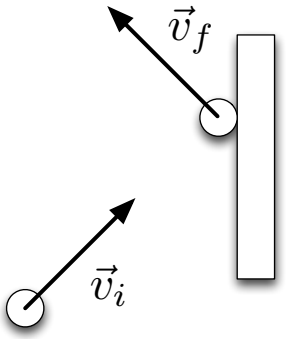


Figure 12.1: Velocity at two locations as it moves along a circle at constant speed.

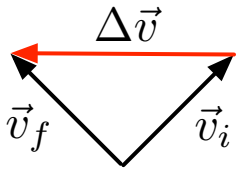
If you draw the velocity vectors at any locations close together on the circle, you will find that the acceleration vector points toward the center. This is a general observation, *the acceleration of an object that travels in a circle at a constant speed points toward the center of the circle.*

Example

Question:
A puck bounces off the wall on an air hockey table as shown below. Draw the acceleration vector of the puck during the time interval of the collision.



Answer:
Sketch the initial and final velocity vectors tail to tail. Then draw $\Delta\vec{v}$ from the head of $\vec{v}_i$ to the head of $\vec{v}_f$. This is the direction of the acceleration. In this case, you can see that it is perpendicular to the wall. The wall must be frictionless.



Newton's second law

The sum of the forces acting on an object is called the *net force*. A net force on an object causes it to accelerate. The object's acceleration is proportional to the net force on the object. The

$$\text{acceleration of an object} = \frac{\text{net force on the object}}{\text{mass of the object}}$$

$$\vec{a} = \frac{\vec{F}_{net}}{m}$$

The acceleration of an object is in the same direction as the net force that acts on it. The net force is what *causes* the acceleration. For a given net force, the larger the mass of the object, the smaller acceleration it will have.

Predicting the future

If we know the net force on an object and its velocity, then we can predict its velocity a small time interval later. Since $\vec{a} = (\vec{v}_f - \vec{v}_i)/\Delta t$, then

$$\begin{aligned} \vec{a} &= \frac{\vec{F}_{net}}{m} \\ \frac{\vec{v}_f - \vec{v}_i}{\Delta t} &= \frac{\vec{F}_{net}}{m} \\ \vec{v}_f &= \vec{v}_i + \frac{\vec{F}_{net}}{m}\Delta t \end{aligned}$$

This means that Newton's second law can predict the future! It can tell you what the velocity of an object will be after a time interval Δt. This equation assumes that the net force is constant. If the net force is not constant–that is if the net force is changing during the time interval Δt– then we have to use a small time interval.

$$\vec{v}_f \approx \vec{v}_i + \frac{\vec{F}_{net}}{m}\Delta t \quad \text{for a non-constant force and small time interval}$$

Do not think of $\vec{v}_f$ as "final" velocity, but rather think about it as the object's new velocity after a time interval Δt. In a simulation, we will call this "updating" the velocity of the object. Perhaps it is easier to write the equation as:

$$\text{new velocity} \approx \text{old velocity} + \frac{\vec{F}_{net}}{m}\Delta t \qquad \text{velocity update equation}$$

Once we know the object's velocity, then we can calculate its new position using the position update equation.

$$\text{new position} \approx \text{old position} + \text{new velocity} * \Delta t \qquad \text{position update equation}$$

This equation is approximate because the object's velocity is changing due to the force, yet we are assuming for the sake of this calculate that the velocity of the object is constant. This assumption only works for a small time step.

Newton's second law not only explains motion in everyday life, it also allows us to make predictions. Given that we can calculate the net force on an object, we can predict the object's position and velocity at any time in the future by doing these calculations iteratively one small time step after another.

Wanna know exactly where Jupiter, Mars, and Venus will be on April 6, 2100? Easy! Just apply Newton's second law and it will tell you.

Summary of iterative method to predict the future

1. Calculate the net force on the object.
2. Calculate the new velocity of the object.
3. Calculate the new position of the object.
4. Repeat step 1.

Example

Question:
A 0.4 kg fancart starts at rest at the origin. The air (due to the turning fan) exerts a constant 2 N on the fan in the $+x$ direction. Use the iterative method to calculate the velocity and position of the cart at $t = 0.1$ s, $t = 0.2$ s, $t = 0.3$ s, $t = 0.4$ s, and $t = 0.5$ s.
Answer:
After the first time step, the new velocity of the fancart is

$$\begin{aligned} \text{new velocity} &\approx \text{old velocity} + \frac{\vec{F}_{net}}{m}\Delta t \\ &= (0,0,0) + \frac{(2,0,0)\text{ N}}{0.4\text{ kg}}0.1\text{ s} \\ &= (0.5,0,0)\text{ m/s} \end{aligned}$$

The new position of the fancart is (approximately)

$$\begin{aligned} \text{new position} &\approx \text{old position} + \text{new velocity} * \Delta t \\ &= (0,0,0) + ((0.5,0,0)\text{ m/s})\,(0.1\text{ s}) \\ &= (0.05,0,0)\text{ m} \end{aligned}$$

and the clock reading will now read $t = 0 + 0.2\text{ s} = 0.2$ s.
After the next time step, the new velocity of the fancart is

$$\begin{aligned} \text{new velocity} &\approx \text{old velocity} + \frac{\vec{F}_{net}}{m}\Delta t \\ &= (0.5,0,0)\text{ m/s} + \frac{(2,0,0)\text{ N}}{0.4\text{ kg}}0.1\text{ s} \\ &= (0.5,0,0)\text{ m/s} + (0.5,0,0)\text{ m/s} \\ &= (1,0,0)\text{ m/s} \end{aligned}$$

