PHY 1200: Physics for Video Games (with GlowScript)

Notes, Labs, and Programs

Aaron Titus

High Point University

Spring 2016

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.

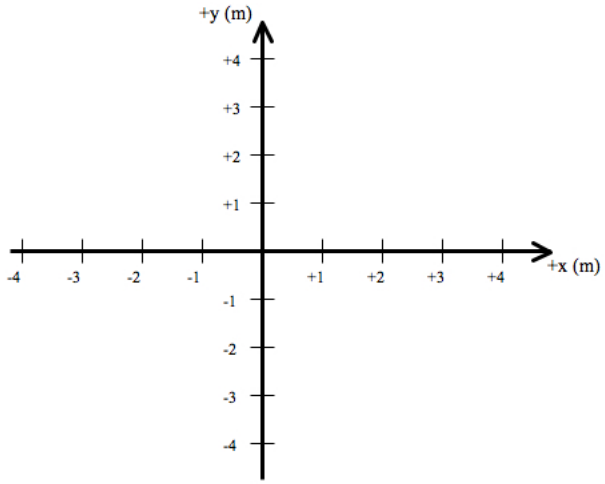


Figure 1.1: A 2-D Cartesian coordinate system.

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.

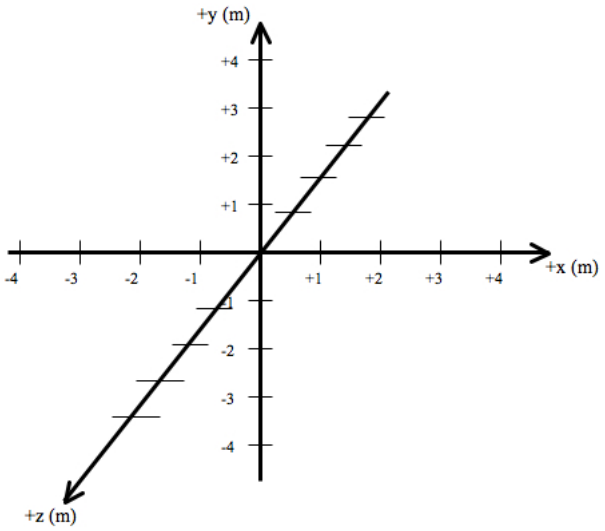


Figure 1.2: A 3-D Cartesian coordinate system.

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.

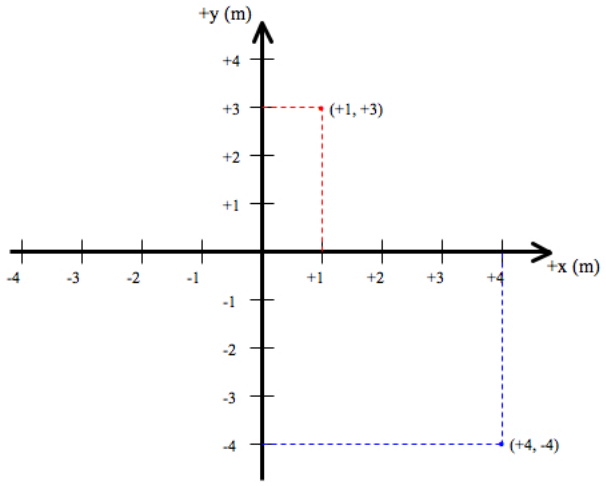


Figure 1.3: Coordinates of certain points on a coordinate system.

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.

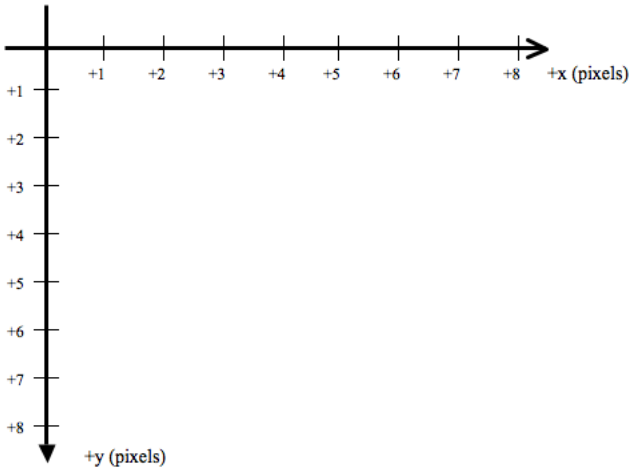


Figure 1.4: Convention for pixel coordinates on a computer monitor.

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.

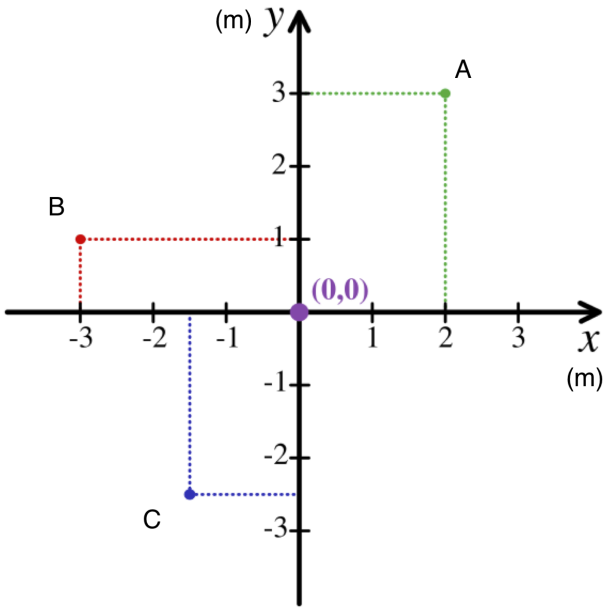


Figure 1.5: Three points on a Cartesian coordinate system.

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?

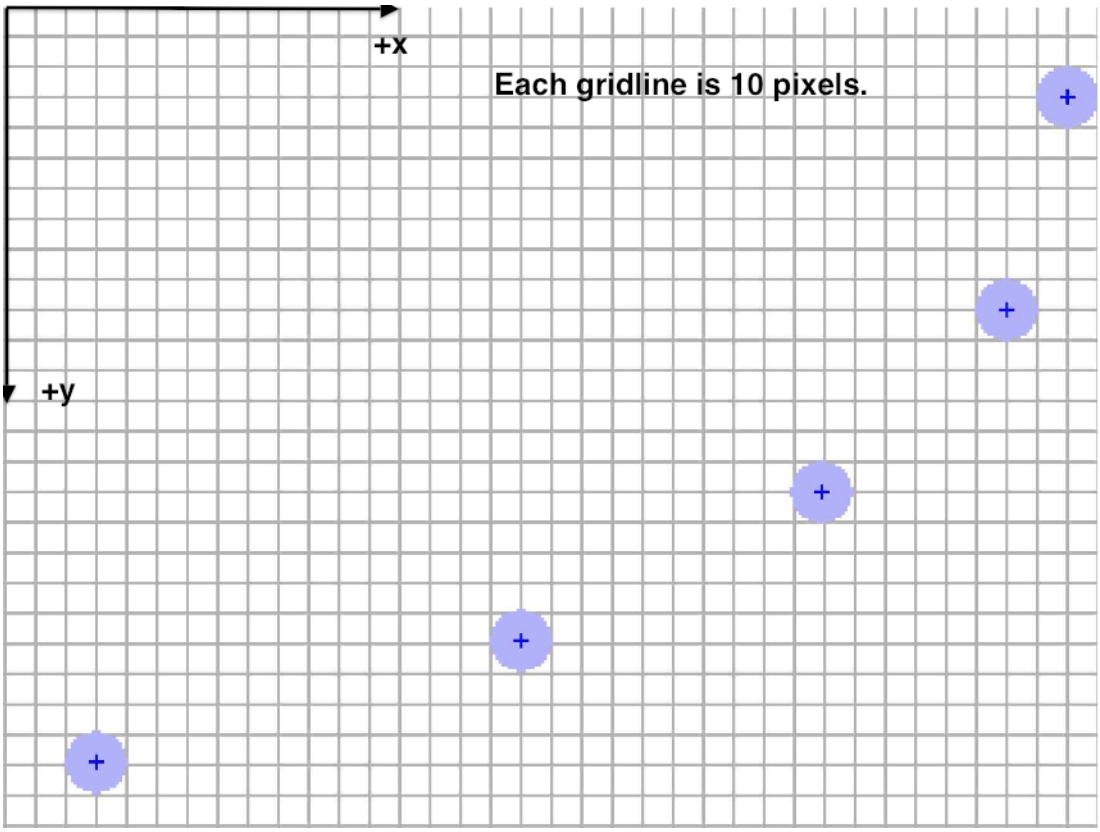


Figure 1.6: A puck on a computer monitor.

2 LAB: Coordinates

Apparatus

meterstick or tape measure
graph paper

Goal

To measure the coordinates of objects in the classroom.

Procedure

1. Define the origin of our coordinate system to be at the left corner of the front of the room (if you are facing toward the front), at the floor.
2. Define the $+x$ direction to be to the right, if facing the front of the room.
3. Define the $+y$ direction to be upward toward the ceiling.
4. Define the $+z$ direction to be along the side wall toward the back of the room.
5. Using a meterstick, determine the coordinates of the center of the seat of your chair.

Further Investigation

C What are the coordinates of the center of your seat?

B (Do C first.) Using graph paper, draw an x-z coordinate system with the origin in the top-left corner of the paper. (Note: it's ok to use your paper in landscape or portrait orientation depending on what is most convenient. You are sketching a top view of the room. Assume that the front wall with the whiteboard is at the top edge of the paper.) Draw the seat of your chair (as a circle) at the correct location on the coordinate system. Assume that the chair is underneath the table. Be sure to indicate the scale of your coordinate system by labeling the axes with both numbers and units. Sketch the boundaries of the room. Select the values of your gridlines so that the coordinate system takes up as much of the paper as possible.

A (Do B and C first.) Show the coordinates of every seat in the classroom. If you think carefully about it, you can make a few measurements and assumptions and then calculate and draw the positions of the chairs in the room. You do not have to measure every chair.

3 PROGRAM: Introduction to GlowScript and VPython

Apparatus

Computer
GlowScript – www.glowscript.org

Goal

The purpose of this activity is to write your first program in a language called Python. We will use the web app GlowScript that converts Python to JavaScript so the program can run in a web browser. GlowScript provides the same functions available in the Python module called Visual. Together Python and Visual are named VPython. As a result, GlowScript can be considered the web-based version of VPython. You will probably read or hear the terms GlowScript and VPython used interchangeably; however, there are some important differences. I tend to think of the language as VPython (Python + Visual) and the web app as GlowScript.

We use VPython because it allows you to do vector algebra and to create 3D objects in a 3D scene. The capability of 3D graphics with vector mathematics makes it a great tool for simulating physics phenomena. In this activity, you will learn:

- how to use GlowScript, the web-based integrated development editor (IDE) for writing and running VPython.
- how to structure a simple computer program in VPython.
- how to create 3D objects such as spheres and arrows.

Setup

Go to http://www.glowscript.org/ and create an account. You will need a Google account because GlowScript uses your Google account for authentication. After logging in, you will see a link to "your programs are here." Click this link to enter the IDE.

Procedure

Creating folders and files

1. Once you log in and follow the link to your programs, you are in the GlowScript IDE. Click the Add Folder tab to create a new folder. A pop-up window appears as shown in Figure 3.1. Because I must run your programs, make the folder public. Name it "phy1200" if you wish.
2. With the folder name highlighted orange (showing you are in the folder), click the link **Create New Program** and name the program `intro` .

Starting a program: Setup statements

3. Notice GlowScript types the first line of the program for you.

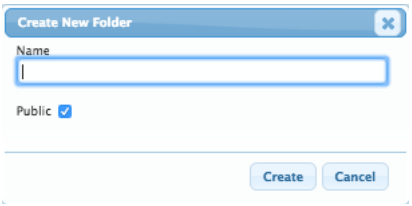


Figure 3.1: Create a new folder in GlowScript.

```
GlowScript 1.1 VPython
```

Every GlowScript program begins with this setup statement. It tells GlowScript you are writing VPython code.

4. Also, notice there is no "save" menu. Like Google Docs, GlowScript automatically saves your program as you are typing it.

Creating an object

5. Now for your first VPython command, let's make a sphere. Skip a line in order to make your code more readable, and on line 3, type:

```
sphere()
```

This statement tells the computer to create a sphere object.

6. Run the program by clicking **Run this program**. GlowScript exits the edit mode and enters the run mode. You should see a white sphere on a black background like Figure 3.2. This is called the `scene` .

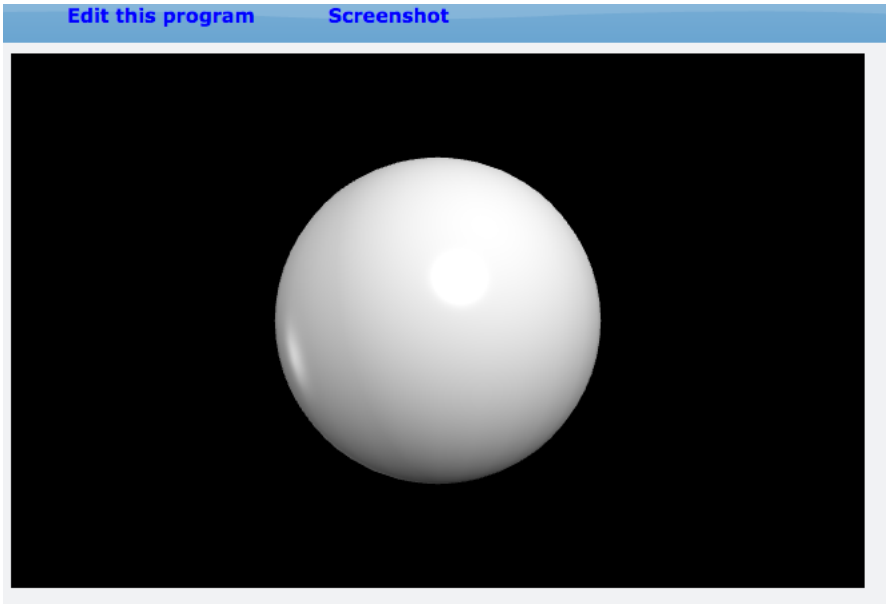


Figure 3.2: Your first VPython program–a sphere.

The 3-D graphics scene

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

7. If you are on a PC, 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. This is how you *zoom* in VPython.

8. 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 Control key while you click and drag the mouse. This is how you *rotate* the scene in VPython. Because this is a sphere, you won't notice a significant change except for lighting.

   By default, when you first run the program, the coordinate system is defined with 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 relative to the camera.

Error messages: Making and fixing an error

GlowScript tells you when there is a syntax error in your program. (Logic errors are much more difficult to fix!) To see an example of an error message, let's try making a spelling mistake.

9. Click **Edit** to return to editing mode, and change line 3 of the program to the following:

```
phere()
```

10. Run the program.

    There is no function or object in VPython called phere(). As a result, an error message pops up. The message gives the *approximate* line number where the error occurred and a description of the error, as shown in Figure 3.3.

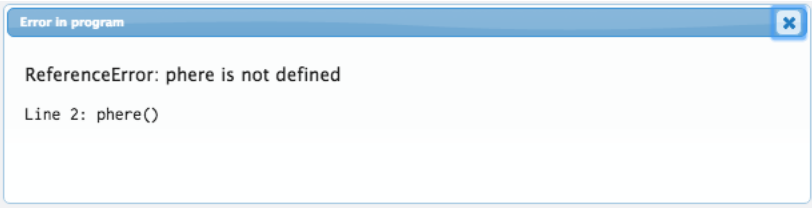


Figure 3.3: An error message in GlowScript.

The line number may be off, as it is in this case but is usually close.

11. Correct the error in the program by clicking **Edit this program** and returning to the editor. Once in editing mode, you can click the ☒ to close the error message.

    There are two types of errors: (1) syntax errors which might be a typing or coding mistake and (2) programmatic errors so the program runs correctly but does something other than what you intended. The error message helps you find the first of these. Finding errors that cause a program to act differently than you intended is much more difficult and is a skill you will develop in this course.

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

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

12. Change line 3 of the program to the following:

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

13. Run the program. Experiment with other changes to pos, radius, and color, running the program each time you change an attribute.

14. Answer the following questions:

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? You can try `color=vector(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 a 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 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 we will often create is a box. A box is defined by its position, axis, length, width, and height as shown in Figure 3.4.

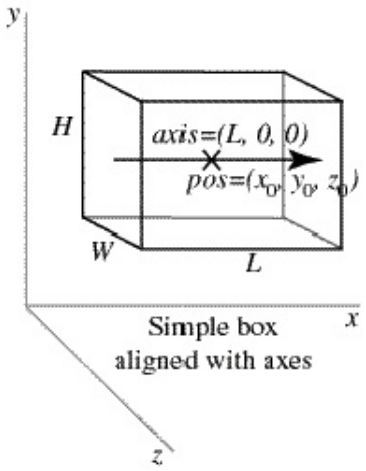


Figure 3.4: Attributes of a box. (Image from `http://www.glowscript.org/docs/VPythonDocs/box.html`)

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

```
box(pos=vector(0,0,0), size=vector(2,1,0.5), color=color.orange)
```

The length, width, and height of the box are expressed as a vector with the attribute: `size=vector(L,H,W)` .

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

17. Now change its height and rerun the program.

18. Similarly change its width and position.

> Which dimension (length, width, or height) should be changed to make a box longer along the y-axis? Change your code now to check your answer.
>
> What point does the position of the box refer to?
>
> (a) the center of the box
>
> (b) one of its corners
>
> (c) the center of one of its faces
>
> (d) 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.

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

```
#box(pos=vector(0,0,0), size=vector(2,1,0.5), color=color.orange)
```

20. Run the program. What did you observe?

21. Uncomment this 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.

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

23. Run your program and verify that it looks as expected. Use your mouse to rotate the scene so you can see the ball relative to the court. Your program should look like the one below.

```
1 GlowScript 1.1 VPython
2
3 box(pos=vector(0,0,0), size=vector(78,4,36), color=color.green)
4
5 sphere(pos=vector(-28, 5, 8), radius=1, color=color.orange)
```

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

25. Run the program.

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

```
tennisball=sphere(pos=vector(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.

27. Start a new line at the end of your program (perhaps line 7) and type:

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

28. Run the program.

29. Look at the text below the 3D scene. The printed vector should be the same as the tennis ball's position.

30. Add a new line to the end of your program (perhaps line 9) 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: 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.) This may happen so quickly that you do not notice the tennis ball drawn at the two locations.

31. Add a new line to the end of your program (perhaps line 11) and type:

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

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

32. 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

All games with graphics include objects on the screen. The game programmer must specify the positions and dimensions (sizes) of the objects using 2D or 3D vectors.

C Do all of the following. You are going to create objects for the game *Frogger*. We will only use spheres and boxes for this part.

1. Click the link to your username to return to your folders.
2. If necessary, click the phy1200 folder. Create a new blank file and name it *frogger-C*.
3. Create a green box for the frog that is at the location $< 0, -100, 0 >$, has a length=10, height=10, and width=10 units. Name the box `frog` .
4. Create a yellow sphere for a lily pad at $< -60, 100, 0 >$ with a radius of 10. Name the sphere `lilypad` .

5. Create a blue box for the water that is at the location $< 0, 0, -10 >$, has a length=150, height=220, and width=10 units. Name the box `water` .
6. Rotate the scene. Is the lily pad inside the water or on top of the water? Is the frog inside the water or on top of the water?
7. Is physics used in this program? Why does the frog not move in this program?

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

1. Return to your phy1200 folder and create a new blank file and name it *frogger-B*.
2. Copy from your previous program (labeled C) and paste it into this program. Often, this is the fastest way to start a new program.
3. Create another yellow sphere for a lily pad at $< 60, 100, 0 >$ with a radius of 10. Name the sphere `lilypad4` .
4. In between these two lily pads, create two more named `lilypad2` and `lilypad3` so the lily pads are equally spaced.
5. Create a gray road that is exactly half the height of the water. It should extend from the middle of the blue box to the bottom end of the blue box.
6. Create a long cyan box on the left side of the road and a short magenta box on the right side of the road, between the frog and the water. Name them `car1` and `car2` .
7. Print the positions of the cars and the frog.

Figure 3.5 is an example program that fits the criteria for a B.

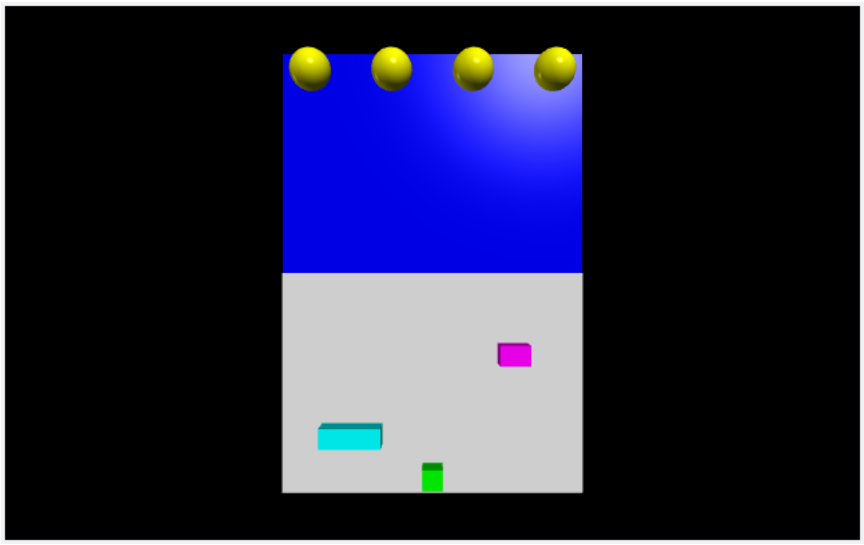


Figure 3.5: The scene required for a B.

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

1. Return to your phy1200 folder and csreate a new blank file and name it *frogger-A*.
2. Copy from your previous program (labeled B) and paste it into this program.
3. In the top right corner of the GlowScript window, click the link to **Help**. This opens the documentation window. Click the menu to **Choose a 3D object** and view the list of objects shown in Figure 3.5.
4. Select the cylinder and read how to create a cylinder.

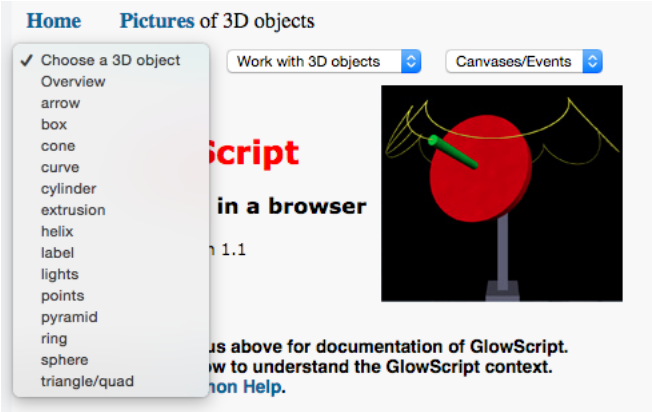


Figure 3.6: GlowScript documentation

5. Change the lily pads so they are thin cylinders that appear to float on top of the water.
6. Use the cylinder object to create 3 logs of different lengths in the water.
7. Now, click the menu to **Work with 3D objects** in the documentation and select **Materials/-Textures**. Read how to specify a texture. You will probably want to click the link to the example program that demonstrates the pre-defined textures.
8. Change the three wooden logs so they use the wood texture.

Figure 3.7 is an example program that fits the criteria for program A.

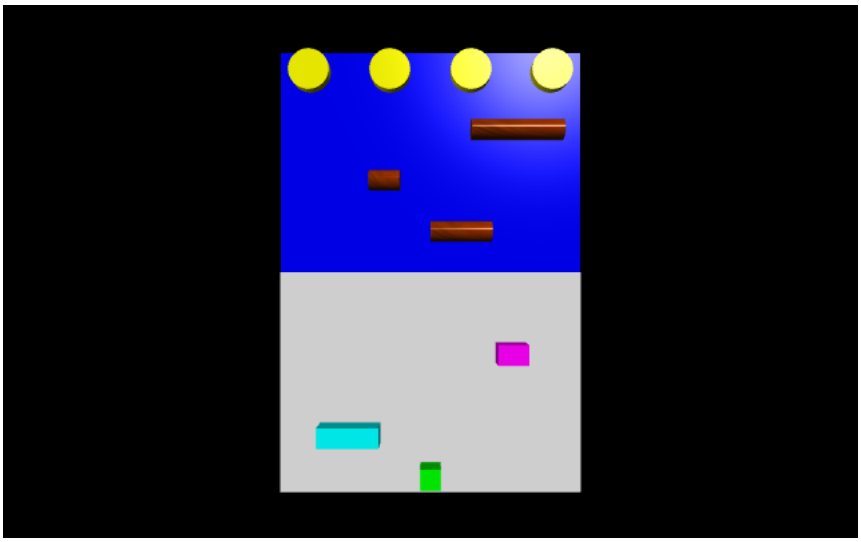


Figure 3.7: The scene required for an A.

4 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 4.1, the head of the vector is at $(+1, +4)$ m, and the tail is at $(-4, -2)$ m.

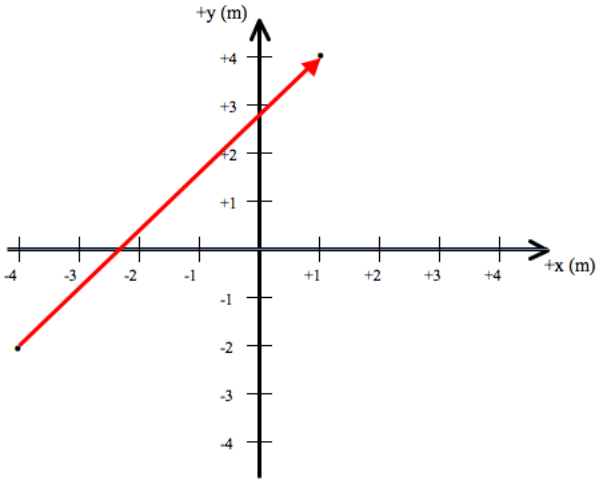


Figure 4.1: A vector is simply an arrow.

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 4.2, 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 4.2, 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

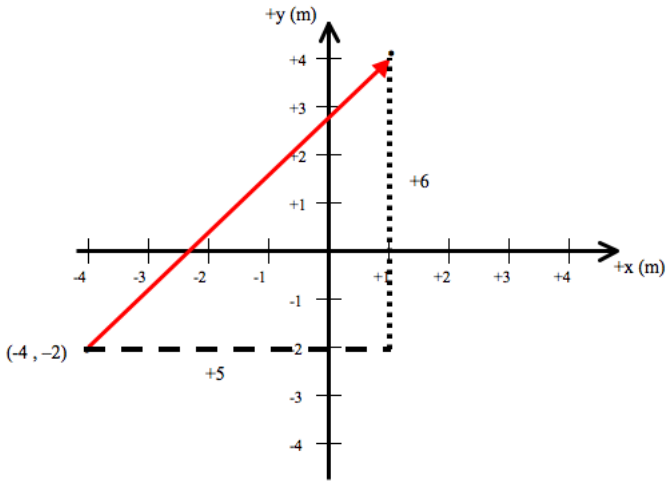


Figure 4.2: The x and y components of a vector.

$\vec{A}$. 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 4.1.)

Table 4.1: 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 $\vec{\mathrm{B}}$ is at (300,100) pixels, and $\vec{\mathrm{B}} = (150, 200)$ pixels. Where is the head of $\vec{\mathrm{B}}$?

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 4.2. 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 4.2?

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

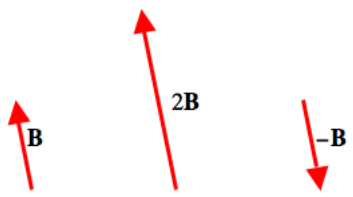


Figure 4.3: Multiplying a vector by a scalar.

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

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

$$\hat{r} = \left(\frac{r_x}{|\vec{r}|}, \frac{r_z}{|\vec{r}|}, \frac{r_z}{|\vec{r}|} \right)$$

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 $\hat{x}$, $\hat{y}$, and $\hat{z}$ which point along the x, y and z axes, respectively. They are written as

$$\hat{x} = (1, 0, 0)$$

$$\hat{y} = (0, 1, 0)$$

$$\hat{z} = (0, 0, 1)$$

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 4.4, an object at the coordinates (2,3) has a position vector $\vec{r} = (2, 3)$ m (if the units are assumed to be meters).

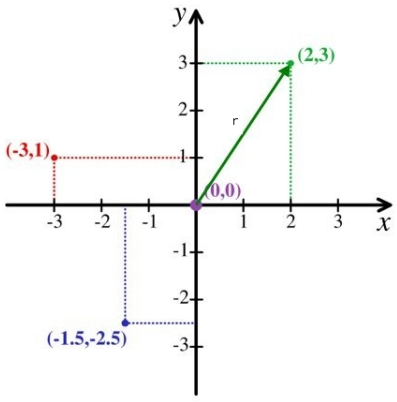


Figure 4.4: A position of an object is a vector from the origin to the coordinates of the object.

Vectors in VPython

One of the reasons that we are using VPython 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 in GlowScript and give it an appropriately descriptive name like **vectors**. (See your previous programs as an example.)

   ### Multiplying by a scalar; Scaling an arrow's axis

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

2. Create a green tennis ball at the position (2,0,0) m and name it `tennisball` .

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

3. Run your program and view the position of the tennis ball relative to the origin.

4. Modify the position of the tennis ball by multiplying its position by a factor 2 as shown below, 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?

For each of these questions, test your answers by editing and running your program.

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

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

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.

5. Place the tennis ball at the position $(2, 0, 0)$ m.

6. Create a sphere at the position (-1,4,0) m and name the sphere `baseball` .

7. Now, let's create an arrow that represents the position of the baseball. It should point from the origin to `baseball.pos` , so the arrow's `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)
```