The new position of the fancart is

$$\begin{aligned} \text{new position} &\approx \text{old position} + \text{new velocity} * \Delta t \\ &= (0.05,0,0)\text{ m} + ((1,0,0)\text{ m/s})\,(0.1\text{ s}) \\ &= (0.15,0,0)\text{ m} \end{aligned}$$

and the clock reading is now $t = 0.2 + 0.2\text{ s} = 0.4$ s. Continue to calculate the new velocity and new position iteratively for each time step.

t (s)	velocity (m/s)	position (m)
0	(0 , 0)	(0, 0)
0.1	(0.5 , 0.0)	(0.05 , 0.0)
0.2	(1.0 , 0.0)	(0.15 , 0.0)
0.3	(1.5 , 0.0)	(0.3 , 0.0)
0.4	(2.0 , 0.0)	(0.5 , 0.0)
0.5	(2.5 , 0.0)	(0.75 , 0.0)

These are approximate calculations. They have some error and would be more accurate if we used smaller time steps, like 0.01 s.

Homework

1. A football bounces off the grass as shown below.

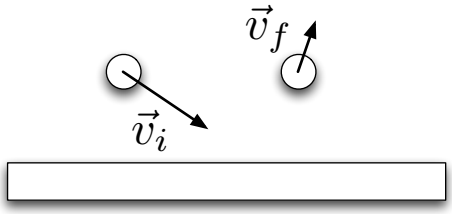


Figure 12.2: Velocity before and after a football bounces off the grass.

   Sketch the direction of the acceleration of the football during the collision with the ground.

2. A ball has an initial position $(-4.5, 0, 0)$ m and an initial velocity $(5.74, 8.19, 0)$ m/s.

   (a) What is its position and velocity at $t = 0.25$ s, $t = 0.5$ s, $t = 0.75$ s, $t = 1.0$ s, $t = 1.25$ s, and $t = 1.5$ s?

   (b) Sketch a coordinate system and sketch the path of the ball by drawing the ball on the coordinate system and connecting the images of the ball with a smooth curve.

13 PROGRAM – Modeling motion of a fan-cart

Apparatus

VPython
computer

Goal

In this activity, you will learn how to use a computer to model motion with a constant net force. Specifically, you will model the motion of a fan cart on a track.

Introduction

We are going to model the motion of a cart using the following data.

mass of cart	0.8 kg
$\vec{F}_{\text{net on cart}}$	$< 0.15, 0, 0 >$ N

Procedure

1. Begin with a program that simulates a cart moving with constant velocity on a track.

```
from visual import *

track = box(pos=vector(0,-0.05,0), size=(3.0,0.05,0.1), color=color.white)
cart = box(pos=vector(-1.4,0,0), size=(0.1,0.04,0.05), color=color.green)

cart.m = 0.8
cart.v = vector(1,0,0)

dt = 0.01
t = 0

scene.mouse.getclick()

while cart.pos.x < 1.5 and cart.pos.x >-1.5:
        rate(100)
        cart.pos = cart.pos + cart.v*dt
        t = t+dt
```

2. Run the program

What does line 12 do? It may help to comment it out and re-run your program to see how it changes things.

What line updates the position of the cart for each time step?

What line updates the clock for each time step?

Is the clock used in any calculations? Is it required for our program?

What line causes the program to stop if the cart goes off the end of the track?

We will now apply Newton's second law in order to apply a force to the cart and update its velocity for each time step. There are generally three things that must be done in each iteration of the loop:

(a) calculate the net force (thought it will be constant in this case)
(b) update the velocity of the cart
(c) update the position of the cart
(d) update the clock (*this is not necessary but is often convenient*)

Your program is already doing the third and fourth items in this list. However, the first two items must be added to your program.

3. Between the `rate()` statement and the position update calculation, insert the following two lines of code:

```
Fnet=vector(-0.15,0,0)
cart.v = cart.v + (Fnet/cart.m)*dt
```

The first line calculates the net force on the cart (though it is just constant in this case). The second line updates the velocity of the cart in accordance with Newton's second law. After making this change, your `while` loop will look like:

```
while cart.pos.x < 1.5 and cart.pos.x >-1.5:
        rate(100)
        Fnet=vector(-0.15,0,0)
        cart.v = cart.v + (Fnet/cart.m)*dt
        cart.pos = cart.pos + cart.v*dt
        t = t+dt
```

This block of code performs the necessary calculations of net force, velocity, position, and clock reading.

4. Run your program and view the motion.

What is the direction of the net force on the cart? Sketch a side view of the fancart that shows the orientation of the fan.

Now we will add an arrow object in order to visualize the net force on the cart. An arrow in VPython is specified by its position (the location of the tail) and its axis (the vector that the arrow represents), as shown in Figure 13.1. The axis contains both magnitude and direction information. The magnitude of the axis is the arrow's length and the unit vector of the axis is the arrow's direction. The components of the axis are simply the components of the vector that the arrow represents.

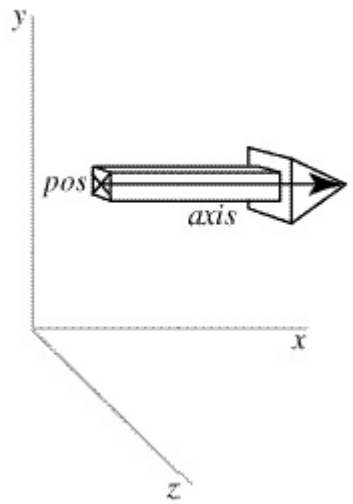


Figure 13.1: A arrow in VPython.

5. Near the top of your program after creating the track and cart, add the following two lines to define a scale and to create an arrow that has the same components $(-0.15, 0, 0)$ as the net force on the cart.

```
scale=1.0
forcearrow = arrow(pos=cart.pos, axis=scale*vector(-0.15,0,0), color=color
    .yellow)
```

6. Run your program.

7. Increase the scale and re-run your program.

What does changing the scale do? Why do we want to use this variable and adjust it?

Why does the arrow not move with the cart?

8. We want to make the arrow move with the cart. Thus, in our loop we need to update the position of the arrow after we update the position of the cart. Also, in some situations, the force changes, so in general it's a good idea to update the arrow's axis as well. At the bottom of your `while` loop, after you've updated the clock, add the following lines in order to update the position of the arrow and the axis of the arrow.

```
forcearrow.pos=cart.pos
forcearrow.axis=scale*Fnet
```

9. Run your program.

Lab Report

C Complete the experiment and report your answers for the following questions.

1. Does the simulation behave like a real fancart?
2. Though the velocity of the cart changes as it moves, does the force change or is the force constant?
3. Does the acceleration of the cart change or is the car's acceleration constant?
4. When the cart passes $x = 0$, turn off the fan (i.e. set the net force on the cart to zero). Describe the resulting motion of the cart. What is the velocity of the cart after the fan turns off?

B Do all parts for **C** do the following.

1. Add a second arrow that represents the velocity of the cart. Update its position and its axis. Give it an appropriate scale.

A Do all parts for **C** and **B** and do the following.

1. Add keyboard interactions that allow the user to make the force zero (i.e. turn off the fan), turn on a constant force to the right, or turn on a constant force to the left. In all of these cases, the arrow should indicate the state of the fan.
2. Check that your code results in correct motion. Compare to how a real fan cart would behave. Describe what you did to test your code, and describe what observations you made that convince you that it works correctly. Your description of the motion of the cart should be accompanied by pictures with force and velocity arrows.

14 GAME – Lunar Lander

Apparatus

Computer
VPython – www.vpython.org

Goal

The purpose of this activity is to create a Lunar Lander game where you have to land the lunar module on the moon with as small a speed as possible and as quickly as possible. If the speed is too high, it crashes. If it takes you forever, then you run out of fuel.

Procedure

In the previous simulation that you wrote, you learned how to model the motion of an object on which the net force was constant. In that case, it was a fancart. You learned how to apply Newton's second law to update the velocity of an object given the net force on the object. Once you can do this, you can model the motion of *any* object. As a reminder, the important steps are to:

1. calculate the net force (though it will be constant in this case)
2. update the velocity of the cart
3. update the position of the cart
4. update the clock (*this is not necessary but is often convenient*)

for each iteration of the loop. The net force may not be constant. For example, you can check for keyboard interactions and turn a force on or off, or the net force might depend on direction of motion (such as friction) or speed (such as drag) or position (such as gravitational force of a star on a planet). This is why you have to calculate the net force during each iteration of the loop.

To develop a lunar lander game, we are going to begin with a bouncing ball that makes an elastic collision with the floor.

A bouncing ball

1. Here is a template for a program that simulates a bouncing ball. **However, a few essential lines are missing.** Type the template below.

```
from visual import *

scene.range=20

ground = box(pos=vector(0,-10.05,0), size=(40.0,1,1), color=color.white)
ball = sphere(pos=(0,9,0), radius=2, color=color.yellow)

ball.m = 1
ball.v = vector(0,0,0)
```

```
10 g=vector(0,-10,0)
11
12 dt = 0.01
13 t = 0
14
15 scale=0.5
16 FgravArrow = arrow(pos=ball.pos, axis=scale*ball.m*g, color=color.red)
17
18 scene.mouse.getclick()
19
20 while 1:
21         rate(100)
22 #         Fgrav=
23         Fnet=Fgrav
24 #         ball.v =
25 #         ball.pos =
26         if(ball.pos.y-ball.radius < ground.pos.y+ground.height/2):
27                 ball.v=-ball.v
28         t = t+dt
29         FgravArrow.pos=ball.pos
30         FgravArrow.axis=scale*Fgrav
```

Line 10 defines a vector $\vec{g}$. What is this vector called? What is its direction, and what is its magnitude?

What is the purpose of lines 26 and 27?

2. The commented lines must be completed for the program to work. Complete each of these lines and run your program.

If line 26 was changed to `if(ball.pos.y < ground.pos.y):` , what would occur and why is this worse than the original version of line 26? (You should comment out line 26 and type this new code in order to check your answer.)

Is the gravitational force on the ball constant or does it change? Explain your answer.

3. The Moon has a gravitational field that is 1/6 that of Earth. Change $\vec{g}$ to model the motion of a bouncing ball on the Moon and re-run your program.

What is the difference in the motion of a ball dropped from a height h on the Moon and a ball dropped from a height h on Earth?

Moon Lander

4. We will now model the motion of a lunar module. Start a new file. Type the following template. Fill in the missing (commented out) lines. Run your program.

```
from visual import *

scene.range=20

ground = box(pos=vector(0,-10.05,0), size=(40.0,1,1), color=color.white)
spaceship = box(pos=vector(0,8,0), size=(2,5,2), color=color.yellow)

spaceship.m = 1
spaceship.v = vector(0,0,0)
g=1/6*vector(0,-10,0)

dt = 0.01
t = 0

scale=5.0
FgravArrow = arrow(pos=spaceship.pos, axis=scale*spaceship.m*g, color=
    color.red)

while 1:
        rate(100)
#         Fgrav=
#         Fnet=
#         spaceship.v =
#         spaceship.pos =
        if(spaceship.pos.y-spaceship.height/2<ground.pos.y+ground.height
            /2):
                print("spaceship has landed")
                break
        t = t+dt
        FgravArrow.pos=spaceship.pos
        FgravArrow.axis=scale*Fgrav
```