8. Now, let's 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, this is calculated as `baseball.pos-tennisball.pos` . So, write:

```
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. In other words, the `axis` of the arrow is just the components of the arrow.

Homework

1. In the game *Missile Command*, 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.

   Create a new GlowScript program. 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. Only use the names and attributes of the objects.

   (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.

   (c) Print the position of the space probe. Again, do not use any numbers in your print statement.

5 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 5.1. 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

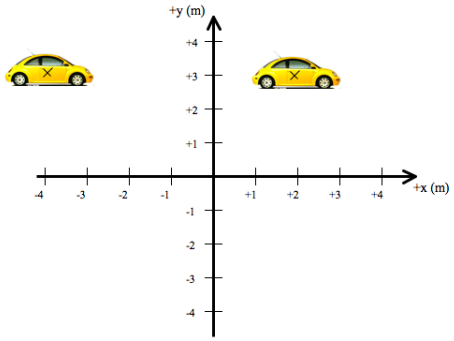


Figure 5.1: A car moves to the right.

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 5.2.

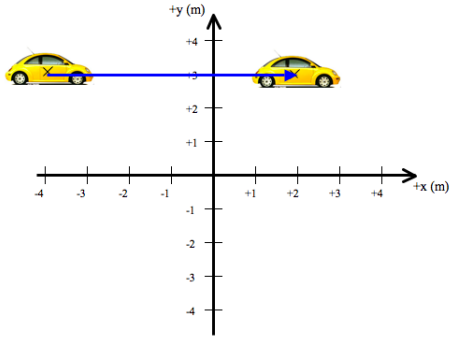
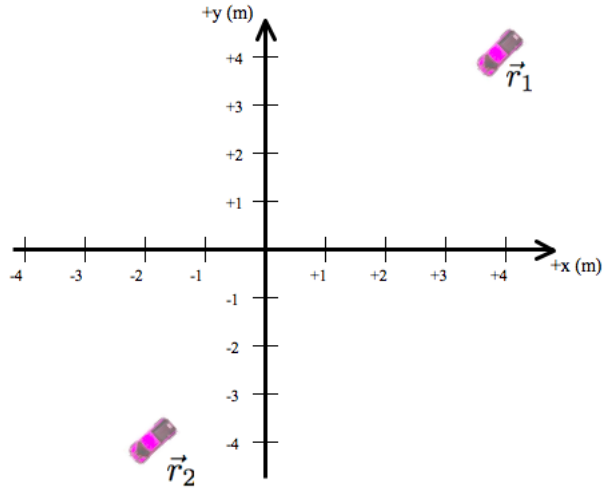


Figure 5.2: The car's displacement vector.

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 5.2.

Example

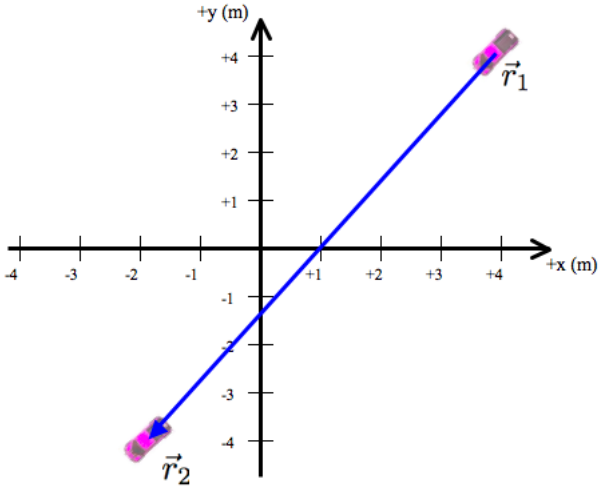
Question:
The top view of a pickup moving from $\vec{r}_1$ to $\vec{r}_2$ is shown below. Sketch and calculate the truck's displacement.



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 5.3.

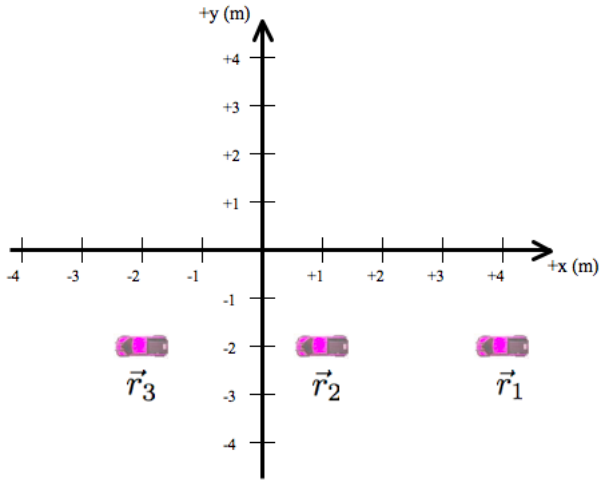


Figure 5.3: A pickup is displaced (-3,0) m to the left.

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 5.3. 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 5.3 and sketch the location of the pickup at $\vec{r}_4$.

Uniform motion

Each step of the pickup shown in Figure 5.3 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 5.3 for the pickup.

For example, consider a simulation of a cart moving to the right on a track shown in Figure 5.4. 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.

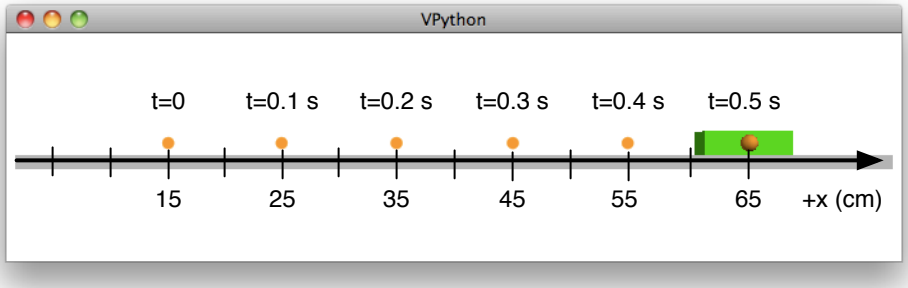


Figure 5.4: A cart on a track moves to the right with uniform motion.

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

Table 5.1: x-positions of the cart in Figure 5.4.

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 5.5. 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.

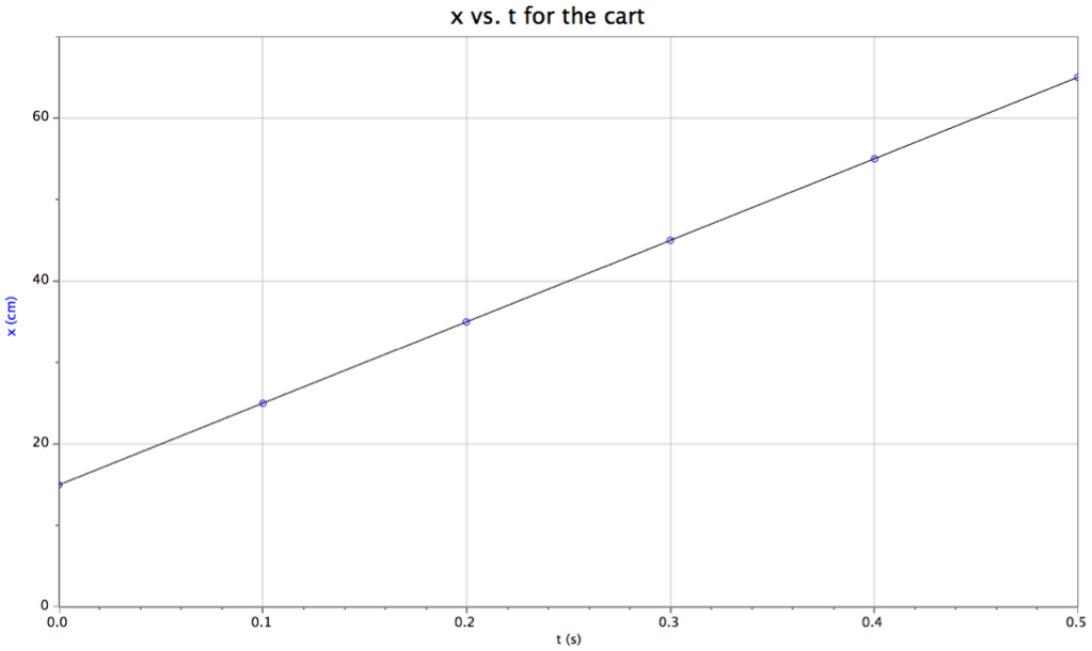
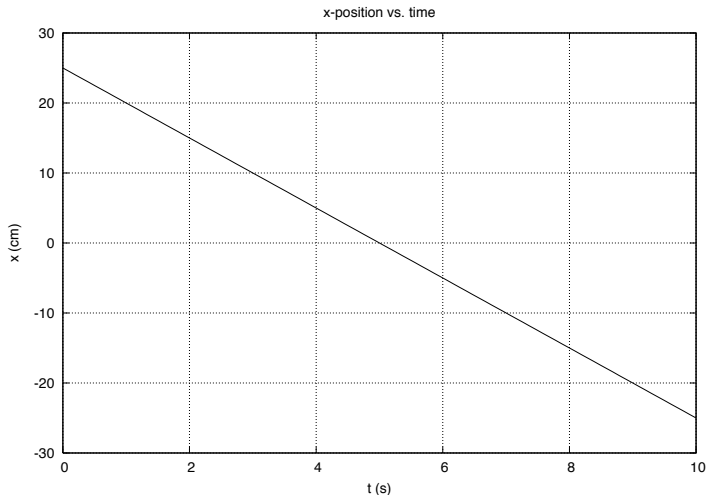


Figure 5.5: A graph of x-position as a function of time for the cart.

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 5.4 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 5.2. The object is drawn at each time step in Figure 5.6.

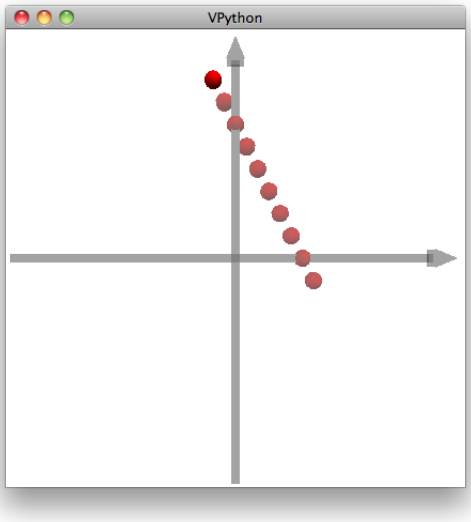


Figure 5.6: Updated positions of the object from Table 5.2.

Table 5.2: 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 5.7.

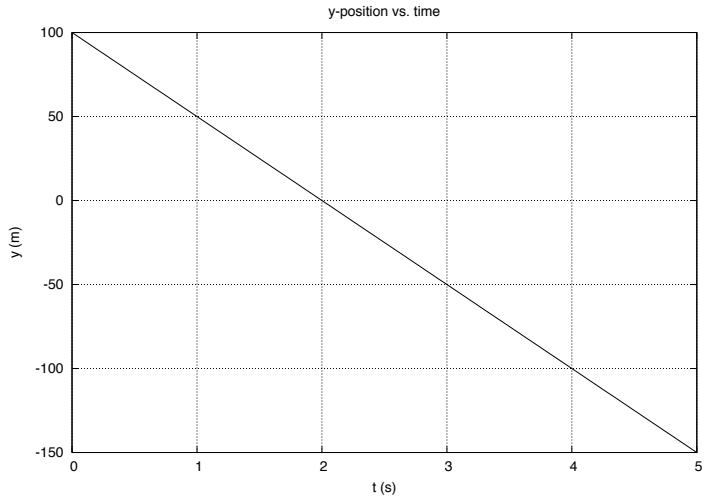


Figure 5.7: y(t) graph for an object

   (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.

6 LAB: Video Analysis of Uniform Motion

Apparatus

Tracker software (free; download from http://www.opensourcephysics.org/items/detail.cfm?ID=7365)
video: `uniform-motion-ball-slow.mov` from http://physics.highpoint.edu/∼atitus/videos/
video: `uniform-motion-ball-fast.mov` from http://physics.highpoint.edu/∼atitus/videos/

Goal

In this experiment, you will measure and graph the x-position of a rolling steel ball as a function of time. In addition, you will learn how to use video analysis software *Tracker* to measure position as a function of time for an object and find the best-fit curve to a graph.

Introduction

A video is basically a set of images recorded at a rate of 30 frames per second, or in other words a time interval of 1/30 s between frames. (However, high-speed video has a higher frame rate.) To measure an object's position in a video, you need to:

1. define a coordinate system including an origin and x,y axes.

2. define a scale; in other words define a standard length, perhaps 1 m, in the video. For this, it helps to have an object of known length, such as a meterstick, in the video. This is called a *calibration*.

It's very important that the calibration instrument, like a meterstick, is in the same plane as the object's motion. If the meterstick is closer to the camera or further from the camera than the object you are studying, then your measurement of position will be inaccurate.

Consider the object in Fig. 6.1. To measure its x-position, draw a perpendicular line from the object to the x-axis. To measure its y-position, draw a perpendicular line from the object to the y-axis.

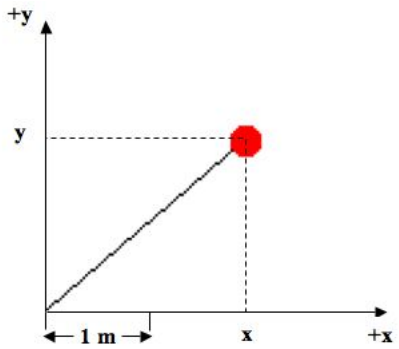


Figure 6.1: Position of an object depends on the origin and scale of the coordinate system.

Using the 1 m scale shown in the image, estimate the (x,y) coordinate of the object and use this to calculate the approximate distance of the object from the origin in Fig. 6.1

Video analysis software makes it easy to measure position coordinates (both x and y) and time for an object. After defining the scale and the coordinate system, you click on the object. The software shows a dot where you clicked and advances the video to the next frame. The software measures the position of where you clicked in units of pixels and then uses the calibration and your definition of the coordinate system to convert this position in pixels to a position in meters (or whatever units are used in the calibration).

The software also measures time because it knows that the video is recorded at 30 frames per second (or perhaps higher for high-speed video). Thus, whenever you advance the video, time advances 1/30 s. With time and position measured by the software, you can calculate velocity by numerically calculating the derivative of the position with respect to time. Other variables can also be calculated and graphed. All of these calculations can be done by the software.

Procedure

(Note: icons and screen captures in this handout refer to an older version of Tracker. You will notice some differences in the newest version of Tracker.)

1. Download the file *constant-velocity-slow.mov* from the given web site by right-clicking on the link and choosing **Save As...** to save it to your desktop.

2. Open the *Tracker* software on your computer.

3. Use the menu **Video→Import...** to import your video, as shown in Figure 6.2.

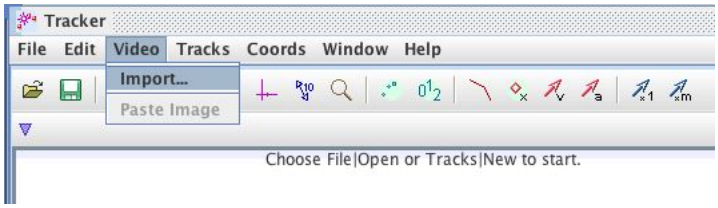


Figure 6.2: Video→Import menu

4. To zoom in or out on the video, click on the toolbar's magnifying glass icon that is shown in Figure 6.3. When it appears with a [+], then clicking once on the video will zoom in (thus making it larger). Clicking the magnifying glass again will make it [−]; then clicking on the video will zoom out (thus making it smaller). Zoom in and out on the video to see how it works.

Figure 6.3: The icon used to expand the video.

5. At this point, it's nice to lay out the video and graphs so that you can clearly see everything. The middle border between panes, seen in Fig. 6.4, can be dragged left and right to make the video pane smaller and graphs larger. The same is true of any other bar that separates panes in the window.

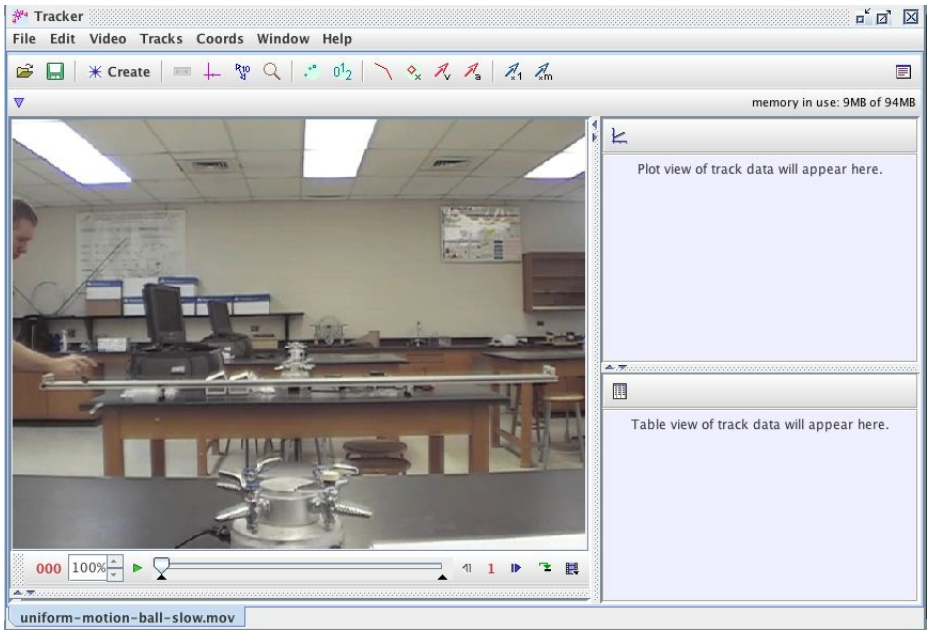


Figure 6.4: Drag the vertical or horizontal bars the make panes larger or smaller.

6. Note the video controls at the bottom of the video pane. Go ahead and play the video, step it forward, backward, etc. in order to learn how the video controls work. Note the counter that merely shows the frame number for any frame. Also, click on each of the icons in the video control bar to see what they are used for. Finally, use the left and right arrow keys on your keyboard, and note that they can be used to control the video as well.

7. Rewind to the first frame of the video. This is the instant that you will begin making measurements of the position of the moving object.

8. Since the ball moves fairly slowly on the track, we can skip frames between marking the ball and thus take fewer data points. Click on the **Step Size** button, as shown in Figure 6.5 and change it to **5**.

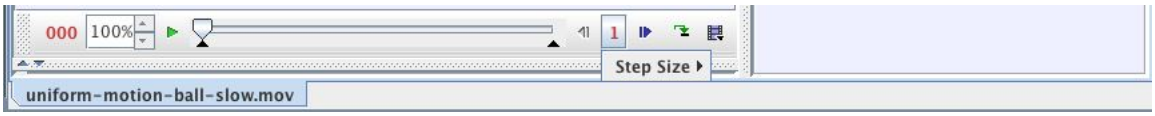


Figure 6.5: Change the step size in order to skip frames.

9. You now need to define the origin of the coordinate system. In the toolbar, click the **Axes** icon shown in Fig. 6.6 to show the axes of the coordinate system. (By now, you have probably noticed that you can hover the mouse over each icon to see what they do).

10. Click and drag on the video to place the origin of the coordinate system at the location where you would like to define (0,0), as shown in Figure 6.7. You can place the origin at any point you choose, but in this case, it makes sense to put the origin at the location of the ball in the first frame being analyzed.

Figure 6.6: Icon used to set the coordinate system axes.

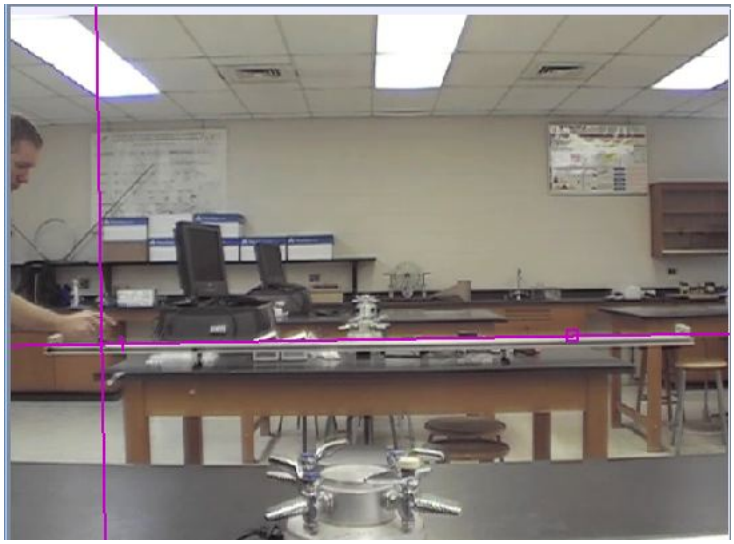


Figure 6.7: Click and drag to set the origin of the coordinate system

11. If you click the x-axis and drag, you can rotate the coordinate system. In this case, the video camera was not level; therefore, rotate the x-axis until it is parallel to the track.

12. Click the **Axes** tool again to hide the axes from the video pane. You can click this icon at any time to show or hide the axes.

13. Now, you must calibrate distances measured in the video. In the toolbar, click on the **Tape Measure** icon shown in Figure 6.8 to set the scale for the video.

Figure 6.8: Icon used to set the scale.

14. A blue double-sided arrow will appear. Move the left end of the arrow to the left end of the track, and move the right end of the arrow to the right end of the track. Double-click the number that is in the center of the arrow, and enter the length of the track, 2.2. (Our units are meters, but Tracker does not use units. You must remember that the number 2.2 is given in meters.) The scale will appear as shown in Figure 6.9.

15. Click the tape measure icon again to hide the blue scale from the video.

16. You are ready to add markers to the video to mark the position of the ball. Let's not show the coordinate system and scale. It's too distracting. So, make sure you've clicked the **Axes** and **Tape Measure** icons in the toolbar to hide them.

Figure 6.9: Enter the length of the track.

17. To add markers, click on the Create button and select **Point Mass** as shown in Figure 6.10. Then, **mass A** will be created, and a new x vs. t graph will appear in a different pane. You will now be able to mark the position of the ball which will be referred to as **mass A**.

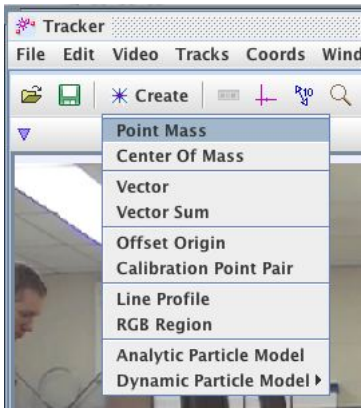


Figure 6.10: The Track Control toolbar.

18. We will want to set the frame step size to 5 so that it will skip 4 frames every time we step forward in the video. Double check that the step size is 5 in the video control toolbar.

19. **To mark the center of the ball, hold the SHIFT key down and click once on the center of the ball.** You should notice that a marker appears at the position of the ball where you clicked and that the video advances one step.

20. Again, shift-click on the ball to mark its position. You should now see two marks.

21. Continue marking the position of the ball until it reaches the right end of the track. Note that only a few of the marks are shown in the video pane. To display all of the marks or a few of the marks or none of the marks, use the **Set Trail Length** icon shown in Figure 6.11. Clicking this icon continuously

will cycle through no trail, short trail, and full trail which will show you no marks, a few marks, or all marks.

Figure 6.11: The Set Trail icon is used to vary the number of marks shown.

After marking the ball as it moves from the left end to the right end of the track, your video should look like the picture shown in Fig. 6.12 if you have set the trail length to show the full trail.

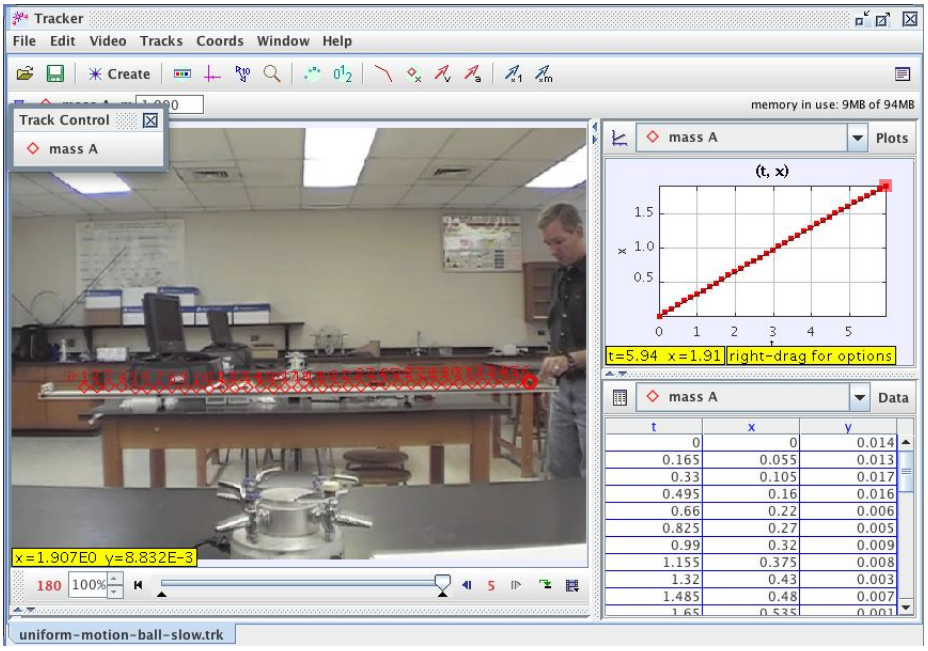


Figure 6.12: Marks showing the ball's position.

22. Tracker uses the frame and frame rate to calculate t, and it uses the scale and coordinates of the marks to calculate x and y coordinates for the ball. It uses numerical differentiation to calculate x-velocity and y-velocity.

Analysis

x vs. t graph

1. We will now analyze the x vs. t graph. You can click and drag the border of the video pane to make it smaller so that you can focus on the graph.

2. Play the video. (You can hide the marks if you wish by clicking the **Show or hide positions** icon, and you can show the path by clicking the **Show or hide paths** icon. Both of these icons are in the toolbar.) Note how the graph and video are synced. Each video frame data point is shown in the graph using a filled rectangle.

   Also, when you click on a data point on the graph, the video moves to the corresponding frame.

3. Observe the x vs. t graph.

Describe in words the type of function that describes this graph of x vs. t? (i.e. linear, quadratic, square root, sinusoidal, etc.)

4. Right-click (or ctrl-click) on the graph and select **Analyze...** . In the resulting window, check the checkbox for **Fit** , and additional input boxes will appear, as shown in Figure 6.13. The Fit Name should be "Line" and the equation will be $x = a * t + b$ where a is the slope and b is the vertical intercept. Check the checkbox for **Autofit** and the best-fit line will appear in the graph.

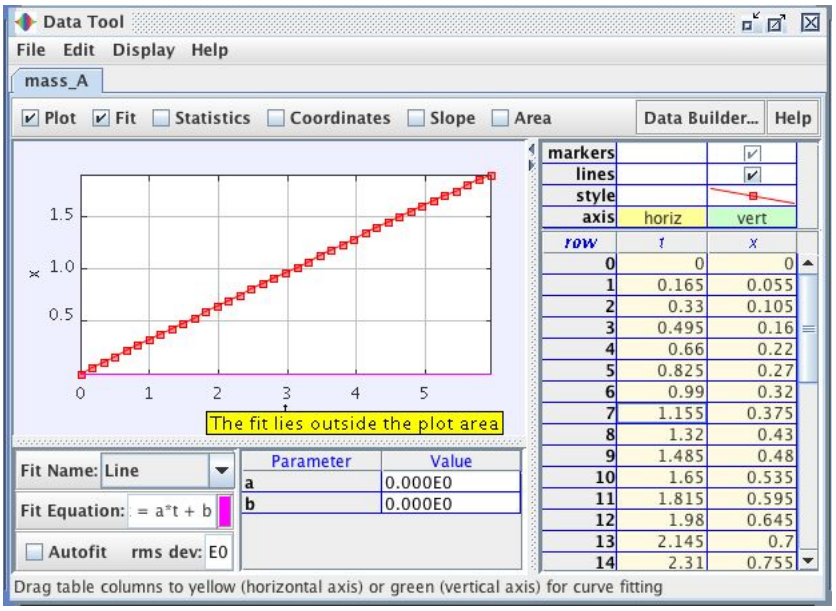


Figure 6.13: The Data Tool for finding the best-fit curve for the data.

Record the function and the values of the constants for your curve fit. Write the function for x(t), with the appropriate constants (also called fit parameters).

In general, what does the slope of the x vs t graph tell you? (Consider its units. Your answer should be a sentence, not a number.)

From the curve fit parameters, determine the x-velocity of the ball.

v_x vs. t graph

5. Now, we will look at the x-velocity vs. time graph.

What do you expect the x-velocity vs. time graph to look like? Sketch your prediction below.

6. Close the Data Tool window and return to the main window. Click once on the vertical axis label and change it from x to vx, as shown in Figure 6.14.

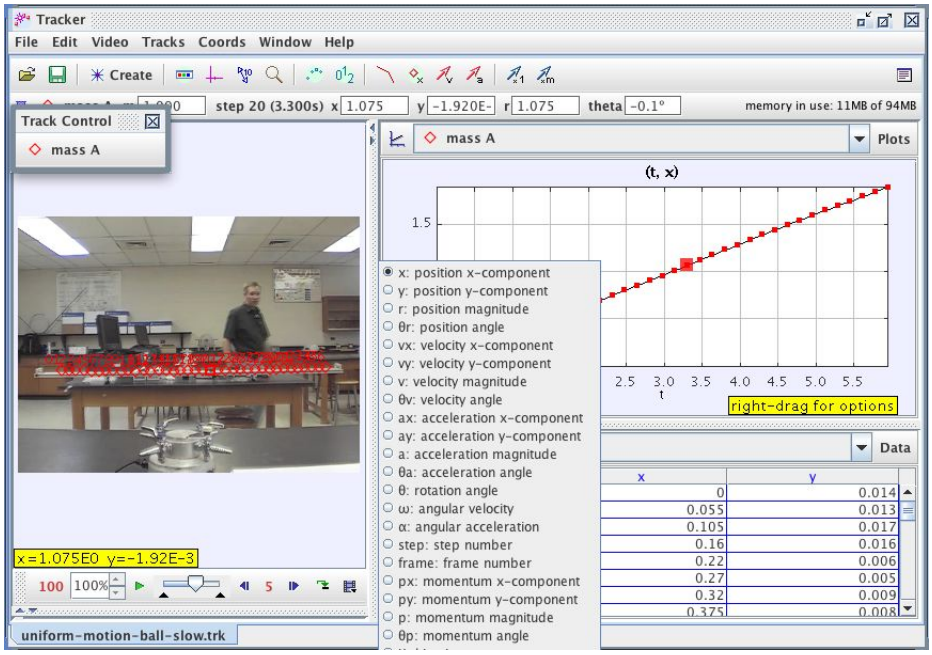


Figure 6.14: Changing the variables plotted on the graph.

Note that the data appears to be all over the place. That's because by default, the graph is "zoomed in" on the data. If you examine the numbers on the vertical axis, you'll notice that the data probably lies between 0.3 m/s and 0.35 m/s (though your data might also vary from mine). That's a very small variation in the velocity during the 5-second time interval that the ball is moving. And the variation is likely due to measurement error such as not clicking exactly on the center of the ball for every mark.

7. Hover over the lowest part of the vertical axis (near the graph's origin, on the vertical axis). Click once and change the minimum on the vertical scale to 0. Do the same thing at the top of the scale

and change the maximum to 0.5. The data now appears to be along a horizontal line, though there is some scattering in the data due to uncertainty in the measurements, as shown in Figure 6.15.

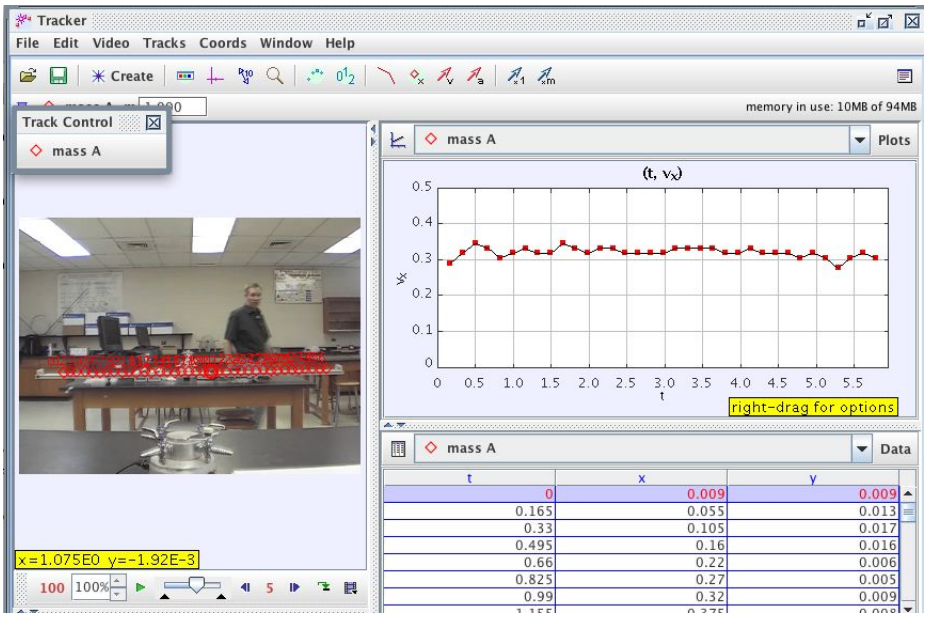


Figure 6.15: The x-velocity vs. time graph.

8. Right-click (or ctrl-click) on the graph and select **Analyze...** in order to analyze the v_x vs. t graph. You may notice that the data for both x and v_x are displayed on the same graph. If this occurs, uncheck the checkboxes for x in the upper right corner of the window.

9. Once again, change the minimum and maximum values on the scale so that the graph is not zoomed in on the data. Do an auto-fit.

Record the fit constants and equation for the curve fit. From the curve fit, determine the initial x-velocity.

You will notice a slight downward slope of this graph. What does this tell you? Answer in a complete sentence.

10. Since the x-velocity is nearly constant, we would like to have an average of the x-velocity measured at each instant. Thus, check the checkbox for **Statistics** .

Record the mean x-velocity and the standard deviation.

Is the slope of the x vs. t graph within $\overline{v_x} \pm \sigma$?

11. Be sure to save the Tracker file in the same folder as your video. Keep these files for your records. In fact, it is best to save as you go. ***Remember, save early and often!***

Lab Report

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

1. If you were to only see the marks in the video and not see any graphs, how would you know that it is uniform motion (as opposed to non-uniform motion)?
2. What is the x-velocity of the ball as measured by the slope of your graph of x vs. t?
3. What is the x-velocity of the ball as measured by the average of the values of v_x on the graph of v_x vs. t?
4. When you found the mean value of v_x, you also recorded the standard deviation. What was the standard deviation and did your results reported in parts (2) and (3) agree within this standard deviation?

B Do all parts for **C**, analyze the motion of the ball in the video *constant-velocity-fast.mov* and answer the following questions.

1. What is the x-velocity of the ball as determined from the x vs. t graph?
2. What is the average and standard deviation of the x-velocity of the ball as determined from the v_x vs. t graph?

A Do all parts for **B** and answer the following questions. Note that you can make a sketch using a ruler. You don't have to use graph paper. However, you may use graph paper or a computer to do your sketches.

1. Car A travels with a constant x-velocity of 30 mph for two hours, and Car B travels with a constant x-velocity of 60 mph for two hours. Each of them start at the origin at $t = 0$. On the same set of axes, sketch a graph of x vs. t for each car. (There should be two lines on your graph. Be sure to label the axes with correct units.)
2. Suppose that the ball in the video started at the right end of the track and traveled with a constant velocity to the left. If the origin is set at the left end of the track with the $+x$ axis pointing to the right (just as before), sketch a graph of x vs. t for the ball.
3. Suppose that the ball in the video started at the right end of the track and traveled with a constant velocity to the left. If the origin is set at the right end of the track with the $+x$ axis pointing to the right, sketch a graph of x vs. t for the ball.

7 PROGRAM – Uniform Motion

Apparatus

Computer
GlowScript – www.glowscript.org

Goal

The purpose of this activity is to learn how to use GlowScript (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. Create a new program in GlowScript and save this file with a new name like `ball-uniform-motion`.
2. Add the line below to create a track that is at the origin and has a length of 3 m, a height of 0.05 m, and a width of 0.1 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=vector(3,0.05,0.1), color=color.
    white)
```