5. We are now going to add a force of thrust due to rocket engines. Before the `while` loop, define a thrust force.

```
Fthrust=vector(0,4,0)
```

6. After defining the thrust vector, create another arrow that will represent the thrust force. Call it `FthrustArrow` as shown.

```
FthrustArrow = arrow(pos=spaceship.pos, axis=Fthrust, color=color.cyan)
```

7. In the while loop, change the net force so that it is the sum of the gravitational force and the thrust of the rocket engine.

```
Fnet=Fgrav+Fthrust
```

8. Also, in the while loop, update the thrust arrow's position and axis.

```
FthrustArrow.pos=spaceship.pos
FthrustArrow.axis=scale*Fthrust
```

9. Run your program and verify that the motion of the spaceship is what we expect from Newton's second law.

Change the thrust to 10/6 N (in the $+y$ direction. Describe the motion. Is this consistent with Newton's second law?

10. Let's use the keyboard to turn on and off the engine. In this case, "on" means that the thrust is non-zero and "off" means that the thrust is zero. In the while loop, after calculating `Fgrav` , type the following `if` statement:

```
if scene.kb.keys:
    k = scene.kb.getkey()
    if k == "up":
        Fthrust=vector(0,4,0)
    else:
        Fthrust=vector(0,0,0)
```

What key is used to turn the thrust on? What key is used to turn the thrust off?

11. Run your program and verify that it works.

Analysis

C Complete this exercise and do the following.

1. Print the speed of the spaceship and the clock reading when it lands.

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

1. If the speed of the spaceship is greater than 1 m/s, print "You lose."
2. If the speed of the spaceship is less than 1m/s, print "You win."

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

1. Use the down arrow to turn off the thrust (instead of any key).
2. Create an engine that fires in the $+x$ direction (the engine is on the left so the arrow points to the right). When the right arrow key is pressed, this engine turns on. When the left arrow key is pressed, this engine turns off.
3. Create an engine that fires in the $-x$ direction (the engine is on the right so the arrow points to the left). When the "a" key is pressed, this engine turns on. When the "d" key is pressed, this engine turns off.
4. Place a target on the ground.
5. Check that the lunar module lands on the target.
6. Check that the x-velocity is very small (perhaps less than 1 m/s for example) when the spaceship hits the target and print "You win" if and only if the spaceship has a very small x-velocity.
7. Since you don't want to waste fuel, assign points based on the time elapsed and fail the player if the clock reading exceeds some amount.
8. Test your game with other users.

15 GAME – Tank Wars

Apparatus

Computer
VPython – www.vpython.org

Goal

The purpose of this activity is to create a Tank Wars game where you will move a tank and adjust the launch angle and launch speed to hit a target.

Procedure

1. Download the program *tank-wars-template.py* from our course web site. For your reference, the program is printed below.

```
from visual import *

def rad(degrees): #converts an angle in degrees to an angle in radians
    radians=degrees*pi/180
    return radians

scene.range=20
scene.width=800
scene.height=800

#create objects
ground = box(pos=vector(0,-15,0), size=(40.0,1,2), color=color.green)
tank = box(pos=vector(-18,-13,0), size=(2,2,2), color=color.yellow)
turret = cylinder(pos=tank.pos, axis=(0,0,0), radius=0.5, color=tank.color
    )
turret.pos.y=turret.pos.y+tank.height/2
angleBar = cylinder(pos=vector(-18,-19,0), axis=(1,0,0), radius=1, color=
    color.magenta)
speedBar = cylinder(pos=vector(5,-19,0), axis=(1,0,0), radius=1, color=
    color.cyan)

#turret
theta=rad(45)
dtheta=rad(1)
L=3
turret.axis=L*vector(cos(theta),sin(theta),0)

#bullets
bulletsList=[]
m=1
muzzlespeed=15
```

```
29 dspeed=1
30
31 #Bar
32 angleBar.axis=(5*theta)*vector(1,0,0)
33 speedBar.axis=(muzzlespeed/2+0.5)*vector(1,0,0)
34
35 #motion
36 g=vector(0,-10,0)
37 dt = 0.01
38 t = 0
39
40 while 1:
41     scene.mouse.getclick()
42     while 1:
43         rate(100)
44         if scene.kb.keys:
45             k = scene.kb.getkey()
46             if k == "up":
47                 theta=theta+dtheta
48                 turret.axis=L*vector(cos(theta),sin(theta),0)
49                 angleBar.axis=(5*theta)*vector(1,0,0)
50             elif k == "down":
51                 theta=theta-dtheta
52                 turret.axis=L*vector(cos(theta),sin(theta),0)
53                 angleBar.axis=(5*theta)*vector(1,0,0)
54             elif k == "left":
55                 muzzlespeed=muzzlespeed-dspeed
56                 speedBar.axis=(muzzlespeed/2+0.5)*vector(1,0,0)
57             elif k == "right":
58                 muzzlespeed=muzzlespeed+dspeed
59                 speedBar.axis=(muzzlespeed/2+0.5)*vector(1,0,0)
60             elif k=="␣":
61                 bullet=sphere(pos=turret.pos+turret.axis, radius=0.5,
                        color=color.white)
62                 bullet.v=muzzlespeed*vector(cos(theta),sin(theta),0)
63                 bulletsList.append(bullet)
64             elif k=="Q":
65                 break
66
67         if muzzlespeed>20:
68             muzzlespeed=20
69         if muzzlespeed<1:
70             muzzlespeed=1
71         if theta>rad(90):
72             theta=rad(90)
73         if theta<0:
74             theta=0
75
76         for thisbullet in bulletsList:
77             if(thisbullet.pos.y<ground.y+ground.height/2):
78                 thisbullet.Fnet=vector(0,0,0)
79                 thisbullet.v=vector(0,0,0)
80             else:
81                 thisbullet.Fnet=m*g
```

```
82  #                thisbullet.v=
83  #                thisbullet.pos=
84
85              t=t+dt
```

2. Read the program above and answer the following questions.

What should lines 82 and 83 be in order to correctly update a bullet's velocity and position for each time step?