3. 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 and name it `ball` .

Figure 7.1: A ball on a track.

4. 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 7.1.

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

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

   Next we will see how to write this in VPython.

5. 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.

6. 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)
```

7. 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.

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

```
dt=0.01
```

9. 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 and name them.
   (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.

10. For now, let's run the animation for 10.0 s. On a new line, begin the `while` statement as shown below. 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.

11. Below 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.

12. 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, check that your `while` loop looks like:

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

13. Run the program. View the clock readings printed below the 3D scene.

14. You can make it run indefinitely (i.e. without stopping) by saying "while true" so you can change the `while` statement to read:

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

The `rate(100)` statement tells the computer to try to run the loop 100 times in one second.

Check that your program (loop) now looks like:

```
while 1:
        rate(100)
        t=t+dt
        print(t)
```

15. Run the program. Now, it will print clock readings continually until you click the **Edit this program** link and return to the editor.

> Stop and reflect on what is going on in this `while` loop. Your understanding of this code is essential for writing 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:

$$\begin{aligned} \text{new position coordinates} &= \text{current position coordinates} + \text{velocity} \times \text{time step} \\ \vec{r}_f &= \vec{r}_i + \vec{v}\Delta t \end{aligned}$$

This is called the *position update equation*. It says, "take the current position of the object, add its displacement, and the result is 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` .

16. 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, check that your `while` loop looks like:

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

17. 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)
```

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

19. Printing the values of the time and the ball's position may slow down the computer. Comment out your print statement by typing the # sign in front of the `print` statement (as in `#print)` . Run your program again.

    Sometimes you need to print data in order to check the computer's calculations. However, it can also be distracting and unnecessary. In general, print when you need to check the computer's calculation and debug your program. Otherwise, don't print.

20. 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.)

    If you want time to advance in the simulation at the same rate as a real clock (meaning, as nearly as possible, the simulation time is equal to real time), then set the values of `dt` and `rate()` so the product is equal to one. For example, if `dt=0.01` , then choose `rate(100)` because $0.01 * 100 = 1$. Or if `dt=0.02` , then choose `rate(50)` .

Analysis

C Do all of the following.

1. Start a new program in GlowScript and save this file with a new name like `ball-uniform-motion-C`.
2. Copy the code from your previous program and paste it into this new program.
3. Simulate the motion of a ball that starts on the right end of the track and travels to the left with a speed of 0.5 m/s for 5 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. Start a new program in GlowScript and save this file with a new name like `ball-uniform-motion-B`.
2. Create two balls on a track: 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.
3. 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.
4. Set the `while` loop to run while $t < 5$ s.
5. Print the clock reading `t` and the position of each ball up to $t = 5$ s.
6. At what clock reading t do they pass through each other?

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

1. Start a new program in GlowScript and save this file with a new name like `ball-uniform-motion-A`.
2. Create a similar track as before, but with the width as 3 m. When the scene is rotated, the track appears as a table top.
3. 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.
4. Set the x-velocities of the balls to: (A) 0.25 m/s, (B) 0.5 m/s, and (C) 0.75 m/s.
5. At what time does Ball C reach the end of the table? (Use a print statement to determine this.)
6. What are the positions of all three balls when Ball C reaches the end of the table? (Use a print statement to determine this.)

8 PROGRAM – Lists, Loops, and Ifs

Apparatus

Computer
GlowScript – www.glowscript.org

Goal

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

Introduction

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. The instructions in this chapter 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. Create a new program and save it with a name like `move-objects.py`.

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

2. Type the `for` loop shown below.

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

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

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

   In order to see all of the digits, you may have to scroll in the text box or click and drag on the bottom right corner of the box to expand it.

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 for the first iteration of the loop, the value of `i` is 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.

4. 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:

5. 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.

6. 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.

7. Run your program to verify that you have four balls at the given locations. The screen should look like Figure 8.1 but perhaps with a black background and different color balls.

8. 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.

9. 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.

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.

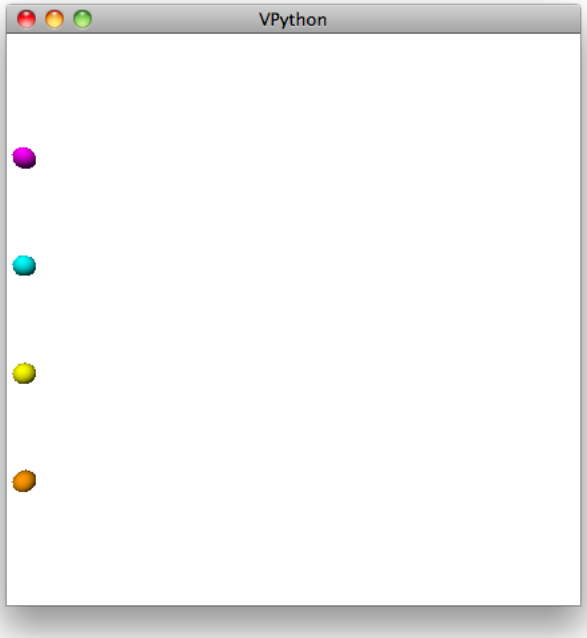


Figure 8.1: Four balls

- Create a `while` loop.
- Update the object's position and update the clock reading.

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

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

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

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

12. 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.

13. 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.

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

15. 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, perhaps on line 2 or 3, add the following lines:

```
scene.width=500
scene.height=500
scene.range=5
scene.autoscale=False
```

The height and width attributes set the size (in pixels) of the scene. 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.

16. 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.

17. 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
```

18. 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 a new program and give it a different name.
2. 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. Create a new program and give it a different name.
2. Copy your program in part (B) and paste it into your new program.
3. 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.
4. 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.

5. 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 where you check that the ball is at the edge of the screen, you move the ball back to its previous position.

9 PROGRAM – Keyboard Interactions

Apparatus

Computer
GlowScript – www.glowscript.org

Goal

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

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.

```
GlowScript 2.0 VPython

scene.width=500
scene.height=500
scene.range=5
scene.autoscale=False

ball1=sphere(pos=vector(-5,3,0), radius=0.2, color=color.magenta)
ball2=sphere(pos=vector(-5,1,0), radius=0.2, color=color.cyan)
ball3=sphere(pos=vector(-5,-1,0), radius=0.2, color=color.yellow)
ball4=sphere(pos=vector(-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 at the bottom left corner 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`.

5. Inside the `while` loop, after the `for` loop updates the positions of the balls, add a line to update the position of the shooter as shown below.

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

6. Run your program. You should see the shooter move to the right, and it will continue moving off the screen. If this does not occur, then check for errors of logic in your program.

   Now we want to use the keyboard to control the velocity of the box. We will 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.

   Moving the box occurs inside the while loop. However, we need the GlowScript environment to continually monitor whether a key has been pressed on the keyboard. Then, when the key is pressed, our code will take over by checking which key it is and setting the velocity of the shooter.

7. On line 3 of your program, immediately after the "GlowScript 2.0 VPython" statement, write the following function.

```
def keyboard(event):
    if event.type=='keydown':
        k = event.which
        print(k)

scene.bind('keydown', keyboard)
```

   Let me explain what this code is doing. GlowScript continually monitors for keyboard and mouse events. The `scene.bind('keydown', keyboard)` function tells VPython that it should call the *keyboard* function whenever a *keydown* event is detected. The *keyboard* function is a custom-defined function. We could have named it anything. (I picked the name *keyboard* just because it made sense to me.) In this function, I first check to see what `type` of event occurred. If it is *keydown* then I get the key and print it.

8. Run your program. Press various keys and note the number that is printed.

What are the numbers that correspond to these keys?
j:
k:
l:
a:
s:
d:
spacebar:
left arrow:
right arrow:
up arrow:
down arrow:

In the previous code, we printed the key because we wanted to see the unique number for each key. But now we want to use the keyboard to control the velocity of the box. Let's set the velocity of the box to be to the right, if the right arrow is pressed, and to the left, if the left arrow is pressed.

9. Comment out the `print` statement since you already figured out the numbers that correspond to the arrow keys.

10. Change the `keyboard` function to be the following:

```
def keyboard(event):
    if event.type=='keydown':
        k = event.which
#         print(k)
        if k == 39:
            shooter.v=2*vector(1,0,0)
        elif k == 37:
            shooter.v=2*vector(-1,0,0)
        else:
            shooter.v=vector(0,0,0)
```

The function *keyboard* checks to see which key is pressed and sets the velocity accordingly.

11. Run your program. Press various keys to see if the program works as expected.

12. Change your program so that pressing `a` causes a fast leftward velocity and pressing `s` causes a fast rightward velocity. In summary, the left and right arrows create slow velocities to the left and right; the `a` and `s` keys create fast velocities to the left and right.

13. Run your program and verify that it works as expected.

14. You probably noticed that it's annoying when the box moves past the edge of the edge of the screen. Use an `if` statement in the `while` loop to check if the box passes the edge of the screen. If it does, then reverse its velocity. The best way to reverse the velocity is to multiply it by -1 as shown below.

```
            shooter.v=-shooter.v
```

15. Run your program. The box should reverse, with the same speed, whenever it reaches the edge of the screen. Furthermore, you should be able to set the velocity (fast or slow) of the box using right arrow, left arrow, a, or s, and stop the box with any other key.

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=[]
```

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

```
        elif k==32:
            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 center 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`. Later, in the `while` loop, we can update the positions of all the bullets in this list.

17. Now we have to update the positions of the bullets (i.e. make them move). In your `while` statement before you update the clock, add a `for` loop that updates the positions of the bullets in the bulletsList.

```
    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=-thisball.v
        elif thisball.pos.x<-5:
            thisball.v=-thisball.v

    shooter.pos = shooter.pos + shooter.v*dt
    if(shooter.pos.x>5):
        shooter.v=-shooter.v
    elif(shooter.pos.x<-5):
        shooter.v=-shooter.v

    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. There are three important sections. One section updates the positions of the balls and reverses their velocities if they reach the edge of the screen. The next section updates the position of the shooter and reverses its velocity if it reaches the edge of the screen. The last section updates the positions of the bullets.

18. Run your program and verify that all aspects work as expected.

Analysis

C Do all of the following.

1. Create a new file and give it an appropriate name. Copy and paste previous code to do the following tasks.
2. Assign variables to the speed of the shooter (both slow and fast) and the speed of the bullet. Call them: `sfast`, `sslow`, and `sbullet`. Write these at the top of your program since you will use them later in the program.
3. When setting the velocity of the shooter in your keyboard function, use the variable for the speed of the shooter. Here is an example:

```
        if k == 39:
            shooter.v=sslow*vector(1,0,0)
```

4. Set `sslow` to be very low, like 0.5. And set `sfast` to be very fast, perhaps 5. Also change `sbullet`. See how changing these values affects the game. By using variables, it makes it much easier to change their values for the purpose of gameplay. If you do not use variables, then you have many lines to change if you want to test higher or lower speeds.
5. Give instructions about your game by adding this code at line 2 or 3 of your program, just after the line that says, `GlowScript 2.0 VPython`. These statements add text to the `title` of the scene and will appear above the scene. HTML tags like `<b>` are needed to format the text.

```
scene.title.append('<h2>Instuctions</h2>')
scene.title.append('<br><br>')
scene.title.append('Use the following keys to control the shooter.')
scene.title.append('<br> -- Right arrow - slow, to the right')
scene.title.append('<br> -- Left arrow - slow, to the left')
scene.title.append('<br> -- s - fast, to the right')
scene.title.append('<br> -- a - fast, to the right')
scene.title.append('<br> -- spacebar - shoot a bullet')
scene.title.append('<br> -- any other key - stop')
```

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

1. Create a new file and give it an appropriate name. Copy and paste previous code to do the following tasks.
2. It looks strange for the bullets to come from the center of the box. Fire the bullets from the center of the top plane of the box instead of its center. To do this, you'll have to change the initial position of the bullet when it is created.
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.
4. Add additional keystrokes that will fire a bullet to the left, to the right, or downward. You may wish to use the arrow keys to fire bullets and other keys for changing the velocity of the shooter. Feel free to reassign keys to whatever makes sense. Change the instructions at the top to match the keys you choose.

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

1. Create a new file and give it an appropriate name. Copy and paste previous code to do the following tasks.
2. Add a counter variable 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. To do this, you must add the following line immediately after the defining the keyboard function with **def** keyboard(). It should look like the lines shown below.

```
def keyboard(event):
        global shots
```

This line makes the shots variable, which was defined outside the function, available within the function.

3. Suppose that the shooter only has 10 bullets. Write code so that if the shooter reaches a maximum of 10 bullets, hitting the spacebar will no longer file a bullet.
4. Create a keystroke that will replenish the shooter, meaning that after hitting this keystroke, you can fire 10 more bullets.

10 PROGRAM – Collision Detection

Apparatus

Computer
GlowScript – www.glowscript.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.

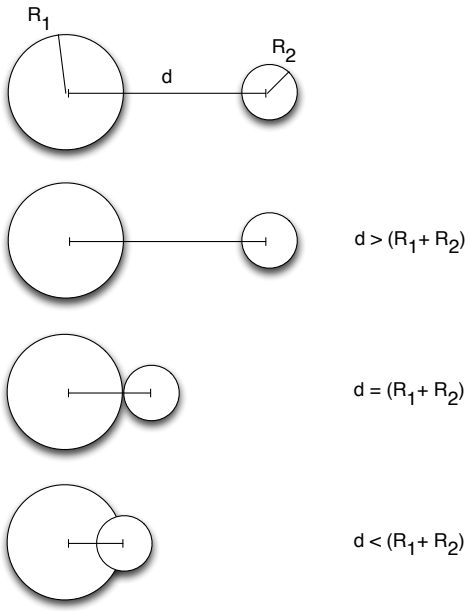


Figure 10.1: Condition for whether two spheres collide.

As shown in Figure 10.1:

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 10.2.

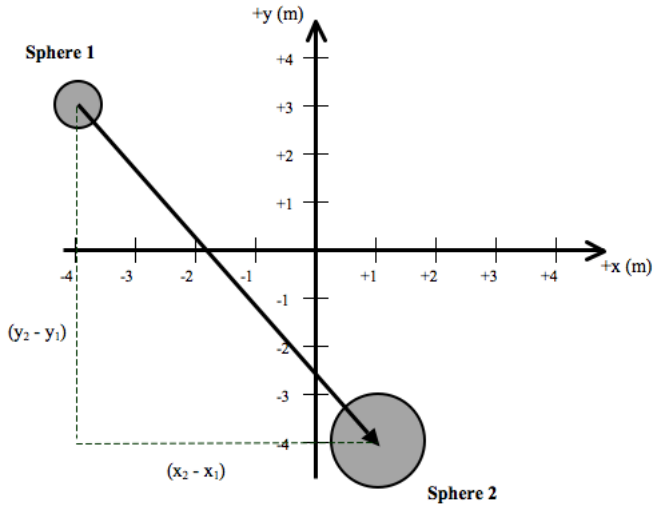


Figure 10.2: Distance between two spheres.

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

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

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

Exercises

Ball1 is at $(-3, 2, 0)$ m and has a radius of 0.05 m. Ball2 is at $(1, -5, 0)$ m and has a radius of 0.1 m. What is the distance between them?

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

Procedure

Starting program

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

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

```
GlowScript 2.0 VPython

def keyboard(event):
    if event.type=='keydown':
        k = event.which
#         print(k)
        if k == 39:
            shooter.v=2*vector(1,0,0)
        elif k == 37:
            shooter.v=2*vector(-1,0,0)
        elif k == 65:
            shooter.v=4*vector(-1,0,0)
        elif k == 83:
            shooter.v=4*vector(1,0,0)
        elif k==32:
            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)

scene.bind('keydown', keyboard)

scene.width=500
scene.height=500
scene.range=5
scene.autoscale=False

ball1=sphere(pos=vector(-5,3,0), radius=0.2, color=color.magenta)
ball2=sphere(pos=vector(-5,1,0), radius=0.2, color=color.cyan)
ball3=sphere(pos=vector(-5,-1,0), radius=0.2, color=color.yellow)
ball4=sphere(pos=vector(-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=vector(-4.5,-4.5,0), width=1, height=1, length=1, color=
    color.red)
shooter.v=2*vector(1,0,0)

t=0
dt=0.01

bulletsList=[ ]

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

    shooter.pos = shooter.pos + shooter.v*dt
    if(shooter.pos.x>5):
        shooter.v=-shooter.v
    elif(shooter.pos.x<-5):
        shooter.v=-shooter.v

    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 `GlowScript` statement, perhaps on line 2 or 3.

```
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 web page.

3. Inside of and 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 iteration (each time step) of the loop.

5. Change your program 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 `GlowScript 2.0 VPython` 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 the names of two spheres. The function 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: (You should not need to type this, just compare it to your program.)

```
    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. If you zoom outward, you will see the balls. (They are drawn on top of each other, so you might only see one of them.)

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 Create a new file. Copy and paste your program from this lesson. Add all of the following features.

1. If a bullet exits the scene (i.e. `bullet.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. Remember, you increment a variable like the example shown below.

```
hits=hits+1
```

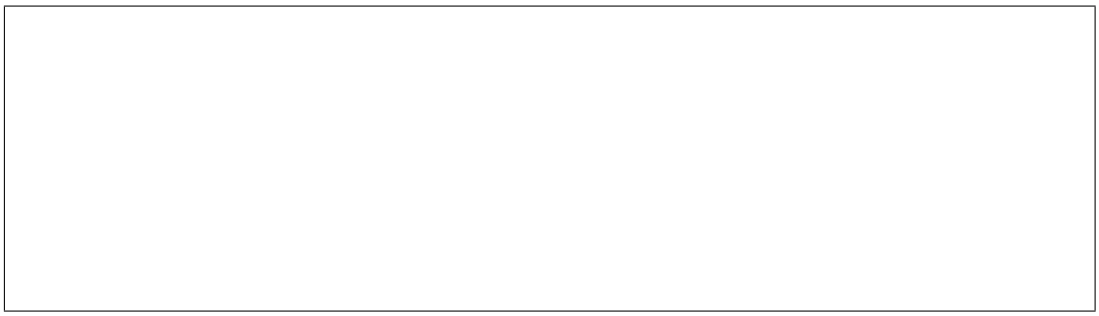
3. Print `hits` every time a missile hits a ball.

B Create a new file. Copy and paste your program from **C**. Add the following features.

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 bullet is fired.

A Create a new file. Copy and paste your program from **B**. Add the following features.

1. The score should not be simply based on whether a bullet 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 bullets shot, then you should get a higher score. Also, if you hit all four balls with only four bullets shot in only 1 s, then you should get a higher score than if it required 10 s. Design a scoring system based on bullets fired, hits, and time. This will require a mathematical function of your choosing that gives you the desired outcome. Write your scoring system below. Describe the goals of your scoring system, how points are awarded or subtracted, and write a mathematical function that either computes the score as a function of shots, hits, and time or write a function that updates the score whenever a shot, hit, or time changes.



2. Program your scoring system into the code. Use a variable `score` for the total score. Use a `print()` statement to update the score every time it changes.
3. After you are confident that it is working, write down your top 5 scores.

4. Ask at least three friends to play the game one or more times and write down the top score(s) by each friend.

5. What would you like to change about your scoring system or game based on the experience of your friends?

11 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 11.1.)

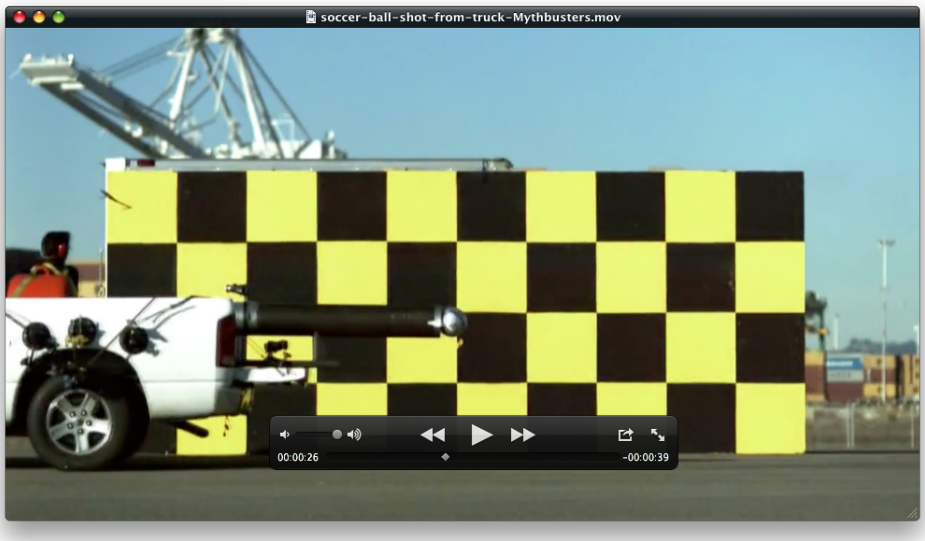


Figure 11.1: A soccer ball shot out the back of a pickup truck.

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:

$$\begin{array}{rcl} \vec{v'} & : & \text{velocity of an object measured by an observer in the } \textit{Other} \text{ frame} \\ \vec{v} & : & \text{velocity of an object measured by an observer in the } \textit{Home} \text{ frame} \\ \vec{\beta} & : & \text{velocity of the } \textit{Other} \text{ frame as measured in the } \textit{Home} \text{ frame} \end{array}$$

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

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 $v'_x = +20$ 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.

$$\begin{aligned} v_x &= v'_x + \beta_x \\ &= 20\text{m/s} - 25\text{m/s} \\ &= -5\text{m/s} \end{aligned}$$

The x-velocity is $v_x = -5$ m/s which means that the ball is moving to the left with a speed $|v| = 5\ m/s$ 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 bullet in the reference frame of the shooter?

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?

12 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 learned 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.

For an object colliding with a stationary, rigid barrier along an axis perpendicular to the surface of the barrier, the coefficient of restitution (COR) is defined as:

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

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 the collision is *elastic*. If COR< 1, then the collision is *inelastic*. If COR> 1, then the collision is *superelastic*. (This might happen for a spring-loaded bumper like in a pinball machine, but this would not be a rigid barrier.)

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.

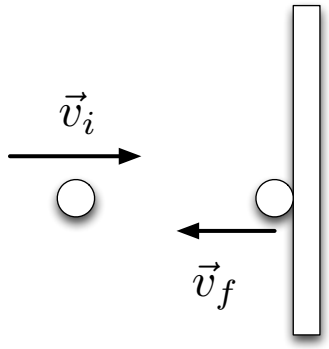


Figure 12.1: A ball collides with a rigid wall.

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

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

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 12.2, 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 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.

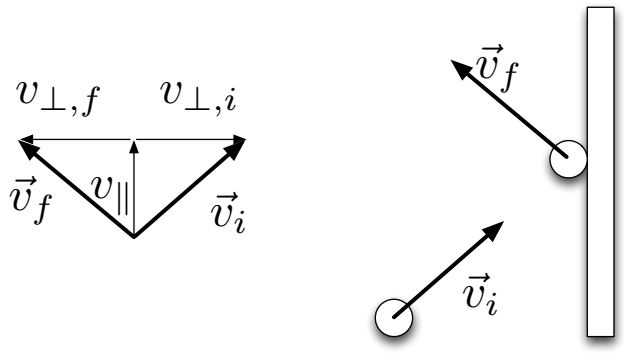


Figure 12.2: A 2-D collision with a frictionless rigid wall.

After colliding with the wall, the y-velocity 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.

For a frictionless surface, 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, depending on the COR. The perpendicular component of the velocity after the collision will be

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

If it is an elastic collision, then $v_{\perp,f}$ merely changes direction as shown in Figure 12.2.

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, friction always causes $v_\parallel$ to decrease in magnitude. If the barrier is moving, then it's possible to increase $v_\parallel$.

Consider a hockey puck bouncing off a rigid, stationary hockey stick as shown in Figure 12.3. Let's assume, for the sake of simplicity, that it is an elastic collision. In this case, the frictional force on the puck 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.

In other words, to get the direction of the friction correct, you have to calculate the velocity of the puck *relative to the stick* (i.e. in the stick's reference frame). This takes us back to the previous chapter on Galilean Relativity. If the stick's reference frame, the velocity of the puck parallel to the stick before the collision is

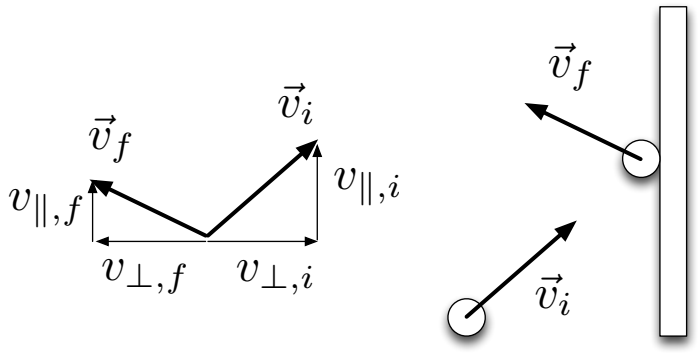


Figure 12.3: A 2-D collision with a frictionless rigid wall.

$$v'_{\parallel,i} = v_{\parallel,i} - \beta$$

where I am assuming that β is along the same axis as $v_{\parallel,i}$. The frictional force by the surface will be opposite $v'_{\parallel,i}$. This may or may not be opposite to $v_{\parallel,i}$.

Question:

A puck is traveling with a velocity $(1, 1, 0)$ m/s when it collides with a hockey stick. For each of these cases, (1) describe and sketch the direction of the frictional force on the puck by the hockey stick and the (2) initial and final velocity of the puck. Assume that the puck collides elastically in the perpendicular direction.

1. A rigidly held hockey stick is traveling in the $+y$ direction at 0.6 m/s when the puck collides with the stick.

2. A rigidly held hockey stick is traveling in the $+y$ direction at 1.3 m/s when the puck collides with the stick.

3. A rigidly held hockey stick is traveling in the $+y$ direction at 1.0 m/s when the puck collides with the stick.

Answer:

The direction of the frictional force depends on the velocity of the puck relative to the moving stick. For each case, compute the velocity of the puck relative to the stick.

1. The parallel direction is the +y direction. The puck's parallel component of its velocity relative to the stick is:

$$\begin{aligned} v'_{\parallel,i} &= v_{\parallel,i} - \beta \\ &= 1 \text{ m/s} - 0.6\text{m/s} \\ &= 0.4\text{m/s} \end{aligned}$$

The puck is traveling upward relative to the stick. So there is a friction force in the downward direction that decreases $v_{\parallel,i}$. As a result, the puck rebounds at a steeper angle, as shown in Figure 12.4.

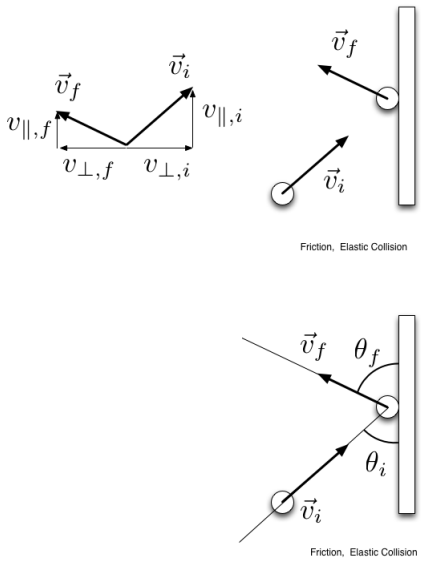


Figure 12.4: Friction decreases $v_{\parallel,i}$.

2. The puck's parallel component of its velocity relative to the stick is:

$$\begin{aligned} v'_{\parallel,i} &= v_{\parallel,i} - \beta \\ &= 1 \text{ m/s} - 1.3\text{m/s} \\ &= -0.3\text{m/s} \end{aligned}$$

The puck is traveling downward relative to the stick. So there is a friction force in the upward direction that increases $v_{\parallel,i}$. As a result, the puck rebounds at a flatter angle, as shown in Figure 12.5.

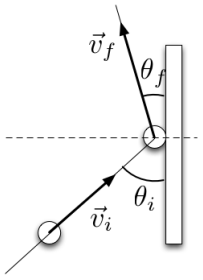


Figure 12.5: Friction increases $v_{\parallel,i}$.

3. The puck's parallel component of its velocity relative to the stick is:

$$\begin{aligned} v'_{\parallel,i} &= v_{\parallel,i} - \beta \\ &= 1 \text{ m/s} - 1.0\text{m/s} \\ &= 0 \text{ m/s} \end{aligned}$$

Relative to the stick, the puck is not moving in the parallel direction. In other words, in the stick's frame, the puck is incident perpendicular to the surface as shown in Figure 12.6. As a result, there is no frictional

force on the puck, and the puck rebounds with the same perpendicular component (elastic collision) and same parallel component of velocity. In other words, it rebounds at the same angle and merely "reflects" from the surface as shown in Figure 12.7.

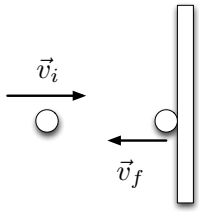


Figure 12.6: There is no friction by the stick on the puck in the stick's reference frame.

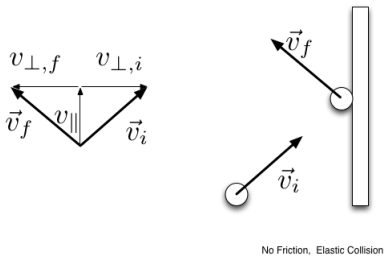


Figure 12.7: The puck reflects at the same angle.

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 12.8.

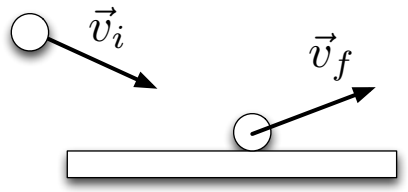


Figure 12.8: A 2-D collision with a frictionless rigid wall.

   (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 12.9. 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?

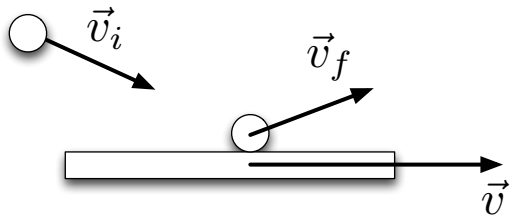


Figure 12.9: A 2-D collision with a moving rigid object.

13 LAB: Coefficient of Restitution

Apparatus

Tracker software (free; download from http://www.cabrillo.edu/~dbrown/tracker/)
video: *cue-ball.mov* from our course web site
video: *tennis-ball.mov* from our course web site
video: *puck.mp4* from our course web site

Goal

In this experiment, you will measure the coefficient of restitution for both 1-D and 2-D motion using video analysis.

Procedure

1. Download the file *cue-ball.mov* from the given web site by right-clicking on the link and choosing **Save As...** to save it to your desktop.
2. Open the *Tracker* software on your computer.
3. Use the menu **Video→Import...** to import your video, as shown in Figure 13.1.

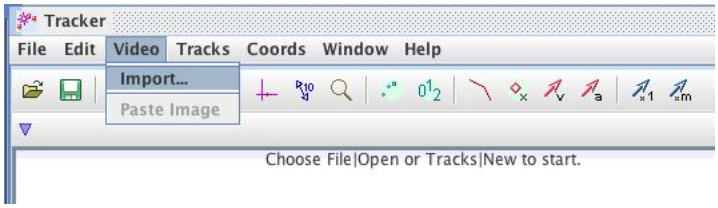


Figure 13.1: Video→Import menu

4. To zoom in or out on the video, click on the toolbar's magnifying glass icon that is shown in Figure 13.2. Zoom in and out on the video to see how it works.

Figure 13.2: The icon used to expand the video.

5. At this point, it's nice to lay out the video and graphs so you can clearly see everything. The middle border between panes can be dragged left and right to make the video pane smaller and graphs larger. The same is true of any other bar that separates panes in the window. Make your application window and video pane as large as possible on your monitor.
6. Note the video controls at the bottom of the video pane. Go ahead and play the video, step it forward, backward, etc. in order to learn how the video controls work. Note the counter that merely shows the

frame number for any frame. Also, click on each of the icons in the video control bar to see what they are used for.

7. Rewind to the first frame of the video. This is the instant that you will begin making measurements of the position of the moving object.

8. This video was recorded at 2000 frames per second. We need to tell Tracker the frame rate so that it can calculate the time correctly. Click the clip settings icon (Figure 13.3). In the **Clip Settings** windows, set the frame rate to 2000 frames per second. Note that Tracker puts in the unit for you.

Figure 13.3: Icon used to change the settings of the video.