Is the world in this simulation Earth? Cite a particular line number in the code in order to support your answer.

What does the `rad()` function do?

What does the variable L tell you?

What is the variable name for the initial speed of the bullet? How much does the launch speed of the bullet change when you press the right or left arrow key?

What is the variable name for the angle the bullet is launched at? How much does the angle change when you press the right or left arrow key and is the unit radians or degrees?

After a bullet hits the ground, what is the net force on the bullet and what is its velocity? Cite the particular line numbers that support your answer.

What key strokes are used to change the launch angle?

What keystrokes are used to change the launch speed?

What is the maximum and minimum launch speeds allowed? Which line numbers tell you this?

What is the maximum and minimum angles allowed? Which line numbers tell you this?

3. Edit lines 82 and 83 so that they use correct physics to update the velocity and position of each bullet and run the program.

   The reason that there are nested `while` loops is that you are going to create a target, and after a bullet hits the target, you may want to pause the game, reset certain variables, add a barrier, count

a score, etc. The inner while loop handles the animation and the outer while loop is where you'll do other things after the target is hit. At this point, the outer loop only runs one iteration and only pauses the program, waiting for a mouse click.

We are now going to add a few features to the game.

Creating a Target

4. Create a target that is a box named `target` that is the size of the tank and place it somewhere else on the terrain. Run your program to see the target.

   We will need to check for collisions between a bullet and the target. It is convenient to create a function that does the math to see if the bullet (sphere) and target (box) overlap. If they do overlap, it returns `True` . If they do not overlap, the it returns `False` .

5. At the top of your program near where the `rad()` function is defined, add the following function. You do not have to type the commented lines. You will need to double check your typing to make sure you do not have any typos. The condition of the `if` statement is rather long, so check it for accuracy.

```
#determines whether a sphere and box intersect or not
#returns boolean
def collisionSphereAndBox(sphereObj, boxObj):
    result=True
    if((sphereObj.pos.x-sphereObj.radius<boxObj.pos.x+boxObj.length/2 and
        sphereObj.pos.x+sphereObj.radius>boxObj.pos.x-boxObj.length/2) and
         (sphereObj.pos.y-sphereObj.radius<boxObj.pos.y+boxObj.height/2
        and sphereObj.pos.y+sphereObj.radius>boxObj.pos.y-boxObj.height/2)
        ):
         result=True
         dist=sqrt((sphereObj.pos.x-boxObj.pos.x)**2+(sphereObj.pos.y-
             boxObj.pos.y)**2)
         if(dist>sphereObj.radius+sqrt((boxObj.length/2)**2+(boxObj.width
             /2)**2)):
              result=False
        return result
    else:
         result=False
    return result
```

6. At the `if` statement at line 77 in the above printout where the program checks whether the bullet hits the ground, add an `elif` statement (between the if and else) to check whether the bullet collides with the target. It should look like this.

```
elif collisionSphereAndBox(thisbullet,target):
    thisbullet.Fnet=vector(0,0,0)
    thisbullet.v=vector(0,0,0)
    print("direct hit!")
    break
```

7. Run your program. Fire a bullet that hits the target and observe the console after the collision. You should see it printing "direct hit!" over and over.

Why does it continuously print "direct hit!" instead of printing it just one time?

We need a boolean variable that keeps track of whether the target is hit or not. If it is hit, then we will set this boolean to True and break out of the inner while statement. In fact, we will make this the condition of the inner while statement. We will only go through the inner while loop as long as the target is not hit.

8. Before the outer `while` loop, add the following code:

```
hit = False
```

This variable `hit` will be False until the target is hit. After the target is hit, we will set the variable to `True`.

9. Change the condition of the inner while loop to read:

```
while hit==False:
```

It is no longer an infinite loop. It only runs while `hit=False`. If `hit` is chagned to `True`, then the inner loop will end, and the outer loop will iterate.

10. Inside the `elif` statement where you check for a collision between the bullet and target and print "direct hit!", set `hit=True` before the `break` statement.

11. Run your program.

After the target is hit, the program seems to stop. The reality is that it is stuck in the infinite outer loop and never running back through the inner loop. Why?

12. We would like to do three things before running the inner loop again (or after–it doesn't matter). These three things effectively resets the game back to the initial state.

    (a) Set the hit variable to False.

    (b) Erase all of the bullets in the scene.

    (c) Reset the bulletsList to an empty list.

Add the following code before the inner `while` loop and after the `getclick()` statement.

```
hit = False
for thisbullet in bulletsList:
    thisbullet.visible=False
bulletsList=[]
```

These three lines do the three things outlined above to reset the game back to the initial conditions. (Technically we haven't completely reset it. It might be nice to reset the launch speed and launch angle back to their initial values for example.)

13. Run your program. Verify that it works as expected.

Analysis

C Complete this exercise and do the following.

1. Print the launch speed of the bullet, the launch angle of the bullet, and the x-position of the bullet (called the range) when the bullet hits the target or ground.
2. Change the maximum speed of the bullet to 30.
3. Set the initial launch angle to 60 degrees (do this in the code) and find the launch speed necessary to hit the target if the target is at $x = 18$. Set the initial launch speed to this value in the code so that when your code is run for the first time, the projectile will be launched at 60 degrees and will hit the target which is at $x = 18$.

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

1. Add a key stroke that will move the tank left and right, but do not allow the tank to go past the center of the screen or off the left side of the screen.

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

1. Create a barrier (a box) that sits between the tank and the target. If a bullet hits the barrier, it will stop just it stops when it hits the ground. Remember to use your `collisionSphereAndBox()` function to check for a collision between the bullet and barrier.
2. Create different levels of the game. Create a variable called `level` that is an integer from 1–5. In the first level, there is no barrier. In the second level, there is a barrier. For other levels, place the tank at a different height or the target at a different height, for example. After getting to the fifth level, either end the game or go back to the first level.

16 GAME – Asteroids

Apparatus

Computer
VPython – www.vpython.org

Goal

The purpose of this activity is to modify an asteroids game so that when a large asteroid breaks into two smaller asteroids, the center of mass continues with the same velocity, as you observed in the experiment of the colliding pucks.

Procedure

Playing Asteroids

1. Play the game *Asteroids*. A link is available from our course web site. Note the actions of the up, down, left, and right keystrokes and how they affect the motion of the spaceship.
2. Download the file *asteroids-template.py* from our course web site, and open the file in VPython.
3. Run the program. Note how the up, down, left, and right keystrokes affect the motion of the spaceship. It's different than in the other Asteroids game. Also, note that hitting an asteroid with a bullet does not make it break up into pieces. The program is printed below for reference.

```
from visual import *
import random

#converts an angle in degrees to an angle in radians
def rad(degrees):
    radians=degrees*pi/180
    return radians

#pause and wait for mouse or keyboard event, then continue
def pause():
    while True:
        rate(50)
        if scene.mouse.events:
            m = scene.mouse.getevent()
            if m.click == 'left': return
        elif scene.kb.keys:
            k = scene.kb.getkey()
            return

#checks for a collision between two spheres
def collisionSpheres(sphere1, sphere2):
    dist=mag(sphere1.pos-sphere2.pos)
    if(dist<sphere1.radius+sphere2.radius):
        return True
```

```
25      else:
26          return False
27
28  #checks for a collision between a cone and a sphere
29  def collisionConeSphere(c, s):
30
31      #result is the variable that we will return
32      #default is False
33      result=False
34
35      #check pos of cone
36      if(collisionSphereAndPoint(s,c.pos)):
37          result=True
38      #check tip of cone
39      result=False
40      tip=c.pos+c.axis
41      if(collisionSphereAndPoint(s,tip)):
42          result=True
43      #check edge of radius in x-y plane 1
44      r1=c.radius*cross(vector(0,0,1),norm(c.axis))
45      if(collisionSphereAndPoint(s,r1+c.pos)):
46          result=True
47      #check edge of radius in x-y plane 2
48      r2=-c.radius*cross(vector(0,0,1),norm(c.axis))
49      if(collisionSphereAndPoint(s,r2+c.pos)):
50          result=True
51
52      #return result
53      return result
54
55  #determines whether a point is within a sphere or not
56  #returns boolean
57  def collisionSphereAndPoint(sphereObj, targetVector):
58      dist=mag(sphereObj.pos-targetVector)
59      if(dist<sphereObj.radius):
60          return True
61      else:
62          return False
63
64  #creates four asteroids, one on each side of the scene
65  def createAsteroids():
66
67      #asteroid comes from the right
68      asteroid=sphere(pos=vector(20,0,0), radius=1, color=color.cyan)
69      asteroid.pos.y=random.randrange(-20,20,5)
70      asteroid.m=1
71      asteroid.v=vector(0,0,0)
72      asteroid.v.x=-random.randint(1,5)
73      asteroid.v.y=random.choice((1,-1))*random.randint(1,5)
74      asteroidList.append(asteroid)
75
76      #asteroid comes from the left
77      asteroid=sphere(pos=vector(-20,0,0), radius=1, color=color.cyan)
78      asteroid.pos.y=random.randrange(-20,20,5)
```

```
79      asteroid.m=1
80      asteroid.v=vector(0,0,0)
81      asteroid.v.x=random.randint(1,5)
82      asteroid.v.y=random.choice((1,-1))*random.randint(1,5)
83      asteroidList.append(asteroid)
84
85      #asteroid comes from the top
86      asteroid=sphere(pos=vector(0,20,0), radius=1, color=color.cyan)
87      asteroid.pos.x=random.randrange(-20,20,5)
88      asteroid.m=1
89      asteroid.v=vector(0,0,0)
90      asteroid.v.x=random.choice((1,-1))*random.randint(1,5)
91      asteroid.v.y=-random.randint(1,5)
92      asteroidList.append(asteroid)
93
94      #asteroid comes from the bottom
95      asteroid=sphere(pos=vector(0,-20,0), radius=1, color=color.cyan)
96      asteroid.pos.x=random.randrange(-20,20,5)
97      asteroid.m=1
98      asteroid.v=vector(0,0,0)
99      asteroid.v.x=random.choice((1,-1))*random.randint(1,5)
100     asteroid.v.y=random.randint(1,5)
101     asteroidList.append(asteroid)
102
103 #scene size
104 scene.range=20
105 scene.width=700
106 scene.height=700
107
108 #create the spaceship as a cone
109 spaceship = cone(pos=(0,0,0), axis=(2,0,0), radius=1, color=color.white)
110 fire = cone(pos=(0,0,0), axis=-spaceship.axis/2, radius=spaceship.radius
        /2, color=color.orange)
111
112 #initial values for mass, velocity, thrust, and net force
113 spaceship.m=1
114 spaceship.v=vector(0,0,0)
115 thrust=0
116 Fnet=vector(0,0,0)
117
118 #bullets
119 bulletspeed=10
120 bulletsList=[]
121
122 #angle to rotate
123 dtheta=rad(10)
124
125 #clock
126 t=0
127 dt=0.005
128
129 #asteroids
130 Nleft=0 #counter for number of asteroids left in the scene
131 asteroidList=[]
```

```
createAsteroids()

while spaceship.visible==1:
    rate(200)

    if scene.kb.keys:
        k = scene.kb.getkey()
        if k == "up": #turn thruster on
            thrust=6
        elif k=="left": #rotate left
            spaceship.rotate(angle=-dtheta, axis=(0,0,-1));
        elif k=="right": #rotate right
            spaceship.rotate(angle=dtheta, axis=(0,0,-1));
        elif k==" ": #fire a bullet
            bullet=sphere(pos=spaceship.pos+spaceship.axis, radius=0.1,
                color=color.yellow)
            bullet.v=bulletspeed*norm(spaceship.axis)+spaceship.v
            bulletsList.append(bullet)
        elif k=="q": #pause the game
            pause()
        else: #turn thruster off
            thrust=0

    Fnet=thrust*norm(spaceship.axis)
    spaceship.v=spaceship.v+Fnet/spaceship.m*dt
    spaceship.pos=spaceship.pos+spaceship.v*dt
    fire.pos=spaceship.pos
    fire.axis=-spaceship.axis/2


    #check if the spaceship goes off screen and wrap
    if spaceship.pos.x>20 or spaceship.pos.x<-20:
        spaceship.pos=spaceship.pos-spaceship.v*dt
        spaceship.pos.x=-spaceship.pos.x
    if spaceship.pos.y>20 or spaceship.pos.y<-20:
        spaceship.pos=spaceship.pos-spaceship.v*dt
        spaceship.pos.y=-spaceship.pos.y

    #update positions of bullets and check if bullets go off screen
    for thisbullet in bulletsList:
        if thisbullet.pos.x>20 or thisbullet.pos.x<-20:
            thisbullet.visible=0
        if thisbullet.pos.y>20 or thisbullet.pos.y<-20:
            thisbullet.visible=0
        if thisbullet.visible != 0:
            thisbullet.pos=thisbullet.pos+thisbullet.v*dt

    #update positions of asteroids
    for thisasteroid in asteroidList:
        if thisasteroid.visible==1:
            thisasteroid.pos=thisasteroid.pos+thisasteroid.v*dt
            #check for collision with spaceship
            if(collisionConeSphere(spaceship,thisasteroid)):
                spaceship.visible=0
```

```
185                 fire.visible=0
186             #wrap at edge of screen
187             if thisasteroid.pos.x>20 or thisasteroid.pos.x<-20:
188                 thisasteroid.pos=thisasteroid.pos-thisasteroid.v*dt
189                 thisasteroid.pos.x=-thisasteroid.pos.x
190             if thisasteroid.pos.y>20 or thisasteroid.pos.y<-20:
191                 thisasteroid.pos=thisasteroid.pos-thisasteroid.v*dt
192                 thisasteroid.pos.y=-thisasteroid.pos.y
193             #check for collision with bullets
194             for thisbullet in bulletsList:
195                 if(collisionSpheres(thisbullet,thisasteroid)and thisbullet
                        .visible==1):
196                     thisasteroid.visible=0
197                     thisbullet.visible=0
198
199     Nleft=0 #have to reset this before counting
200     for thisasteroid in asteroidList:
201         if thisasteroid.visible:
202             Nleft=Nleft+1
203
204     #create more asteroids if all are gone
205     if Nleft==0:
206         createAsteroids()
207
208     #update fire
209     if thrust==0:
210         fire.visible=0
211     else:
212         fire.visible=1
213
214     t=t+dt
```