9. Since there are many frames of video in this clip, we can skip frames between marking the ball and thus take fewer data points. Click on the **Step Size** button, as shown in Figure 13.4 and change it to **5**.

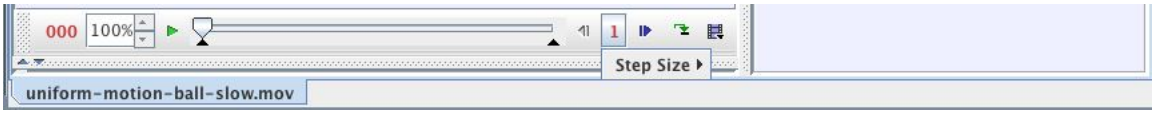


Figure 13.4: Change the step size in order to skip frames.

10. Now, you must calibrate distances measured in the video. In the toolbar, click on the **Tape Measure** icon shown in Figure 13.5 to set the scale for the video.

Figure 13.5: Icon used to set the scale.

11. A blue double-sided arrow will appear. Move the left end of the arrow to the left side of the ball, and move the right end of the arrow to the right side of the ball. Double-click the number that is in the center of the arrow, and enter the diameter of the cue ball, 5.715 cm. (Our units are cm, but Tracker does not use units. You must remember that the number 5.715 is given in cm.)

12. Click the tape measure icon again to hide the blue scale from the video.

13. You now need to define the origin of the coordinate system. In the toolbar, click the **Axes** icon shown in Fig. 13.6 to show the axes of the coordinate system. (By now, you have probably noticed that you can hover the mouse over each icon to see what they do).

Figure 13.6: Icon used to set the coordinate system axes.

14. Click and drag on the video to place the origin of the coordinate system at the location where you would like to define (0,0). You can place the origin at any point you choose, but in this case, it perhaps makes sense to put the origin at the bumper of the pool table. (It won't affect our results.)

15. Click the **Axes** tool again to hide the axes from the video pane. You can click this icon at any time to show or hide the axes.

16. You are ready to add markers to the video to mark the position of the ball. Let's not show the coordinate system and scale. It's too distracting. So, make sure you've clicked the **Axes** and **Tape Measure** icons in the toolbar to hide them.

17. To add markers, click on the Create button and select **Point Mass** . Then, **mass A** will be created, and a new x vs. t graph will appear in a different pane.

    We are going to mark the left edge of the ball and the right edge of the ball. Then we will let Tracker calculate the center of the ball.

18. Click mass A and select **Name...** to change its name to *right side*.

19. **To mark the right side of the ball, hold the SHIFT key down and click once on the middle, right edge of the ball.** You should notice that a marker appears at the position of the ball where you clicked and that the video advances one step.

20. Again, shift-click on the right side of the ball to mark its position. You should now see two marks.

21. Continue marking the right-side of the ball until the last frame of the video. Note that only a few of the marks are shown in the video pane. To display all of the marks or a few of the marks or none of the marks, use the **Set Trail Length** icon shown in Figure 13.7.

Figure 13.7: The Set Trail icon is used to vary the number of marks shown.

Now we will mark the left side of the ball.

22. To add a new marker, click on the Create button and select **Point Mass** . Then, **mass B** will be created.

23. Change the name of mass B to *left side*.

24. Holding the shift key down, mark the middle left side of the ball in all frames.

    Now we will let Tracker calculate the center of the ball.

25. Click the Create button and select **Center of Mass** , as shown in Figure 13.8.

26. You will see a new tab in the Track Control toolbar named **cm**.

27. An additional window will pop up so that you can select the masses. Select both masses "mass A" and "mass B" (or left side and right side) in this window and click OK as shown in Figure 13.9.

28. You will see a track for the center of mass and you will see a graph of x vs. t for the center of mass.

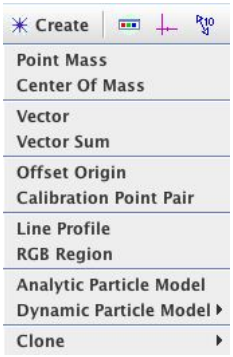


Figure 13.8: Select **Center of Mass** from the menu.

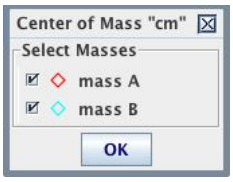


Figure 13.9: Check both masses in this window.

Analysis

x vs. t graph

1. With the graph showing $x(t)$ for the center of mass, right-click on the graph and choose **Analyze** .
2. In the Graph Analysis window, select the part of the graph that occurred before the collision. Do a linear curve fit, and record $v_{i,x}$.
3. Select the part of the graph that occurred after the collision. Do a linear curve fit, and record $v_{f,x}$.
4. Calculate the coefficient of restitution C_R for the collision.

Lab Report

C Complete the experiment and report your results.

B Do all parts for **C**, analyze the motion of the tennis ball in the video *tennis-ball.mov* , and answer the following questions.

1. What is $v_{i,x}$ and $v_{i,y}$ (before the collision with the table)?
2. What is $v_{f,x}$, $v_{f,y}(afterthecollisionwiththetable)$
3. Is the surface frictionless?
4. Is the collision elastic?
5. What is the coefficient of restitution COR for the collision.

A Do all parts for **B**, analyze the motion of the puck in the video *puck.mp4* and answer the following questions. If you want to see the puck traversing around a square air table and colliding with all of the walls, view the video *puck.flv*

1. What is $v_{i,x}$ and $v_{i,y}$ (before the collision with the wall)?
2. What is $v_{f,x}$, $v_{f,y}$ (after the collision with the table)?
3. Is the surface frictionless?
4. Is the collision elastic?
5. What is the coefficient of restitution COR for the collision.

14 GAME – Pong

Apparatus

Computer
GlowScript – www.glowscript.org

Goal

The purpose of this activity is to study the classic arcade game Pong and use GlowScript 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 puck when colliding with the wall and a paddle.

1. Is the collision of the puck and a side wall elastic or inelastic? Explain your answer by referring to observations of the motion of the puck.

2. Are the side walls frictionless or not? Explain your answer by referring to observations of the motion of the puck.

3. Is the collision of the puck and a paddle elastic or inelastic? Explain your answer by referring to observations of the motion of the puck.

4. Is the paddle frictionless or not? Explain your answer by referring to observations of the motion of the puck.

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 puck and refer to it as a puck.

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 puck at the center using the cylinder object. For a cylinder, the axis determines the length (or height) and orientation of the cylinder. In this case, we want a top view of the cylinder so the axis points in the $+z$ direction.

```
puck=cylinder(pos=vector(0,0,0), axis=vector(0,0,0.1), radius=0.5, color=
    color.white)
```

7. Define the initial velocity of the puck (`puck.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 puck.

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

11. Use an `if-elif` statement to check for a collision between the puck and each wall. If there is a collision, change the velocity of the puck in an appropriate way.

12. Run your program and verify that it works properly.

Creating inelastic collisions

In the real world, a puck 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. Create a new program with a different name and copy all of your code to this new program. This way, you are saving your previous work as a reference.

14. Make the left wall a "real wall" that causes the puck to lose speed upon colliding with the wall. In other words, after colliding with the left wall, the puck's perpendicular velocity component would be reduced by a factor less than 1. 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 puck will not bounce off the wall. A COR of 1 is an elastic collision. A COR greater than one is superelastic, like a bumper in pinball.

    Since the left wall is vertical, then the perpendicular component of the puck's velocity is its x component.

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

It helps to use a smaller COR, like 0.5 or less, to notice the effect after one collision.

Describe the motion of the puck 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 puck during the collision, similar to 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 puck.

Adding friction to a collision

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

17. Create a new program with a different name and copy all of your code to this new program. This way, you are saving your previous work as a reference.

18. Start by make all collisions elastic collisions (i.e. $COR = 1$).

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

19. When the puck 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. 20% is a factor of 0.2.)

```
puck.v.y=0.2*puck.v.y
```

20. Run your program.

Describe the motion of the puck 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 program. We will start with a blank page.

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 puck and make its initial velocity something like (15,12,0) m/s.

26. Create variables for the clock and time step.

    A this point, there is no motion, and your scene should look something like Fig. 14.1.

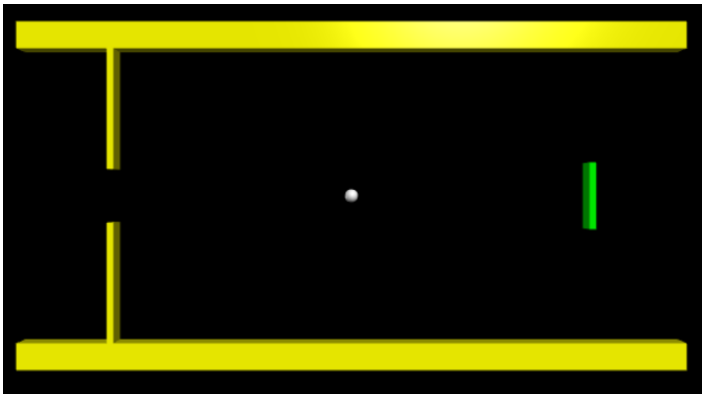


Figure 14.1: The initial scene of our version of Pong.

27. Create an infinite while loop. Update the position of the puck. To check for collisions with the walls and paddle, set the initial velocity of the puck so it collides off the various objects. For now, assume elastic, frictionless collisions. Run your program, check every wall and the paddle, and verify that everything works as expected.

Controlling the paddle

Think about how you want to control the paddle. You can use the up and down arrow on the keyboard, click and drag with the mouse, or simply hover with the mouse and move the paddle up and down in sync with the mouse as the mouse moves up and down. All of these ideas are possible, but with different levels of difficulty and experience required. We will explore these options.

Controlling the paddle with the keyboard

28. Create a new program with a different name and copy all of your previous code to this new program. This way, you are saving your previous work as a reference.

29. We are going to make the paddle move up and down by pressing the up and down arrow keys. Therefore, create a velocity for the paddle and set its initial value to zero. At the same place in the code where you define the puck's velocity, define the paddle's velocity. I called my object `paddle2` so you will have to be consistent with using the name of your paddle object.

```
paddle2.v=vector(0,0,0)
```

30. We will have to update the position of the paddle (i.e. make it move). In the same place in the code where you update the position of the puck (inside the `while` loop) , add a line to update the position of the paddle.

```
    paddle2.pos=paddle2.pos+paddle2.v*dt
```

31. Now you need to create a function that checks for a keypress of the up and down arrow keys. Near the top of the program, at approximately line 3, add the following `movePaddlewithKeyboard` function. Read what it does! If an up or down arrow key is pressed, it sets the velocity of the paddle to be upward or downward. When the key is released (`keyup`) The `scene.bind` function tells GlowScript to look for a `keydown` or `keyup` event and called the `movePaddlewithKeyboard` function. Note that the name of the function `movePaddlewithKeyboard` is not important. I could have named this anything I want. In fact, I happened to pick a really long name, but at least it's descriptive.

```
def movePaddle(event):
#     print(event.type, event.which)
    if event.type=='keydown':
            k = event.which
            if k == 38:
                paddle2.v=50*vector(0,1,0)
            elif k == 40:
                paddle2.v=50*vector(0,-1,0)
    elif event.type=='keyup':
            paddle2.v=vector(0,0,0)

scene.bind('keydown keyup', movePaddle)
```

32. Run the program and see if you can control the paddle. You might find that it is quite dissatisfying. For example, the paddle can go through the top and bottom walls. Also, it is hard to stop the paddle at just the right moment to collide with the puck. To improve the paddle control, adjust the paddle's velocity that is set in this function.

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 puck goes past the paddle or through the goal, increment a score, reset the puck 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. This function pauses the program and waits for a mouse click to continue.

```
scene.waitfor('click')
```

2. Change your program so that the puck 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 puck to bounce in this way.

15 LAB: Video Analysis of a Fan Cart

Apparatus

Tracker software (free; download from http://www.cabrillo.edu/~dbrown/tracker/)
video: *SpeedAway.mov* available from our course web site.
video: *SpeedTo.mov* available from our course web site.
video: *SloToward.mov* available from our course web site.

Goal

In this experiment, you will measure and graph the x-velocity of a cart as a function of time as the cart is accelerating.

Speeding up to the right

Procedure

In all cases, we will define the $+x$ direction to be to the right. When describing the direction of the velocity of the cart, *positive* means that the cart is moving to the right and *negative* means that it is moving to the left.

1. Download the file `SpeedAway.mov` by right-clicking on the link and choosing **Save As...** to save it to your desktop.
2. Open the *Tracker* software on your computer.
3. Use the menu **Video→Import...** to import your video, as shown in Figure 15.1.

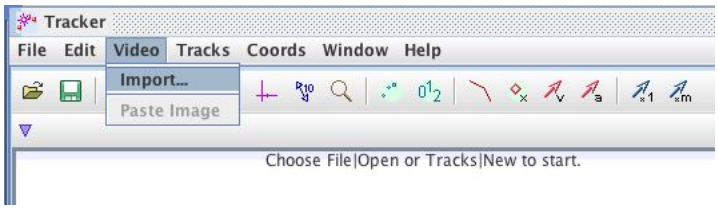


Figure 15.1: Video→Import menu

4. Play the video and watch the motion of the two carts.

Describe in words the motion of the cart.

5. You now need to define the origin of the coordinate system. In the toolbar, click the **Axes** icon shown in Fig. 15.2 to show the axes of the coordinate system.

Figure 15.2: Icon used to set the coordinate system axes.

6. Click and drag on the video to place the origin of the coordinate system. Let's define $x = 0$ to be the location of the motion detector at the left end of the track.

7. Next, we will use the meterstick to set the scale for the video. It is 1.0 m. In the toolbar, click on the **Calibration** icon shown in Figure 15.3 to set the scale for the video.

Figure 15.3: Icon used to set the scale.

8. A blue double-sided arrow will appear. Move the left end of the arrow to the left end of the meterstick, and move the right end of the arrow to the right end of the meterstick. Double-click the number that is in the center of the arrow, and enter the length of the meterstick 1.0. (Our units are meters, but Tracker does not use units. You must remember that the number 1.0 is given in meters.)

9. Click the tape measure icon again to hide the blue scale from the video.

10. To mark the position of the fancart in each frame, first click on the Create button and select **Point Mass** . Then, **mass A** will be created, and a new x vs. t graph will appear in a different pane. You will now be able to mark the position of the cart which will be referred to as **mass A**.

11. Now, to mark the position of the fancart, hold down the shift key and click on the red dot on the fancart. (This is called a shift-click). The video will advance one frame. Continue to shift-click on the red dot on the fancart until you have marked the location of the cart in all frames of the video.

12. It's possible that only a few of the marks are shown in the video pane. To display all of the marks or a few of the marks or none of the marks, use the **Set Trail Length** icon shown in Figure 15.4. You can select no trail, short trail, and full trail which will show you no marks, a few marks, or all marks, respectively.

Figure 15.4: The Set Trail icon is used to vary the number of marks shown.

Analysis

1. On the graph, click on the vertical axis variable and select the x-velocity. View the v_x vs. t graph.

2. Play the video. (You can hide the marks if you wish by clicking the **Show or hide positions** icon, and you can show the path by clicking the **Show or hide paths** icon. Both of these icons are in the

toolbar.) Note how the graph and video are synced. The data point corresponding to the given video frame is shown in the graph using a filled rectangle.

Also, when you click on a data point on the graph, the video moves to the corresponding frame.

Describe in words the type of function that describes this graph of v_x vs. t? (i.e. linear, quadratic, square root, sinusoidal, etc.)

According to the v_x vs. t graph, is the x-velocity constant, increasing, or decreasing? Explain your answer.

3. Right-click on the graph (or ctrl-click for Mac users) and select **Analyze...** . In the resulting window, check the checkbox for **Fit** , and additional input boxes will appear. Select the linear curve fit. Check the checkbox for **Autofit** and the best-fit curve will appear in the graph.

Neatly sketch the graph, including axes and labels, below.

Record the function and the values of the constants for your curve fit. Write the function for $v_x(t)$, with the appropriate constants (also called fit parameters).

What does the slope tell you and what are its units?

What does the intercept tell you and what are its units?

Speeding up to the left

Procedure

1. Download the video *SpeedTo.mov* . Play the video.

Describe its motion in words.

Sketch a prediction of what you think that the x-velocity vs. time graph will be. Think carefully about this before you move on.

2. Analyze this video. Fit a curve to the x-velocity vs. time graph.

Analysis

Sketch the $v_x(t)$ graph and record the curve fit.

What is the acceleration of the cart?

What is the initial velocity of the cart?

What does the sign of the initial velocity tell you?

Some people think that a negative acceleration means that the object is slowing down. Is this idea consistent with what you measured for the acceleration?

Does the acceleration and initial velocity have the same sign or different signs?

Slowing down to the left

Procedure

1. Download the video *SloToward.mov* . Play the video.

Describe the cart's motion in words.

Sketch a prediction of what you think that the x-velocity vs. time graph will be. Think carefully about this before you move on.

2. Analyze this video. Fit a curve to the x-velocity vs. time graph.

Analysis

Sketch the $v_x(t)$ graph and record the curve fit.

What is the acceleration of the cart?

What is the initial velocity of the cart?

What does the sign of the initial velocity tell you?

Some people think that a negative acceleration means that the object is slowing down. Is this idea consistent with what you measured for the acceleration?

Does the acceleration and initial velocity have the same sign or different signs?

Lab Report

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

1. In the video *SpeedAway.mov* , what was the acceleration and initial velocity of the cart?
2. In the video *SpeedTo.mov* , what was the acceleration and initial velocity of the cart?
3. In the video *SloToward.mov* , what was the acceleration and initial velocity of the cart?

B Do all parts for **C** and answer the following questions.

1. In general, how do you get the initial velocity from a $v_x(t)$ graph that is linear?
2. In general, what does the slope of a $v_x(t)$ graph tell you and what are its units if velocity is in m/s and time is in s?
3. If you only know that an object has a positive acceleration (and you know nothing else), can you say whether it is speeding up or slowing down?
4. If you only know that an object has a negative acceleration (and you know nothing else), can you say whether it is speeding up or slowing down?
5. If you are told whether the initial velocity is positive or negative and if you are told whether the acceleration is positive or negative, how can you know whether the object is speeding up or slowing down?

A Do all parts for **B** and answer the following questions.

1. A car is traveling in the -x direction when the driver pushes the brakes. Is the car's acceleration positive or negative?
2. A car is traveling in the -x direction when the driver pushes the gas pedal to the floor. Is the car's acceleration positive or negative?
3. A car is moving at a x-velocity of 30 m/s and a clock reads 12:35:25 PM when the driver hits the brakes and slows down. When her x-velocity is 10 m/s, a clock reads 12:35:35 PM. What is her acceleration during this time interval?
4. Driver A is moving with a x-velocity of 25 m/s when she hits the brakes and comes to a stop. It takes 15 s for her come to rest. Driver B is moving with a x-velocity of 25 m/s when she hits a barrier head-on and comes to rest in 2 s. which driver has a greater acceleration?

16 Newton's Second Law

Acceleration

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

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

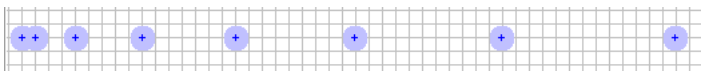
Since velocity is a vector with both magnitude and direction, an object as non-zero acceleration if the (1) magnitude of velocity changes; (2) the direction of velocity changes; or (3) both magnitude and direction of velocity changes.

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 16.1. Its velocity at two different instances of time is indicated. What is the direction of its acceleration during this time interval?

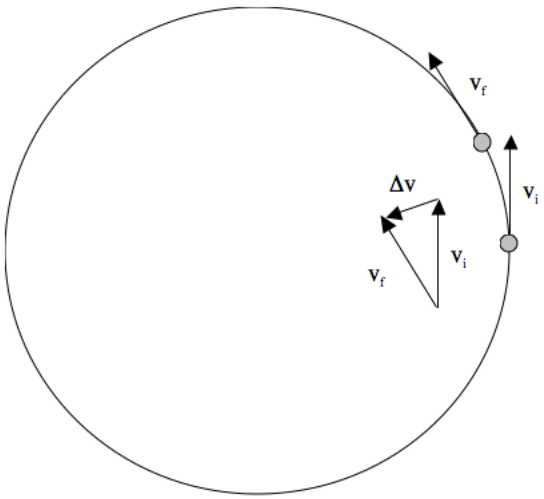
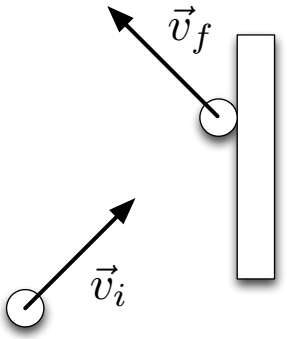


Figure 16.1: Velocity at two locations as it moves along a circle at constant speed.

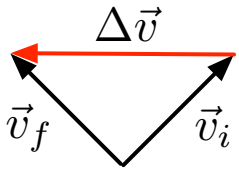
If you draw the velocity vectors at any locations close together on the circle, you will find that the acceleration vector points toward the center. This is a general observation, *the acceleration of an object that travels in a circle at a constant speed points toward the center of the circle.*

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 $t = 0 + 0.1$ s $= 0.1$ 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 $t = 0.1 + 0.1$ s $= 0.2$ 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.

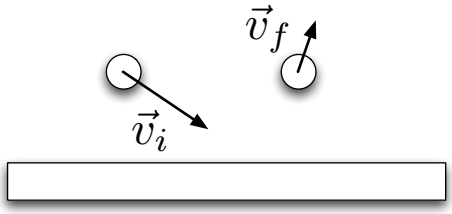


Figure 16.2: Velocity before and after a football bounces off the grass.

   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. The mass of the ball is 0.2 kg and the force on the ball is the gravitational force by Earth on the ball, $\vec{F}_{net} = m\vec{g}$, where $\vec{g} = (0, -10, 0)$ N/kg at the surface of Earth. The variable $\vec{g}$ is called Earth's *gravitational field strength*.

   (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.

17 PROGRAM – Modeling motion of a fan-cart

Apparatus

GlowScript
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.

```
GlowScript 2.1 VPython

track = box(pos=vector(0,-0.05,0), size=vector(3.0,0.05,0.1), color=color.white)
cart = box(pos=vector(-1.4,0,0), size=vector(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.waitfor("click")

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 (i.e. between lines 15 and 16), 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 17.1. 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.

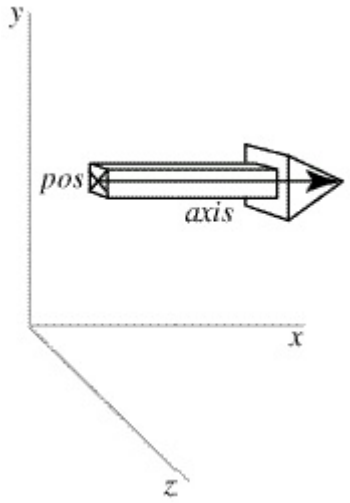


Figure 17.1: A arrow in VPython.

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.

18 GAME – Lunar Lander

Apparatus

Computer
GlowScript – www.glowscript.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 is constant. In that case, the object 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 in each iteration of the loop are to:

1. calculate the net force (although in some cases it is constant)
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*).

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 but do not run the code (since lines are missing). It is fine if the first line references a more recent version of GlowScript.

```
GlowScript 2.1 VPython

scene.range=20

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

```

```
8  ball.m = 1
9  ball.v = vector(0,0,0)
10 g=vector(0,-10,0)
11
12 dt = 0.01
13 t = 0
14
15 scale=1
16 FgravArrow = arrow(pos=ball.pos, axis=scale*ball.m*g, color=color.red)
17
18 scene.waitfor("click")
19
20 while 1:
21     rate(100)
22 #    Fgrav=
23 #    Fnet=
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?

2. Line 22 should compute the gravitational force on the ball. Fill in this line using the variables for the mass of the ball and Earth's gravitational field.

3. Line 23 is the net force on the ball. This is computed by summing all forces on the ball. But the only force on the ball in this case is the gravitational force. Fill in line 23 with the variable representing the gravitational force on the ball.

4. Line 24 updates the velocity of the ball and line 25 updates the position. Fill in each of these lines with the appropriate calculation for updating the velocity and position of the ball. Refer to the previous chapter on the fancart if you forget how to do this.

5. Run your program and make sure it shows a bouncing ball.

What is the purpose of lines 26 and 27?

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.

6. 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 primary difference in the motion of a ball dropped on the Moon and a ball dropped from the same height on Earth? In other words, if you were to see an animation of each ball, side by side, how would you know which animation is of the ball on the Moon?

Moon Lander

We will now model the motion of a lunar module that is landing on the moon.