4. Answer the following questions.

   (a) What is the magnitude of the thrust of the engine when it is firing?

   (b) What line number calculates the net force on the spaceship?

   (c) What line numbers update the velocity and position of the spaceship?

   (d) What line numbers update the positions of the asteroids?

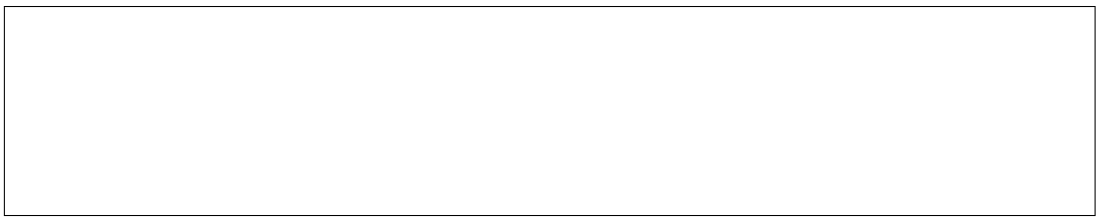
   (e) What line numbers update the positions of the bullets?

   (f) What line number makes the asteroid disappear when it is hit by a bullet?

   (g) How many asteroids are created when the function `createAsteroids()` is called and where do these asteroids come from?

   (h) The velocities of the asteroids are randomized. What is the maximum possible velocity of an asteroid? What is the minimum possible velocity of an asteroid? (Note that the directions are randomized as well. By "maximum velocity" I am referring to the maximum absolute value of the x and y components of the velocity vector. By "minimum velocity", I am referring to the minimum absolute value of the x and y components of the velocity vector.)

   (i) If you decrease the mass of the spaceship (i.e. change it from 1 to a smaller number like 0.5), how would it affect the spaceship's motion when the engine is firing? Answer the question, then test your answer to see if it is correct, and then comment on whether your observation matched your prediction.



Adding Asteroids Fragments

In the last chapter, you learned that the center of mass of a two-body system is constant. Whether it is a collision of two objects or an exploding fireworks shell, during the collision or explosion, the center of mass of the system is the same.