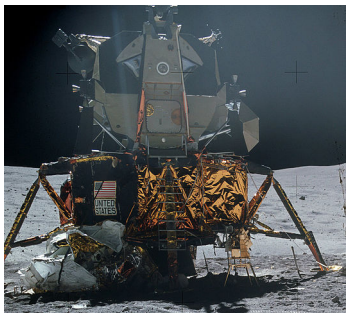


Figure 18.1: Apollo 16 LM *Orion*

7. Start a new program and type the following code into GlowScript.

```
GlowScript 2.1 VPython

scene.range=20
```

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

8. Fill in lines 20-23 with the appropriate expressions.

9. 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)
```

10. 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=scale*Fthrust, color=color.
    cyan)
```

11. 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
```

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

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

13. 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?

Let's use the keyboard to turn on and off the engine. In this case, "on" means that the vertical thrust is $(0, 4, 0)$ and "off" means that the thrust is zero, $(0, 0, 0)$.

14. Create a new program. Copy and paste your last program into this new file, and set the value of `Fthrust` to zero, $(0, 0, 0)$. Run your program and verify that the lunar module accelerates downward and stops when reaching the Moon's surface. (Make sure that $g = (0, -10/6, 0)$ N/kg.

15. Near the top of your program at approximately line 2, add the following function.

```
##add keyboard control
def process(event):
    global Fthrust
    if event.type=='keydown':
        k = event.which
        if k == 38: #up arrow turns on the vertical thruster
            Fthrust=vector(0,4,0)
    elif event.type=='keyup': #releasing the key turns off the thruster
        Fthrust=vector(0,0,0)

    FthrustArrow.axis=scale*Fthrust

scene.bind('keydown keyup', process)
```

What key is used to turn on the thruster? What causes the thruster to turn off?

16. Run your program and verify that it works. Use the up arrow key to control the thruster. Land the lunar module as gently as possible on the Moon's surface.

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 a minimum requirement (like 1 m/s), print "You lose."
2. If the speed of the spaceship is less than this minimum, print "You win."

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

1. 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.
2. Create an engine that fires in the $-x$ direction (the engine is on the right so the arrow points to the left) when the left arrow key is pressed.
3. Place a target on the ground.
4. Check that the lunar module lands on the target.
5. 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.
6. Since you don't want to waste fuel, assign points based on the time elapsed and cause the player to lose if the clock reading exceeds some amount. If you want, you can create a timer that starts at some value like 20 s and counts down to zero.

19 LAB: Video Analysis of Projectile Motion

Apparatus

Tracker software (free; download from http://www.cabrillo.edu/∼dbrown/tracker/)
video: `basketball.mov` from our course web site.

Goal

In this experiment, you will measure and graph the x-position, y-position, x-velocity, and y-velocity of a projectile, in this case a basketball as it moves freely through air with negligible air resistance.

Introduction

Suppose a projectile moves along a parabolic path. Fig. 19.1 shows an object at intervals of 1/30 s between the first image A and the last image I.

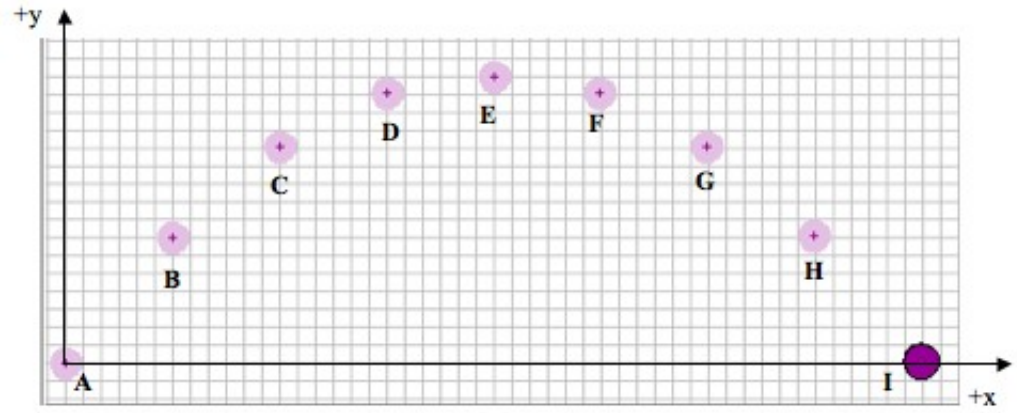


Figure 19.1: Position of a projectile at equal time intervals of 1/30 s.

Suppose that we define $t = 0$ to occur at the first position of the object. In Fig. 19.1, label the time t for each subsequent position of the object.

Draw a vertical line at the location of each image. Where each vertical line intersects the x-axis, sketch a circle on the x-axis at the x-coordinate of each image in Fig. 19.1. An example is shown in Fig. 19.2.

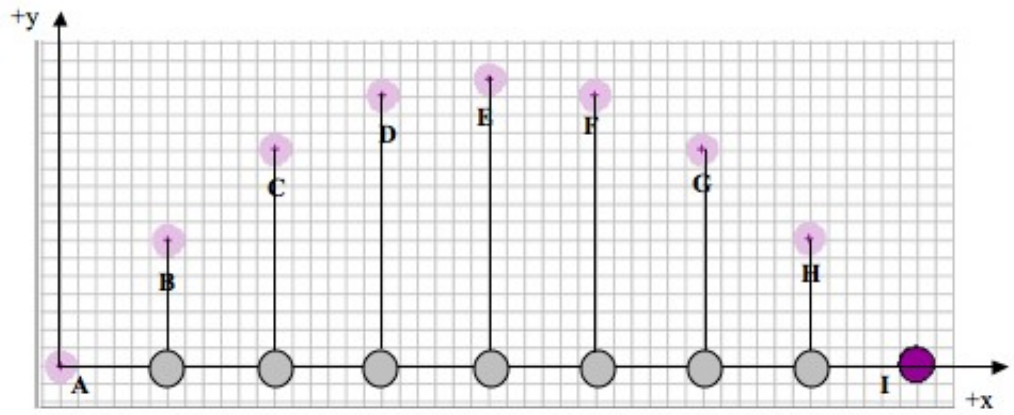


Figure 19.2: x-positions of the projectile.

By examining the x-coordinate of the object, is the x-motion characterized by uniform motion (i.e. zero net force) or constant acceleration (i.e. constant net force)? Give a reason for your answer.

If you were to graph x vs. t, what would it look like? Draw a sketch.

In Fig. 19.1, mark the y-position of the projectile by drawing horizontal lines from the object to the y-axis and drawing a circle where this line intersects the y-axis. Is the y-motion characterized by uniform motion (zero net force) or constant acceleration (constant net force)? Give a reason for your answer.

If you were to graph y vs. t, what would it look like? Draw a sketch.

Procedure

1. Download the file `basketball.mov` by right-clicking on the link and choosing **Save As...** to save it to your desktop.
2. Open the *Tracker* software on your computer.
3. Use the menu **Video→Import...** to import your video, as shown in Figure 19.3.

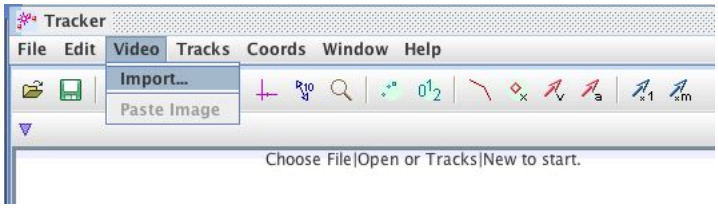


Figure 19.3: Video→Import menu

4. Play the video and watch the motion of the basketball.

What is the shape of its path?

After the ball leaves the player's hand, what interacts with the ball and in what direction does it exert a force on the ball?

5. Use the video control bar to advance the video to the first frame after the ball leaves the player's hand. We are only studying the motion of the ball while it is in the air, not while it is in his hand. Use the Video Settings to make this the first frame of the video. Also set the last frame of the video to be the frame when the ball hits the floor.
6. You now need to define the origin of the coordinate system. In the toolbar, click the **Axes** icon shown in Fig. 19.4 to show the axes of the coordinate system.

Figure 19.4: Icon used to set the coordinate system axes.

7. Click and drag on the video to place the origin of the coordinate system at the location where you would like to define (0,0). For consistency with your classmates, place the origin on the floor at the center of the player's feet.

8. We will use the 2-m long stick that is along the base of the wall to set the scale for the video. In the toolbar, click on the **Tape Measure** icon shown in Figure 19.5 to set the scale for the video.

Figure 19.5: Icon used to set the scale.

9. A blue double-sided arrow will appear. Move the left end of the arrow to the left end of the stick, and move the right end of the arrow to the right end of the stick. Double-click the number that is in the center of the arrow, and enter the length of the stick (2.0). (Our units are meters, but Tracker does not use units. You must remember that the number 2.0 is given in meters.) The scale will appear as shown in Figure 19.6.

Figure 19.6: Enter the length of the stick, 2.0 m.

10. Click the tape measure icon again to hide the blue scale from the video.

11. To mark the position of the basketball in each frame, first click on the Create button and select **Point Mass** . Then, **mass A** will be created, and a new x vs. t graph will appear in a different pane. You will now be able to mark the position of the ball which will be referred to as **mass A**.

12. Now, to mark the position of the basketball, hold down the shift key and click on the ball. (This is called a shift-click). The video will advance one frame. Continue to shift-click on the ball until you have marked the location of the ball in all frames of the video.

    Keep in mind that you only want to analyze frames where the ball is in the air. Therefore, you should have skipped the first few frames when it is in the player's hand, and you should stop marking the position of the ball just before it hits the floor.

13. It's possible that only a few of the marks are shown in the video pane. To display all of the marks or a few of the marks or none of the marks, use the **Set Trail Length** icon shown in Figure 19.7. Clicking this icon continuously will cycle through no trail, short trail, and full trail which will show you no marks, a few marks, or all marks.

Figure 19.7: The Set Trail icon is used to vary the number of marks shown.

A picture of the basketball with marks is shown in Figure 19.8.

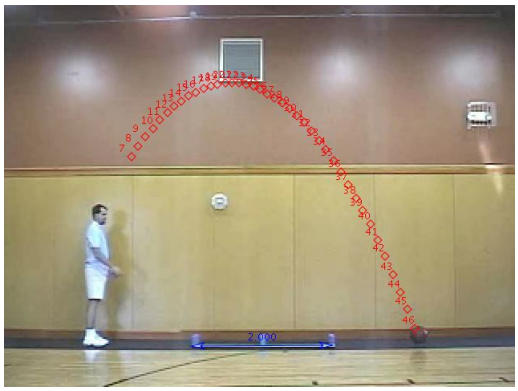


Figure 19.8: Marks showing the motion of the basketball.

Analysis

1. View the x vs. t graph. You can click and drag the border of the video pane to make it smaller so that you can focus on the graph.

2. Play the video. (You can hide the marks if you wish by clicking the **Show or hide positions** icon, and you can show the path by clicking the **Show or hide paths** icon. Both of these icons are in the toolbar.) Note how the graph and video are synced. The data point corresponding to the given video frame is shown in the graph using a filled rectangle.

   Also, when you click on a data point on the graph, the video moves to the corresponding frame.

Describe in words the type of function that describes this graph of x vs. t? (i.e. linear, quadratic, square root, sinusoidal, etc.)

According to the x vs. t graph, is the x-velocity constant, increasing, or decreasing? Explain your answer.

Based on this graph, what is the x-component of the net force on the ball while it is in the air, $F_{net,x}$?

3. Right-click on the graph (or ctrl-click for Mac users) and select **Analyze...** . In the resulting window,

check the checkbox for **Fit** , and additional input boxes will appear, as shown in Figure 19.9. Select the Fit Name "Line," and the equation will be $x = a*t+b$ where a and b are coefficients (or parameters) of the curve fit. Check the checkbox for **Autofit** and the best-fit curve will appear in the graph.

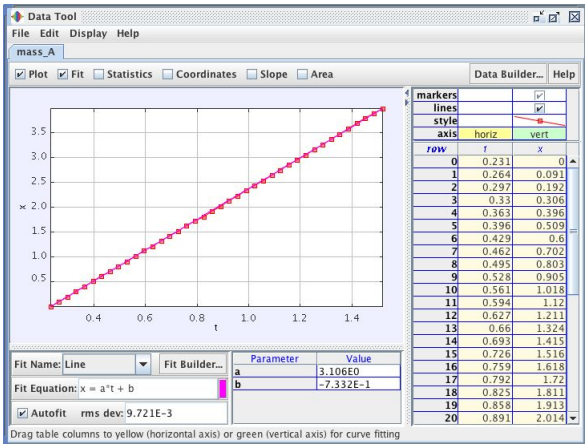


Figure 19.9: The Data Tool for finding the best-fit curve for the data.

4. Find the best-fit curve for the graph. Be sure to select the linear fit.

Neatly sketch the graph, including axes and labels, below.

Record the function and the values of the constants for your curve fit. Write the function for x(t), with the appropriate constants (also called fit parameters).

What do you expect the x-velocity vs. time graph to look like? Sketch your prediction below.

5. Close the Data Tool window and return to the main Tracker window. Click on the label of the vertical axis on the graph and select **vx** .

What function describes this graph of v_x vs. t? (i.e. linear, quadratic, square root, sinusoidal, etc.)

6. Right-click the graph and select **Analyze** to analyze the v_x vs. t graph. You may notice that the data for both x and v_x are displayed on the same graph. If this occurs, uncheck the checkboxes for x in the upper right corner of the window.

Neatly sketch the graph of v_x vs. t, including axes and labels, below.

Use this graph and data to determine v_x. Describe in detail how you determined v_x.

7. Close this window and return to the main window. Change the label on the vertical axis to y vs. t and examine this graph.

According to the y vs. t graph, during what time interval is the magnitude of the y-velocity decreasing? During what time interval is the magnitude of the y-velocity increasing? At what time is the y-velocity zero?

8. Change the label on the vertical axis to v_y vs. t and examine this graph.

Based on this graph, is the y-component of the net force on the ball, $F_{net,y}$, zero or constant (non-zero)? If it is constant (and non-zero), explain your reasoning and say whether the net force on the ball is in the +y or -y direction.

9. Right-click the graph and select **Analyze** to analyze the v_y vs. t graph. Fit a curve to the data and record the best-fit function, including the fit parameters. An example is shown in Figure 19.10.

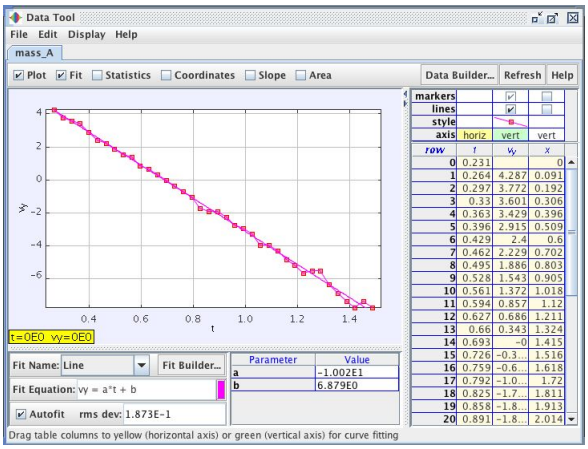


Figure 19.10: The Data Tool for finding the best-fit curve for the v_y vs. t data.

What is the y-velocity v_y of the ball at the moment it leaves the player's hand?

What is the ball's y-acceleration, a_y? Since the mass of a basketball is about 0.62 kg, also calculate the y-component of the net force on the basketball, $F_{net,y}$.

Lab Report

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

1. What is the x-component of the net force on the ball? (Explain your reasoning or show your calculation.)
2. What is the x-velocity of the basketball as determined by the $x(t)$ graph? (Explain your reasoning.)
3. What is the y-acceleration of the ball as determined by the $v_y(t)$ graph? (Explain your reasoning.)
4. What is the y-component of the net force on the ball? (Explain your reasoning or show your calculation.)

B Do all parts for **C** and answer the following questions.

1. What is the velocity of the ball at the instant it leaves the player's hand ($t = 0$)? Write it as a vector and sketch it.
2. What is the position (x and y components) of the basketball at the first instant that it leaves the player's hand ($t = 0$)?
3. At what clock reading t is the ball at its peak and explain how you can get this independently by looking at the $y(t)$ graph and by looking at the $v_y(t)$ graph.
4. What is the velocity of the ball when it is at its peak? Express your answer as a vector.

A Do all parts for **B** and answer the following questions.

1. Starting with Newton's second law and the initial position and velocity of the ball, calculate the position and velocity of the ball at 0.05 s time steps. What is its position and velocity at $t = 0.25$ s?
2. Compare your theoretical calculation in the previous question with the actual data for position and velocity at or near $t = 0.25$ s. How do they compare? (Note that the iterative calculation is approximate and is only accurate for very small time steps. There may be some discrepancy since we used a larger time step.)
3. Go back to your iterative calculation and continue until the ball reaches its maximum height (i.e. $v_y \approx 0$). What is the approximate clock reading t when the ball reaches its peak? Compare your theoretical calculation with what you measured. (Again there may be some discrepancy due to the fact that the iterative method is an approximation.)

20 LAB: Angry Birds

Apparatus

Tracker software (free; download from http://www.cabrillo.edu/∼dbrown/tracker/)
video: `Angry_Birds.mp4` from our course web site.
Tracker file: `Angry_Birds_projectile.trk` from our course web site.

Goal

In this experiment, you will measure the motion of a bird in Angry Birds and will assume $g = 10$ m/s^2 in order to calibrate distance in the video. You will also see if the resulting motion of the bird is consistent with projectile motion. In other words, has the programmer of Angry Birds used correct physics in the game?

Introduction

In the game Angry Birds, birds are launched with a slingshot. Is their motion described by ideal projectile motion? If so, what is the acceleration of a bird and can we use its acceleration to figure out the length of a bird?

When we do video analysis, we typically use an object of known length in the video to calibrate the video and determine how many pixels is 1 m. In the case of Angry Birds, instead of scaling the video with a known object on the screen, we can scale the video by the acceleration due to gravity, assuming the Angry Birds world is Earth. That is, we can assume that $a_y = -g = -10$ m/s^2, then calculate the scaling factor, and use the scaling factor determine the length of objects in the video in units of meters.

Procedure

1. Download the file *Angry_ Birds.mp4* from our course web site.

2. Download the file *Angry_ Birds_ projectile.trk* from our course web site.

3. The *.trk* file is a partially marked Tracker file. Open the Tracker software. Go to **File→Open File...** and select the file *Angry_ Birds_ projectile.trk* . It should load the video or ask you where it is located.

4. Play the video and notice that the "camera" both moves (i.e. pans) and zooms. This makes analyzing the video more difficult than what you've encountered before.

   In order to track the bird, we will need a fixed origin (the sling slot), and since the origin goes off screen, we need an offset point (the distance from the sling shot to a blade of grass that shows up for most of the trajectory of the bird).

   We also need a set length since the movie zooms in and out. It turns out that the height to the fork of the slingshot is the same as the height of the first pedestal the pig sits on. We will define the height of the fork of the slingshot as "1" *slingshot* in the Tracker file. So, even as the image zooms and pans, the length of the slingshot and the pig's pedestal is always "1" and the location of the origin is set. DO NOT adjust the "Coordinate Offset" or the "Calibration Stick" or the data will no longer account for the panning and zooming of the camera.

Remember that our unit of distance in the video will be *slingshots* because this is the height of the slingshot from its base to the fork. Thus, all distances are in *slingshots* and all velocities are in $slingshots/s$.

The Tracker file already has the position of the angry bird marked. The track of the marked points is not a parabola on the video. Why not?

5. Verify that the graph plots $y(x)$. If necessary, click on the vertical axis variable and change it to y, and click on the horizontal axis variable and change it to x. This will display the calculated path of the bird after accounting for the zooming and panning of the camera.

What is the calculated path of the bird? Describe it in words and sketch it.

Explain why some points are missing.

If the bird follows correct physics, what should a graph of x(t) look like? Sketch it below.

6. Change the vertical axis variable to x and change the horizontal axis variable to t.

Explain why the plot of x(t) is a straight line.

What is the best-fit function for $x(t)$?

From this best-fit function, what is v_x in units of slingshots/s?

7. Change the vertical axis variable to y and change the horizontal axis variable to t. Notice that it is parabolic. At first the slope decreases, showing that the bird slows down as it rises. Then the slope increases, showing that the bird speeds up as it falls.

8. Change the vertical axis variable to v_y. Due to measurement error there is variation in the data; however, it generally appears linear. Do a linear fit.

What is the best-fit function for the $v_y(t)$ graph?

From the best-fit function, what is the y-acceleration in units of slingshots/s^2?

What is the initial y-velocity in slingshots/s?

By comparing the measured y-acceleration in slingshots/s^2 to free-acceleration on Earth of -10 m/s^2, how many slingshots are in 10 m?

How many meters is 1 slingshot?

Use the tape measure to measure the radius of the bird in units of slingshots. Convert this to meters. Is this a realistic size for a bird?

Convert the x-velocity and initial y-velocity to m/s. What is the initial velocity vector in m/s? What is the initial speed of the bird in m/s?

Lab Report

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

1. What is the x-velocity of the bird in units of slingshots/s?
2. Is the x-velocity of the bird increasing, decreasing, or constant?
3. What is the x-acceleration?
4. What is the initial y-velocity of the bird in units of slingshots/s?
5. Is the y-velocity of the bird increasing, decreasing, or constant?
6. What is the y-acceleration of the bird in units of slingshots/s^2?

B Do all parts for **C** and answer the following questions.

1. Generally, the videos that we analyze have an object of known distance. In this case, what is the "known" quantity that we used to determine distance in meters in the video?
2. How many meters are in 1 slingshot?
3. What is the radius of the bird in units of meters?
4. What is the initial velocity vector of the bird in m/s?
5. What is the initial speed of the bird in m/s?

A Do all parts for **B** and answer the following questions.

1. Suppose that you analyze a video screen capture of a tank wars game. You use the length of the tank as the unit "1 tank" and find that the free-fall y-acceleration in the world of the game is $-2.5\ tanks/s^2$. How many meters long is the tank?
2. You analyze a bullet in the tank wars game. Its x(t) graph with x in meters is shown below. What is the x-velocity of the projectile?

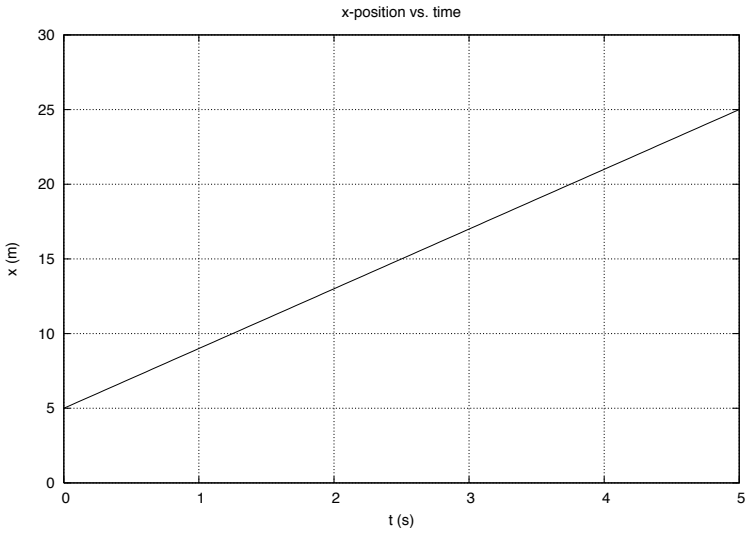


Figure 20.1: x(t) for a projectile.

3. What is the x-acceleration of the projectile?
4. The projectile's $v_y(t)$ graph is shown below. What is the initial y-velocity of the projectile?
5. What is the y-acceleration of the projectile?

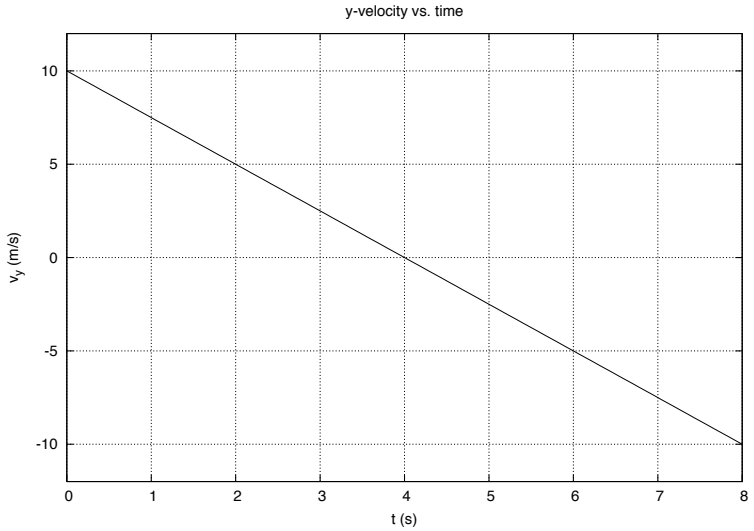


Figure 20.2: $v_y(t)$ for a projectile.

6. At what time (i.e. clock reading) did the projectile reach its peak?
7. What is the initial velocity vector of the projectile?
8. What is the initial launch speed of the projectile?

21 GAME – Tank Wars

Apparatus

Computer
GlowScript – www.glowscript.org

Goal

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

Procedure

1. Create a new file in Glowscript and copy and paste the following code template from our Trinket site. For your reference, the program is printed below, but it will be much faster to copy and paste from our Trinket site.