In the classic Asteroids game, when a large asteroid is shot with a bullet, it breaks into two fragments. Since the center of mass of the system must remain constant then

$$\vec{v}_{ast} = \frac{m_1\vec{v}_1 + m_2\vec{v}_2}{m_1 + m_2}$$

Define the total mass of the fragments as $M = m_1 + m_2$, which is the mass of the asteroid before the explosion. Then,

$$\vec{v}_{ast} = \frac{m_1\vec{v}_1 + m_2\vec{v}_2}{M}$$

In our game, let's assume that the asteroid breaks into two equal mass fragments that are each 1/2 the total mass of the asteroid. Then, $m_1 = m_2 = 1/2M$. Thus,

$$\begin{aligned}\vec{v}_{ast} &= \frac{1/2M\vec{v}_1 + 1/2M\vec{v}_2}{M} \\ \vec{v}_{ast} &= \frac{\vec{v}_1 + \vec{v}_2}{2}\end{aligned}$$

In other words, since the fragments have equal masses, then the asteroid's velocity is the arithmetic mean of the velocities of the fragments (i.e. the sum of their velocities divided by 2).

In our game, we will randomly assign the velocity of fragment 1 so that it will shoot off with a random speed in a random direction. Then we will calculate the velocity of fragment 2 so that

$$\vec{v}_2 = 2\vec{v}_{ast} - \vec{v}_1$$

5. Above the `while` loop where the asteroidList is created, you need to create a list called `fragmentList`. To do this, type the following line:

   ```
   fragmentList=[]
   ```

6. Create a function that will create the fragments when an asteroid is hit. Type the following code and replace the commented line with the correct calculation for the velocity of fragment 2.

```
def createFragments(asteroid):
    fragment1=sphere(pos=asteroid.pos, radius=0.5, color=color.magenta)
    fragment2=sphere(pos=asteroid.pos, radius=0.5, color=color.magenta)
    fragment1.m=0.5
    fragment2.m=0.5
    fragment1.v=vector(0,0,0)
    fragment1.v.x=random.choice((1,-1))*random.randint(1,5)
    fragment1.v.y=random.choice((1,-1))*random.randint(1,5)
#     fragment2.v=
    fragmentList.append(fragment1)
    fragmentList.append(fragment2)
```

7. In the block of code that checks for a collision between a bullet and asteroid, call the `createFragments` function using:

```
createFragments(thisasteroid)
```

8. You will need to update the positions of the fragments (i.e. make them move). It's easiest to copy and paste the section of code for the asteroids and change the variables names to the appropriate names for the fragments. Here's an example of what it should look like:

```
#update positions of fragments
for thisfragment in fragmentList:
    if thisfragment.visible==1:
        thisfragment.pos=thisfragment.pos+thisfragment.v*dt
        #check for collision with spaceship
        if(collisionConeSphere(spaceship,thisfragment)):
            spaceship.visible=0
            fire.visible=0
        #wrap at edge of screen
        if thisfragment.pos.x>20 or thisfragment.pos.x<-20:
            thisfragment.pos=thisfragment.pos-thisfragment.v*dt
            thisfragment.pos.x=-thisfragment.pos.x
        if thisfragment.pos.y>20 or thisfragment.pos.y<-20:
            thisfragment.pos=thisfragment.pos-thisfragment.v*dt
            thisfragment.pos.y=-thisfragment.pos.y
        #check for collision with bullets
        for thisbullet in bulletsList:
            if(collisionSpheres(thisbullet,thisfragment)and thisbullet
                .visible==1):
                thisfragment.visible=0
                thisbullet.visible=0
```

9. The section of code that counts how many asteroids are left also needs to count how many fragments are left. After the `for` loop that counts the asteroids that are left, add a second for `loop` that counts the remaining fragments.

```
for thisfragment in fragmentList:
    if thisfragment.visible:
        Nleft=Nleft+1
```

10. That should do it. Run your code and fix any errors.

11. Note that we could have written our code more efficiently. Whenever you find that you are copying and pasting the same code over and over, you might want to consider putting it into a function and calling the function. For example, we have to wrap the motion of the spaceship, asteroids, and fragments. We

can create a `wrap()` function that takes an argument of the object (like spaceship) and checks to see if it is off screen. If it is off screen, it wraps the position of the object. If you notice ways like this to reduce the number of lines of code, feel free to make those changes.

Analysis

C Complete this exercise and do the following.

1. Add a point system that tallies points whenever an asteroid is shot.
2. Add a keystroke for the "down" arrow key that sets the thrust to a negative value (such as `thrust=-6`).
3. Print the total number of points at the end of the game when the spaceship collides with an asteroid.

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

1. Place the entire `while spaceship.visible==1:` loop inside another `while 1:` loop so that after the spaceship collides with an asteroid the program will not end, but rather it will begin again.
2. You will need to reinitialize your variables for the velocity, thrust, and net force on the spaceship, along with your lists, in order to reset the game. You can copy code like the following and paste it just after the `while 1:` statement.

```
spaceship.visible=1
spaceship.v=vector(0,0,0)
thrust=0
Fnet=vector(0,0,0)

#bullets
bulletsList=[]

#asteroids
Nleft=0 #counter for number of asteroids left in the scene
asteroidList=[]
createAsteroids()

#fragments
fragmentList=[]
```

3. Call the `pause()` function after resetting your variables and lists and before the `while spaceship.visible==1:` statement.
4. Allow a total of 3 lifetimes for the spaceship and end the program after the third one.

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

1. Add fragmentation for the asteroids so that the physics is correct and all fragments have to be shot before the game is reset.