```
GlowScript 2.1 VPython

def control(event):
    global theta, muzzlespeed, bulletsList
    if event.type=='keydown':
        k = event.which
        if k == 38:
            theta=theta+dtheta
            if theta>rad(90):
                theta=rad(90)
            turret.axis=L*vector(cos(theta),sin(theta),0)
            angleBar.axis=(5*theta)*vector(1,0,0)
        elif k == 40:
            theta=theta-dtheta
            if theta<0:
                theta=0
            turret.axis=L*vector(cos(theta),sin(theta),0)
            angleBar.axis=(5*theta)*vector(1,0,0)
        elif k == 37:
            muzzlespeed=muzzlespeed-dspeed
            if muzzlespeed<1:
                muzzlespeed=1
            speedBar.axis=(muzzlespeed/2+0.5)*vector(1,0,0)
        elif k == 39:
            muzzlespeed=muzzlespeed+dspeed
            if muzzlespeed>20:
                muzzlespeed=20
            speedBar.axis=(muzzlespeed/2+0.5)*vector(1,0,0)
        elif k==32:
```

```
30              bullet=sphere(pos=turret.pos+turret.axis, radius=0.5, color=
                    color.white)
31              bullet.v=muzzlespeed*vector(cos(theta),sin(theta),0)
32              bulletsList.append(bullet)
33
34  scene.bind('keydown', control)
35
36  def rad(degrees): #converts an angle in degrees to an angle in radians
37      radians=degrees*pi/180
38      return radians
39
40  scene.range=20
41  scene.width=600
42  scene.height=400
43
44  #create objects
45  ground = box(pos=vector(0,-15,0), size=vector(60,2,2), color=color.green)
46  tank = box(pos=vector(-18,-13,0), size=vector(2,2,2), color=color.yellow)
47  turret = cylinder(pos=tank.pos, axis=vector(0,0,0), radius=0.5, color=tank
        .color)
48  turret.pos.y=turret.pos.y+tank.height/2
49  angleBar = cylinder(pos=vector(-18,-19,0), axis=vector(1,0,0), radius=1,
        color=color.magenta)
50  speedBar = cylinder(pos=vector(5,-19,0), axis=vector(1,0,0), radius=1,
        color=color.cyan)
51
52  #turret
53  theta=rad(45)
54  dtheta=rad(1)
55  L=3
56  turret.axis=L*vector(cos(theta),sin(theta),0)
57
58  #bullets
59  bulletsList=[]
60  m=1
61  muzzlespeed=15
62  dspeed=1
63
64  #Bar
65  angleBar.axis=(5*theta)*vector(1,0,0)
66  speedBar.axis=(muzzlespeed/2+0.5)*vector(1,0,0)
67
68  #motion
69  g=vector(0,-10,0)
70  dt = 0.01
71  t = 0
72
73  while True:
74      rate(100)
75
76      for thisbullet in bulletsList:
77          if(thisbullet.pos.y<ground.pos.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=thisbullet.v+thisbullet.Fnet/m*dt
83          thisbullet.pos=thisbullet.pos+thisbullet.v*dt
84
85      t=t+dt
```

2. Run the program above. Then, study it and use it to answer the following questions.

What line updates the velocity of the bullet?

What line updates the position of the bullet?

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 increase or decrease when you press the right or left arrow key only once? (And what is this variable called?)

What is the variable name for the angle the bullet is launched at? How much does the angle increase or decrease when you press the up or down arrow key once? 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.

When the bullet is in the air, what is the net force on the bullet? Cite the particular line number that supports your answer.

What key strokes are used to change the launch angle?

What key strokes are used to change the launch speed?

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

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

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

Creating a Target

3. 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. (You can think of this as the opposing player's tank, perhaps.)

   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` .

4. 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. Again, this code snippet is on our Trinket site if you want to copy and paste.

```
#determines whether a sphere and box intersect or not
#returns boolean
def collisionSphereAndBox(sphereObj, boxObj):
    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
    else:
        result=False
    return result
```

5. At the `if` statement at line 77 in the original template above, 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!")
```

6. 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 would like to effectively reset the game to its initial state. To do this we will need to:

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

(b) Reset the bulletsList to an empty list.

7. Inside the `elif` statement that you just created, after you print "direct hit" add the following lines:

```
            scene.waitfor("click")

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

            break
```

   These lines will pause the game until the user clicks the scene. Then it will make each bullet invisible. Then, it creates an empty list of bullets. Finally, it breaks out of the bullet loop.

   Technically you might want to do other things after a target is hit, like increment a score or reset the launch speed and launch angle back to their initial values for example. Or, you might blow up the target.

8. 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-distance traveled by the bullet (called the range) when the bullet hits the target or ground. Note that the range is not the same as the x-position of the bullet because the bullet isn't launched from the origin.
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 stops. Remember to use your `collisionSphereAndBox()` function to check for a collision between the bullet and barrier.
2. Create additional gameplay features like a scoring system and a max number of bullets.
3. 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.

22 PROGRAM – Modeling motion with friction

Apparatus

computer
GlowScript

Goal

In this activity, you will learn how to add a frictional force to a simulation.

Introduction

We are going to model the motion of a puck in air hockey that is slowed by a frictional force between the puck and the table. Though in practice, a puck in air hockey may be more influenced by air drag than friction with the table, we will treat the frictional force as *sliding friction*. Sliding friction occurs when one objects slides against the surface of another object. In the simplest situations, slide friction is:

$$\vec{f}_{slide} = \mu_k F_\perp(-\hat{v})$$

where μ_k is the coefficient of kinetic friction, $F_\perp$ is the perpendicular component of the contact force, and $\hat{v}$ is the unit vector pointing in the direction of the velocity of the object. In the case of a puck sliding on a level air hockey table, $F_\perp = mg$, the weight of the puck. Notice the negative sign for the direction of the frictional force. This means that sliding friction is always opposite the velocity of the object relative to the surface it is in contact with.

The coefficient of kinetic friction μ_k is a constant that depends on the materials in contact. For example, wood sliding on glass, wood sliding on concrete, and wood sliding on sand paper have different values of μ_k. A higher coefficient of friction means that there is a greater frictional force. Smaller coefficient of friction is less friction, and zero coefficient of friction is what we call "frictionless" (which doesn't exist in practice though friction might be negligible if it is small enough).

Rolling friction

A rolling ball also experiences a frictional force that decreases its center-of-mass velocity. If you push an object on wheels across carpet, you can see the carpet bunch up in front of the wheel. This is what happens at the microscopic scale for any surface. Rolling friction can be characterized as $\vec{f}_{roll} = \mu_{roll} mg(-\hat{v})$, where μ_{roll} depends on the properties of the surfaces in contact.

Procedure

In this program, we will model the motion of a golf ball rolling on a level green. This code template can be copied from our course web site on Trinket.

1. Begin by typing the following template for a golf ball rolling in the x-direction on a green.

```
1  GlowScript 2.1 VPython
2
3  scene.range=20
4  scene.width=400
5  scene.height=400
6
7  ground = box(pos=vector(0,0,0), size=vector(40,40,1), color=color.green)
8  ball = sphere(pos=vector(-18,0,0), radius=0.5, color=color.white)
9  ground.pos.z=ground.pos.z-ground.width/2-ball.radius
10 hole = cylinder(pos=vector(15,0,ground.pos.z+ground.width/2),axis=vector
       (0,0,1), radius=3*ball.radius, color=vector(0.8,0.8,0.8))
11 hole.pos.z=hole.pos.z-mag(hole.axis)*0.9
12
13 #ball, friction, and grav
14 ball.m=0.045
15 g=10
16 mu=0.1
17
18 #speed
19 initialspeed=5
20
21 #velocity vector
22 ball.v=initialspeed*vector(1,0,0)
23 scale=5/initialspeed
24 varrow = arrow(pos=ball.pos, axis=scale*ball.v, shaftwidth=0.5, color=
       color.yellow)
25
26
27 #clock
28 dt=0.01
29 t=0
30
31 scene.waitfor("click")
32
33 while 1:
34         rate(100)
35         vhat=ball.v/mag(ball.v)
36         Fnet=mu*ball.m*g*(-vhat)
37 #         ball.v=
38 #         ball.pos=
39
40         varrow.pos=ball.pos
41         varrow.axis=scale*ball.v
42
43         t=t+dt
```

2. Fill in lines 38 and 39 and run the program.

What does line 36 do?

If you wanted to add other forces to the simulation, which line would you change?

What does the arrow represent?

What variable represents the coefficient of friction?

3. Try different values of the initial speed until you can get the ball to stop in the hole.

4. Now change the coefficient of friction to either a smaller or larger value and find the new initial speed needed to get the ball into the cup.

Motion on a Hill

Suppose an object like a golf ball travels across a hill of constant slope. Let's define $+y$ in the plane of the hill and directly uphill. Define $+x$ to the right, in the plane of the hill, perpendicular to the $+y$ axis. Then, with this coordinate system, $+z$, is perpendicular to the hill toward the sky (but not directly outward from Earth due to the inclination of the hill). Suppose that the hill is inclined at an angle α relative to vertical (as established by hanging a weight on a string).

Sketch a top view and side view of the hill. Show the $+x$, $+y$, and $+z$ directions in each view.

The motion of the ball is similar to projectile motion, but its acceleration is modified by the angle of inclination of the hill. (And of course there is rolling friction.) In this case,

$$F_{grav,y} = -mg\sin(\alpha)$$

and

$$F_{grav,z} = -mg\cos(\alpha)$$

Since the ball has no acceleration in the z direction, the perpendicular component of the force by the ground on the ball is equal in magnitude to $F_{grav,z}$, but in the $+z$ direction. Thus,

$$F_{ground,z} = mg\cos(\alpha)$$

Then the rolling frictional force is:

$$\begin{aligned}\vec{f}_{roll} &= \mu_{roll}F_{\perp}(-\hat{v}) \\ &= \mu_{roll}mg\cos(\alpha)(-\hat{v})\end{aligned}$$

The net force on the ball is the sum of the gravitational force and force by the ground (both parallel and perpendicular to the ground). Thus,

$$\begin{aligned}\vec{F}_{net} &= \vec{F}_{grav} + \vec{F}_{roll} + \vec{F}_{ground,z} \\ &= (0, -mg\sin(\alpha), mg\cos(\alpha)) + \mu_{roll}mg\cos(\alpha)(-\hat{v}) + (0, 0, mg\cos(\alpha))\end{aligned}$$

For simplicity, I will combine $\vec{F}_{grav}$ and $\vec{F}_{ground,z}$ to give:

$$\vec{F}_{net} = (0, -mg\sin(\alpha), 0) + \mu_{roll}mg\cos(\alpha)(-\hat{v})$$

where g is the magnitude of the gravitational field of Earth at its surface. You can think of this conceptually as the component of the gravitational force parallel to the hill plus the frictional force:

$$\vec{F}_{net} = \vec{F}_{grav,\parallel} + \vec{F}_{roll}$$

If you strike the golf ball at an angle θ relative to the x-axis with a speed v_i, then its velocity in the plane of the hill is:

$$\vec{v} = v_i(\cos(\theta), \sin(\theta), 0)$$

Using the net force and initial velocity of the ball, you can model the motion of a golf ball that includes both "break" (due to the slope of the hill) and rolling friction.

5. Create a new program.

6. Copy and paste the following template from our course web site.

```
GlowScript 2.1 VPython

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

scene.range=20
scene.width=400
scene.height=400

ground = box(pos=vector(0,0,0), size=vector(40,40,1), color=color.green)
ball = sphere(pos=vector(-18,0,0), radius=0.5, color=color.white,
    make_trail=True)
ground.pos.z=ground.pos.z-ground.width/2-ball.radius
hole = cylinder(pos=vector(15,0,ground.pos.z+ground.width/2),axis=vector
    (0,0,1), radius=3*ball.radius, color=vector(0.8,0.8,0.8))
hole.pos.z=hole.pos.z-mag(hole.axis)*0.9

#ball, friction, and grav
ball.m=0.045
g=10
mu=0.2
alpha=rad(5)

#speed and angle
initialspeed=12
theta=15

#velocity vector
#ball.v=
scale=5/initialspeed
varrow = arrow(pos=ball.pos, axis=scale*ball.v, shaftwidth=0.5, color=
    color.yellow)


#clock
dt=0.01
t=0

scene.waitfor("click")

while 1:
        rate(100)
        vhat=ball.v/mag(ball.v)
#         Fgrav=
#         Ffriction=
        Fnet=Fgrav+Ffriction
        ball.v=ball.v+Fnet/ball.m*dt
        ball.pos=ball.pos+ball.v*dt

        varrow.pos=ball.pos
        varrow.axis=scale*ball.v

```

```
51              t=t+dt
```

7. Fill in lines 28, 42, and 43 and remove the comment marks. Run the program.

What variable will you change in order to make the hill steeper or less steep?

What variable will you change to adjust the coefficient of friction?

What variable will you change to adjust the initial speed of the ball?

What variable will you change in order to strike the golf ball at a greater or less angle (relative to the $+x$ axis).

Lab Report

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

1. Go back to your first program of a golf ball rolling on a level green.
2. Place the cup the location $x = 15$ m, $y = 15$ m, near the top right corner of the green.
3. Find the initial velocity vector needed to get the ball into the cup, if $mu = 0.1$.

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

1. Go back to your program of a golf ball rolling on a hill.
2. Add keyboard interaction so that by using the up and down arrows, you can change the angle at which you strike the ball, and by using the left and right arrows, you can change the initial speed of the ball. Then, use the spacebar to putt the ball. (Use your tank wars program for a reminder on how to write the code.) Use the initial position of the hole at $x = 15$ m, along the x-axis in order to test out your code.
3. Alert the player and stop the ball if the ball goes into the hole.
4. Alert the player and stop the ball if the ball goes off screen.

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

1. Use your program from Part B.
2. Change properties of the green (coefficient of friction or slope of the hill or location of the hole) each time the player sinks a putt and allow the player to play again. Reset the ball to its starting location when the ball misses the hole and stops or if it goes off the screen.

23 LAB – Center of Mass Velocity During an Elastic Collision

Apparatus

Tracker video analysis software
computer

Goal

he purpose of this experiment is to measure the velocity of the center of mass of two pucks that make a collision on an air hockey table. You will measure the center-of-mass velocity before the collision and after the collision, and you will compare the results.

Introduction

The location of the center of mass of a system of two particles is

$$\vec{r}_{cm} = \frac{m_1\vec{r}_1 + m_2\vec{r}_2}{m_1 + m_2}$$

This is a vector equation that must be true for both the x and y directions (for two dimensions).

$$x_{cm} = \frac{m_1x_1 + m_2x_2}{m_1 + m_2}$$

$$y_{cm} = \frac{m_1y_1 + m_2y_2}{m_1 + m_2}$$

Likewise, the center-of-mass velocity is

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

Again, this equation must hold true for both the x and y components of the center-of-mass velocity.

$$v_{cm,x} = \frac{m_1v_{1x} + m_2v_{2x}}{m_1 + m_2}$$

$$v_{cm,y} = \frac{m_1 v_{1y} + m_2 v_{2y}}{m_1 + m_2}$$

Procedure

It is expected that you have completed the other video analysis experiments, so these instructions do not include details about how to use the *Tracker* software.

1. Download the video *collision-pucks.mov* from our course web site.
2. Open *Tracker* and insert the video.
3. Play the video and view the motion of the colliding pucks.
4. Record the mass of each puck that is printed on the video's first frame.

$m_{blue} =$

$m_{red} =$

5. Set the origin of your coordinate system. Any location is fine. I happened to choose the center of the blue puck in the first frame.
6. Set the calibration using the meterstick on the left side of the video. You will have greater accuracy if you use 5 of the 10-cm segments for your calibration. In other words, stretch the calibration stick across 5 segments for a total length of 0.5 m.
7. Mark the blue puck for each frame of the video.
8. Mark the red puck for each frame of the video. (You will have to create another point mass first.)
9. Using the graphs of x vs. t and y vs. t for each puck, measure the following quantities:

Table 23.1: default

	v_{xi}	v_{yi}	v_{xf}	v_{yf}
blue				
red				

Analysis

Calculate the x-component of the center-of-mass velocity *before* the collision, $v_{cm,ix}$.

Calculate the y-component of the center-of-mass velocity *before* the collision, $v_{cm,iy}$.

Calculate the x-component of the center-of-mass velocity *after* the collision, $v_{cm,fx}$.

Calculate the y-component of the center-of-mass velocity *after* the collision, $v_{cm,fy}$.

What is $\vec{v}_{cm,i}$? Write and sketch the vector.

What is $\vec{v}_{cm,f}$? Write and sketch the vector.

Tracker can calculate and track the center of mass for you. The following steps will help you learn how to automatically calculate and tracker the center of mass.

1. We need to define the masses of the pucks. Click the tab for **mass A** in the Track Control toolbar. In the drop-down menu, select **Define...** . In the resulting pop-up window, enter the mass of the puck for the parameter **m** as shown in Figure 23.1.

2. Repeat the previous step for **mass B** and enter its mass.

3. Click the Create button and select **Center of Mass** , as shown in Figure 23.2.

4. You will see a new tab in the Track Control toolbar named **cm**. Click cm to get the menu for the cm object shown in Figure 23.3. Click the **Select Masses...** menu item.

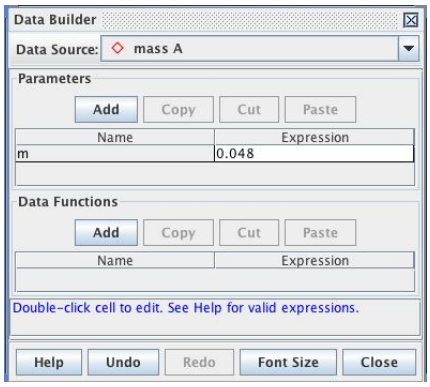


Figure 23.1: Enter of the mass of the puck.

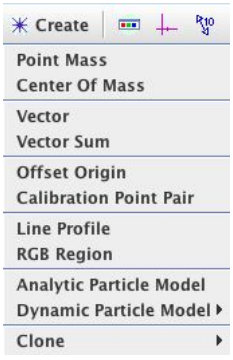


Figure 23.2: Select **Center of Mass** from the menu.

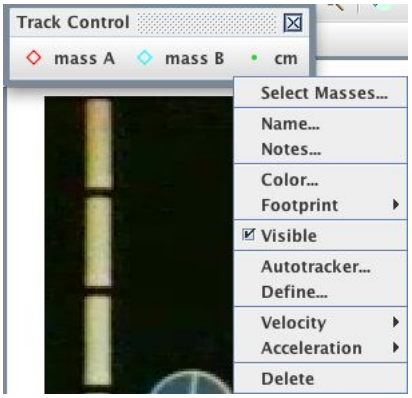


Figure 23.3: Click on **Select Masses...** from the menu.

5. An additional window will pop up. Select both masses "mass A" and "mass B" in this window and click OK as shown in Figure 23.4.

6. You will now see a track for the center of mass and you will see a graph of x vs. t.

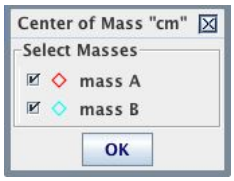


Figure 23.4: Check both masses (i.e. pucks) in this window.

By measuring the slope of x vs. t for the center of mass before the collision, what is $v_{cm,ix}$?

By measuring the slope of y vs. t for the center of mass before the collision, what is $v_{cm,iy}$?

By measuring the slope of x vs. t for the center of mass after the collision, what is $v_{cm,fx}$?

By measuring the slope of y vs. t for the center of mass after the collision, what is $v_{cm,fy}$?

How did the measurements for the center-of-mass velocity compare to what you calculated in the first part of the experiments?

Did the collision significantly affect the center-of-mass velocity?

Lab Report

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

1. Using data from Table 1, what is the center-of-mass velocity vector before the collision?
2. Using data from Table 1, what is the center-of-mass velocity vector after the collision?
3. What is the center-of-mass velocity before the collision, as measured by the slope of the x and y vs. time graphs of the center of mass? Report your answer as a vector.
4. What is the center-of-mass velocity after the collision, as measured by the slope of the x and y vs. time graphs of the center of mass? Report your answer as a vector.

B Do all parts for **C** and answer the following questions.

1. Can you conclude that the center-of-mass velocity is constant (or nearly constant) during the collision?
2. You used Tracker to calculate the location of the center of mass and mark it on the video. What does the fact that it is a straight line (both before and after the collision) and the fact that the spacing between marks is uniform (i.e. the same) tell you about the center of mass velocity?
3. What is the acceleration of the center of mass?
4. What is the net force on the center of mass? (Hint: use Newton's second law in your reasoning.)

A Do all parts for **B** and answer the following questions.

1. The collision occurs in approximately 1/30 of a second, so $\Delta t = 1/30$ s. Use Newton's second law and the data for v_{xi} , v_{yi} , v_{xf} , and v_{yf} for the blue puck to calculate the force on the blue puck during the collision. Express your answer as a vector.
2. Use data for the red puck to calculate the force on the red puck during the collision. Express your answer as a vector.
3. Compare the force on the blue puck and the force on the red puck during the collision. What do you notice? (This is a VERY big observation. It is so important that it is called *Newton's Third Law* and governs objects that interact via electric or gravitational forces).

24 GAME – Asteroids

Apparatus

Computer
Glowscript – www.glowscript.org

Goal

The purpose of this activity is to modify an asteroids game so that when a large asteroid explodes 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 arrow keys and how they affect the motion of the spaceship.
2. Copy the template from our course web site, and paste it into a new file in GlowScript.
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 original Asteroids game. Also, note that hitting an asteroid with a bullet does not make it break up into pieces. (You will add this functionality.)
4. Answer the following questions.

   (a) What are the three functions used for generating random numbers and what is the purpose of each of these functions?

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

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

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

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

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

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

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

(i) 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.)

(j) 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 Asteroid Explsions

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 velocity of the system remains constant.

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

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

where 1 and 2 refer to the two fragments. 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, find the line where the asteroidList is created. Near this point in the program, create a list called `fragmentList` . To do this, type the following line:

```
fragmentList=[]
```

6. Near the top of the program where other functions are defined (e.g. after the `createAsteroids()` function is defined), write a function that creates 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=choice(1,-1)*randint(1,5)
    fragment1.v.y=choice(1,-1)*randint(1,5)
#     fragment2.v=
    fragmentList.append(fragment1)
    fragmentList.append(fragment2)
```

7. In the main `while` loop, there is an `if` statement that checks for a collision between a bullet and asteroid. The line reads:

```
if(collisionSpheres(thisbullet,thisasteroid)and thisbullet.visible==True):
```

Inside this `if` statement, call the `createFragments` function using:

```
createFragments(thisasteroid)
```

This creates the explosion whenever the bullet hits an asteroid.

8. You will need to update the positions of the fragments (i.e. make them move) inside the main `while` loop. 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. You can also copy and paste this code from our course web site. Here's an example of what it should look like:

```
    #update positions of fragments
    for thisfragment in fragmentList:
        if thisfragment.visible==True:
            thisfragment.pos=thisfragment.pos+thisfragment.v*dt
            #check for collision with spaceship
            if(collisionConeSphere(spaceship,thisfragment)):
                spaceship.visible=False
                fire.visible=False
            #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==True):
                    thisfragment.visible=False
                    thisbullet.visible=False
```

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. 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. Add a keystroke for the "down" arrow key that rotates the spaceship 180° (π radians). For the syntax on how to do this, see the left and right arrow keys.
2. Place the entire `while spaceship.visible==True:` loop inside another `while True:` loop so that after the spaceship collides with an asteroid the program will not end, but rather it will begin again. If you select the entire `while spaceship.visible==True:` loop and hit the TAB key, then the selected code will automatically indent.
3. 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 inside the `while True:` statement but at the end. In other words, when the inner loops ends (by virtue of the spaceship not being visible), then this code block below will run before the outer loops starts again.

```
scene.waitfor("click")

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

#bullets
for b in bulletsList:
    b.visible=False
bulletsList=[]

#asteroids
Nleft=0 #counter for number of asteroids left in the scene
for ast in asteroidList:
    ast.visible=False
asteroidList=[]

createAsteroids()

#fragments
for f in fragmentList:
    f.visible=False
fragmentList=[]
```

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

1. Think of an improvement that will require knowledge of physics. Here are some ideas:
   (a) asteroids collide with each other
   (b) bullets have mass. Every time a bullet is fired, it exerts a backwards force on the spaceship and causes the spaceship to lose mass.
   (c) the asteroid fragments should have have the volume of the asteroids and thus should have the appropriate radius.

(d) the asteroid can break into three fragments instead of two.

(e) the fragments can have different masses thought the sum is equal to the mass of the asteroid

2. Talk to Dr. T the improvement you want to make.
3. Implement your idea.

25 LAB: Arduino Gamecontroller

Apparatus

Computer
Anaconda
Jupyter Notebook
VPython (for Jupyter)
Arduino IDE programming editor and compiler
Arduino Uno microprocessor
breadboard
2-axis potentiometer (joystick)
wiring kit
LED
pushbutton SPST switch

Goal

The purpose of this activity is to build a game controller and to use the game controller to operate the thruster in the Lunar Lander game. You will install a number of software packages to make this possible. The project requires: (1) biulding

Procedure

There are four steps:

1. Install software
2. Build the game controller
3. Upload an Arduino program to the Arduino Uno
4. Run Lunar Lander with the Arduino game controller

Install Software

1. Go to our course web site where you will find links to download and install software.
2. Install the Arduino IDE for writing and compiling Arduino programs and uploading to an Arduino board.
3. Install the Anaconda distribution of the Python 2.7 programming language and scientific packages. **IMPORTANT – DOWNLOAD THE INSTALLER FOR PYTHON 2.7.** (not Python 3.5)
4. After you install Anaconda, open a terminal window (also called the command line). On a Mac, you will do this by opening Applications→Utilities→Terminal. It should look similar to the following terminal window.
5. To verify that Anaconda is installed property, at the command line, type the following:

```
which conda
```

Figure 25.1: Terminal window.

The command `which` returns the filepath to the location of the conda program.

6. Now, type `which jupyter` at the command line. It should return the path to the jupyter program as shown in Figure 25.2.

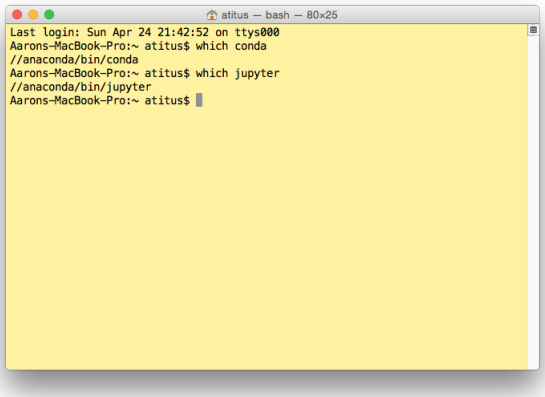


Figure 25.2: Terminal window showing path to conda and path to jupyter.

If this command does not return a path to jupyter, then you need to install Jupyter. In this case, type the command:

```
conda install jupyter
```

7. Install the `vpython` package by typing:

```
pip install vpython
```

8. Install the `pyserial` package by typing:

```
pip install pyserial
```

To test your software installation, we will open a Jupyter Notebook, import vpython and pyserial packages, and create a 3D object. If it is successful and produces no error messages, then we are confident that we will be able to develop programs that use our game controller.

9. At the command line, type

   ```
   jupyter notebook
   ```

   A Jupyter window will open in your web browser showing your files and folders in your home directory (or whatever directory you are in when you launch jupyter notebook), as shown in Figure 25.3.

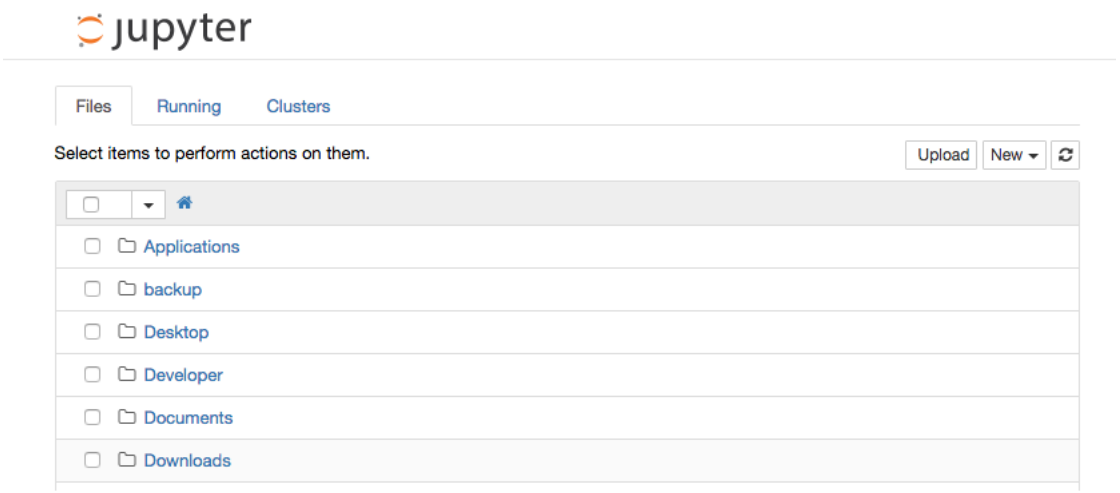


Figure 25.3: A Jupyter window.

10. You can clink the folder links to navigate to the folder of your choice. Then, create a new notebook by clicking the New button in the Jupyter toolbar. In the menu, select the `VPython` notebook, as shown in Figure 25.4.

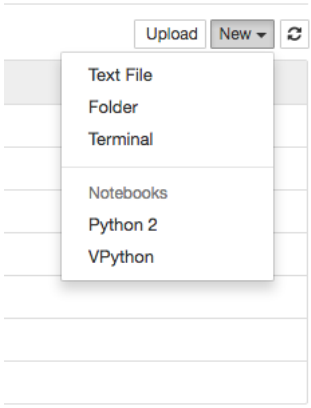


Figure 25.4: Creating a new notebook

This creates a new notebook file as shown in Figure 25.5.

11. Click the name "Untitled" and change the name to something more appropriate like "notebook-test" or something like that. I suggest that you do not use blanks or characters other than a hypen or underscore in filenames.

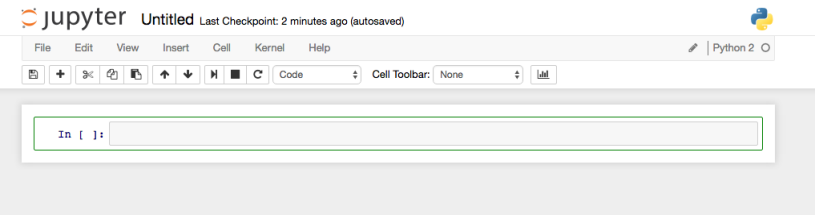


Figure 25.5: A newly created Jupyter notebook

12. In the first cell, type

```
from vpython import *
```

Use shift-RETURN to run the code in the cell.

After running, the cell will receive the number 1 and will be labeled `In [1]:` . The numbering system shows you the sequence that cells were run.

13. In the second cell, type

```
from serial import *
```

Again, use shift-RETURN to run the cell. (Do this for every cell in order to run it.) At this point, there should be no error messages.

14. Now, we have to create a canvas (i.e. scene) and a 3D object. In the third cell, type and run the following code:

```
scene=canvas(title="3D scene")
sphere()
```

You should see a sphere like the example in Figure 25.6.

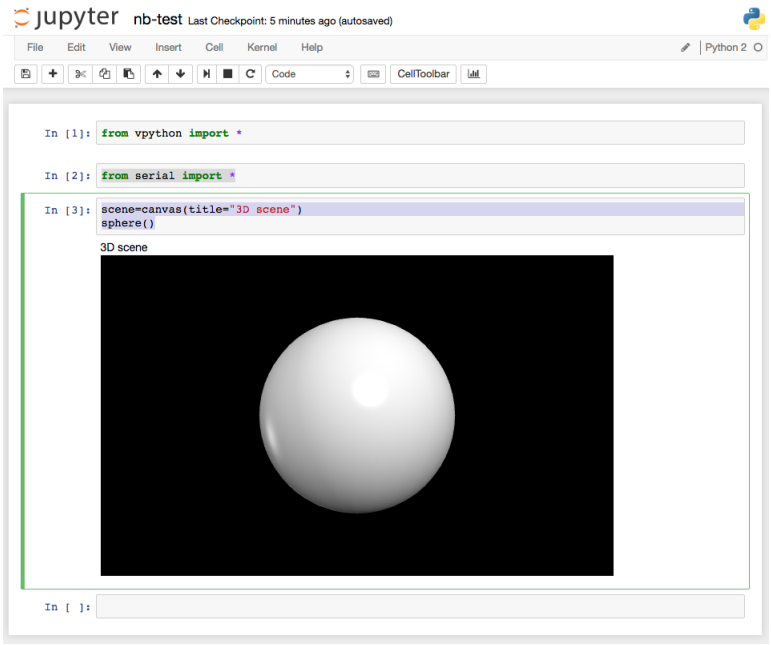


Figure 25.6: A VPython program running in a Jupyter notebook.

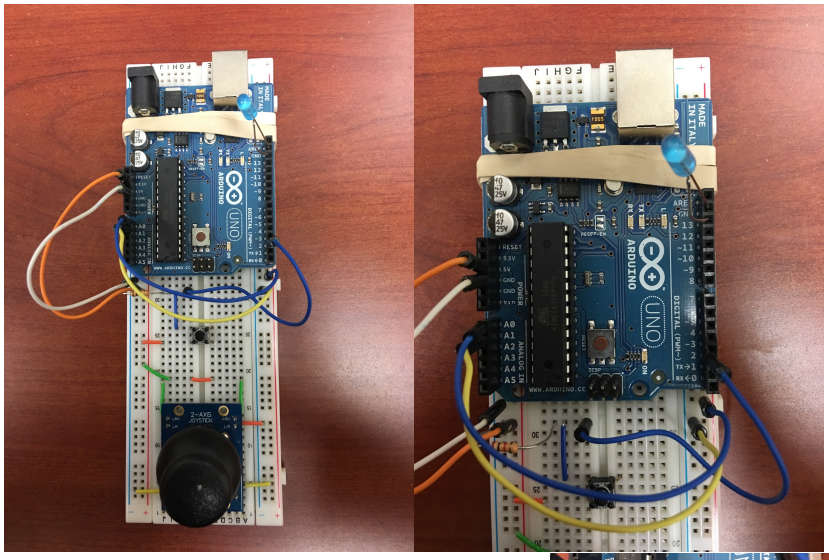
If you receive any errors up to this point, please get help. This program must work before you continue.

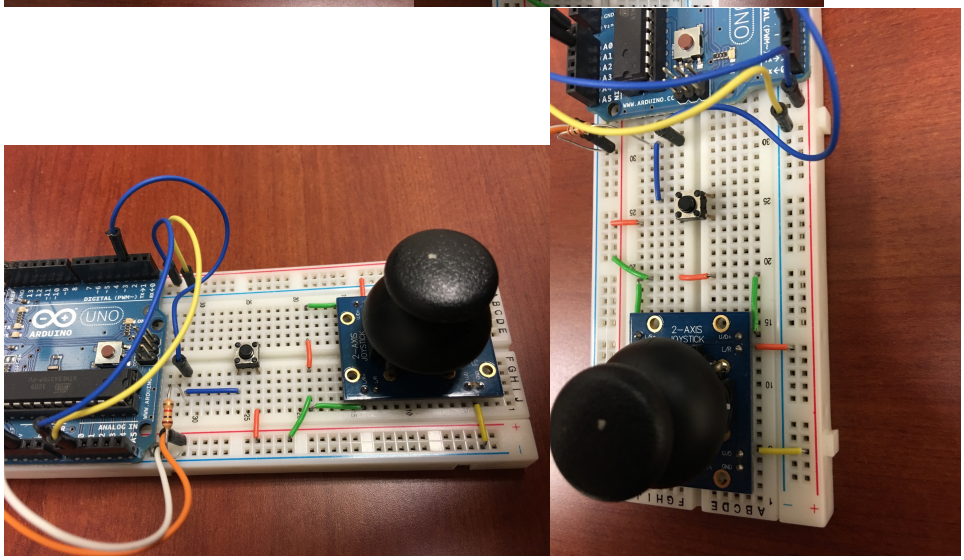
Build the game controller

Here are the parts:

(a) an Arduino Uno microprocessor
(b) a breadboard
(c) a LED
(d) a push button SPST switch
(e) a 1000 Ω resistor
(f) a Parallax 2-axis joystick
(g) a wiring kit
(h) a USB cable
(i) a rubber band

Here are photos that show the wiring for the game controller.





15. Attach the Arduino to the breadboard with the rubber band.

16. Place the LED in pins 13 and GND. The LED has a long lead and a short lead. The long lead goes in pin 13 and the short lead goes in GND.

17. Run wires from GND (white wire) and +5 V (orange wire) pins on the left side of the Arduino to the – and + columns on the left side of the breadboard, respectively. GND is wired to the – column and 5 V is wired to the + column.

18. Run wires from pins A0 and A1 on the left side of the Arduino to the + and – columns on the right side of the breadboard, respectively. A0 is wired to the +column an dA1 is wired to the – column.

19. Attach the joystick to the breadboard.

20. Wire the L/R+ and U/D+ pins on the joystick to +5 V (the + column on the left side of the breadboard).

21. Wire the GND pin on the joystick to GND (the – column on the left side of the breadboard).

22. Wire the U/D pin on the joystick to pin A0 of the Arduino (the + column on the right side of the breadboard).

23. Wire the L/R pin on the joystick to pin A1 of the Arduino (the – column on the right side of the breadboard).

24. The pushbutton switch, resistor, and the wire from pin 3 of the Arduino are not needed for the Lunar Lander game. You can add those parts later if you wish to fire bullets in Asteroids, for example.

Upload an Arduino program to the Arduino Uno

We must upload a program to the Arduino that reads the voltage across each axis of the joystick and passed it to VPython when requested.

25. Download lunarLander.zip from our course web site. Here is the code.

```
#define UD A0
#define LR A1
int received;
char buffer[10];                    // input buffer
int N;                              // how many measurements to make
boolean done = false;


void setup() {
  Serial.begin(9600);
}

void loop() {
        received = 0;
        buffer[received] = '\0';
        done = false;

        // Check input on serial line.
        while (!done) {
                if (Serial.available()) {       // Something is in the
                    buffer
                        N = Serial.read();      // so get the number byte
                        done = true;
                }
        }
```

```
25
26          int LRval = analogRead(LR);
27          int UDval = analogRead(UD);
28          Serial.print(LRval, DEC);
29          Serial.print('\t');
30          Serial.print(UDval, DEC);
31          Serial.print('\n');
32          delay(10);
33      }
```

26. Open `lunarLander.ino` with the Arduino software. The program looks like Figure 25.7.

Figure 25.7: An Arduino program running on the microprocessor.

27. Go to the **Tools** menu and select **Port** . One of these ports corresponds to the Arduino. Make sure the correct one is checked, as shown in Figure 25.8. This usually occurs by default.

    There are two important buttons in the top left corner of the menu bar. One is a checkmark and one is a right arrow, as shown in Figure 25.9.

28. The checkmark button is used to compile the Arduino program. It will tell you if there are any programming errors. Click the checkmark button to compile.

29. If there are no errors, then you are ready to upload the program to the microprocessor. Click the right arrow button to upload the program to the mircoprocessor.

    The program should now be running on the microprocessor. It runs continuously in an infinite loop as long as it has power.

Running Lunar Lander with the Arduino game controller

30. I have already created a Jupyter notebook that you can use as a template. Download the file `lunar-lander-nb.ipynb` from our course web site. Save the file (or move the file) into the folder where you stored your first Jupyter notebook file that you used to test the software.

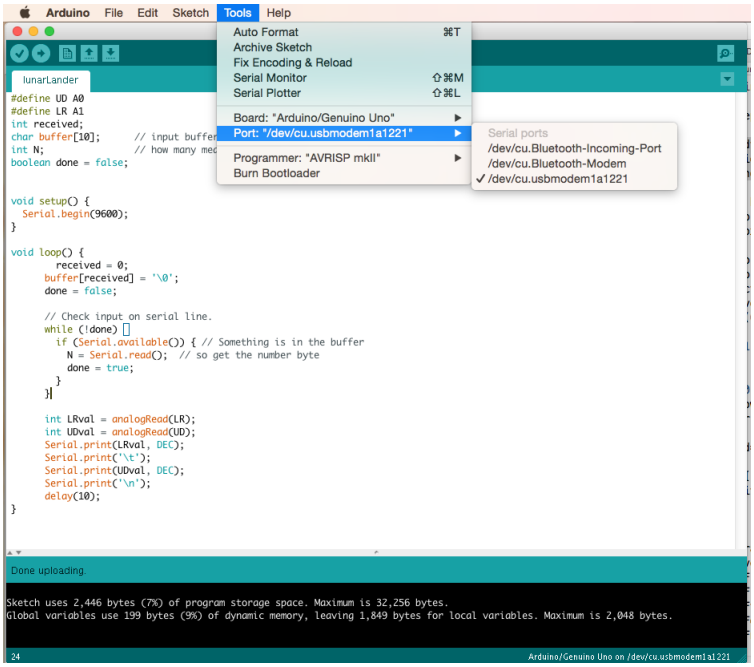


Figure 25.8: Select the serial port for your Arduiono

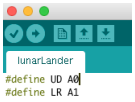


Figure 25.9: Select the serial port for your Arduiono

31. Go to the Jupyter browser window that shows your files and folders. Click on the file you just downloaded. It should open as a VPython notebook (as indicated in the top, right corner).

32. Run the first cell. (It imports packages.)

33. The second cell has the code necessary to communicate with the Arduino. You need to get the serial port for your Arduino. Go to the Arduino software, click the **Tools** menu and select **Port** . Right down the name of the serial port that your Arduino is connected to. In your VPython notebook, edit the `port` variable to match the name of the port used by your Arduino. Mine is:

```
#serial port for the Arduino; get the name for the port from the Arduino
    software
port = '/dev/cu.usbmodem1a1221'
```

    Make sure the name of the port is contained within quotes since it is a string.

34. Run this cell and verify that there are no errors.

35. Run the third cell that contains the code for the game. You should see the lander, and you should be able to control it with your game controller.

Analysis

B Complete this exercise and do the following.

1. Upload a working notebook file produced at the end of this activity.

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

1. Write a new game that uses the game controller and runs in a notebook.

Appendix 1: Tracker Cheat Sheet

Description

This one-page "cheat sheet" will show you the most common steps required to analyze videos with Tracker.Tracker is developed by Doug Brown and is available from `http://www.cabrillo.edu/~dbrown/tracker/`.

Preliminary Steps

1. Go to **Video→Import...** and select the video you wish to analyze.

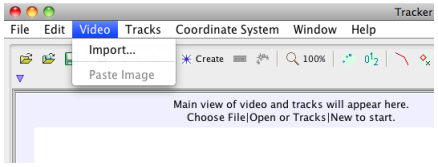


Figure .1: Import your video.

2. Click the movie settings icon  and set the frame rate, start frame, and end frame. Note that setting the start frame and end frame is quite useful for trimming the movie to just the part that is of interest.

3. Click the calibration icon  and select one of the calibration tools. Stretch the tool across an object of known length, such as a meterstick, that is in the plane of the motion. Click on the numeric indicator for the length of the tool and change the value to be the length of your standard object in the video.

4. Click the coordinate system icon  and move the origin of the coordinate system to the desired location. Grab the x-axis and rotate it in order to set the direction of the coordinate system.

Mark an object

1. Click the Create button and select **Point Mass** .

2. In the toolbar, a menu for **mass A** will appear. 

Click on **mass A** in order to select its **Name...** , **Color...** , or other parameters.

3. ***While holding the shift key***, click on the object. A mark will appear, and the video will advance to the next frame.

Analyze a graph

1. Right-click on the graph and select **Analyze...** . The Data Tool window will pop up.

2. Check ☑ Fit, and the Fit menus and parameters will appear.

3. Check ☑ Autofit for the Data Tool to automatically calculate the best-fit parameters.

4. To fit a curve to a portion of the data, select the data in the graph or data table.

Appendix 2: Project 1 – Developing your first game with constant velocity motion

Goal

For this project, you will develop a game using objects that move with constant velocity.

Project Guidelines

You will develop a game that involves objects that move with constant velocity. It is due at midnight Mar 24. Upload your game and documentation via WebAssign. You may request help from Michael Welter and I. We can answer questions, give you advice, and help you troubleshoot syntax or logic errors. However we will be cautious to not write significant portions of your code.

There are four categories that the project's grade is based on.

1. level of difficulty (i.e. "Is the game more interesting than just a ball bouncing back and forth on the screen?" and "Is it sufficiently different from the one developed in class?" and "Is the game fun to play?"). Note that you should be creative, but don't try writing a game that exceeds your abilities at this point. Choose a project that is (1) doable and (2) interesting.
2. level of independence. Using resources is good, but you can't copy another program or have someone else write the code for you. Always cite your references, including people who help you.
3. completeness (i.e. "Does the simulation run?" and "Did the program include objects that move with constant velocity?" and "Does the program work as expected?")
4. quality of documentation (i.e. "Did you include relevant references?" and "Did others test your game?" and "Did you write in a clear, coherent, organized, and grammatically correct way?")

You should:

1. write a VPython program that includes objects that move with constant velocity.
2. test your game by having others play it and by asking them for feedback.
3. answer the questions below.

Documentation

You must write a document that answers the following questions. The document should be complete; it should have correct grammar; and it should be easy to read and understand. Save the document as a pdf file and upload it through WebAssign. Quality writing and organization is expected. Answer the following questions.

1. What is the purpose of your game?
2. What are the rules of your game?

3. How must the game be played (i.e. keystrokes, etc.)?
4. Is this game like any other game that you've seen or played?
5. Who played your game and what did you learn as a result of their feedback?
6. What resources did you use to help you in writing the game? If you used web sites, people (such as Dr. T or the S.I.), books, or any other resources, you must reference them.
7. What did you personally get out of this project?

Appendix 3: Final Project – Developing an Original Game

Goal

For this project, you will develop an original game that incorporates moving objects that obey the laws of physics. This includes objects that move with constant velocity and/or constant acceleration, objects with correct relative motion (such as bullets shot from a moving shooter), projectiles, and objects that explode into pieces (like in the game Asteroids).

Project Guidelines

You will develop a game that involves moving objects that obey the laws of physics. It is due on at the beginning of our Final Exam time. Your game must include accelerated motion due to forces on an object. This may include collisions, explosions, projectiles, thrust, drag, friction, etc.

There are six categories that the project's grade is based on.

1. level of difficulty ("How does your game compare to the ones written for this class in terms of difficulty?" or "Is the code written for your project similar to A-level exercises in our class in terms of difficulty?" or "Does your game include competitive aspects?")
2. level of creativity ("Is this game exactly like ones we've done in class or have you reached to do something innovative?" or "Did you try to duplicate other popular games?" or "Did you add correct physics to a popular game that did not have correct physics?")
3. level of independence. Using resources is good, but you can't copy another program or having someone else write the code for you. Always cite your references, including people who help you. Be sure to cite the page numbers from our class handouts that were used for your project.
4. completeness (i.e. "Does the simulation run?" or or "Did the program include objects that move with constant velocity?" or "Does the program work as expected?")
5. quality of documentation ("Did you include relevant references?" or "Did others test your game?")
6. quality of your presentation ("Did you discuss the purpose of the game?" or "Did you discuss the rules of the game?" or "Did you discuss the physics principles used in the game?")

You should:

1. write a VPython program that includes objects that move. The program must contain accelerated motion due to collisions, explosions, projectile, thrust, drag, friction, etc. This is not an exhaustive list. It is merely an example of the physics that meets this criterion.
2. test your game by having others play it and by asking them for feedback.
3. answer the questions below in a typeset document.
4. create and deliver a presentation about the game.

Documentation

You must write a document in Microsoft Word or pdf format that answers the following questions. The document should be complete; it should have correct grammar; and it should be easy to read and understand. Quality writing and organization is expected. Detail is required. Terse responses will not receive significant credit.

1. What is the purpose of your game?
2. What are the rules of your game?
3. How must the game be played (i.e. keystrokes, etc.)?
4. Is this game like any other game that you've seen or played?
5. What physics principles were used in your game in order to make it realistic? Be sure to cite the physics principle(s) (such as projectile motion, constant velocity motion, constant acceleration motion, relative motion, center of mass motion, etc.) and where these principles were discussed in our course handouts. Be sure to reference the page number(s).
6. Does your game violate any laws of physics? (Note that sometimes this is desirable for playability or artistry.)
7. Who played your game and what did you learn as a result of their feedback?
8. How might you improve your game in the future?
9. What resources did you use to help you in writing the game? If you used web sites, people (such as Dr. T), books, or any other resources, you must reference them.
10. What did you personally get out of this project?

Presentation

You will give a brief presentation less than three minutes. You are also expected to demonstrate your game by playing the game. You are expected to describe the physics in your game. Your presentation will be graded on:

1. whether others acted interested in your game by giving feedback or asking questions.
2. whether you spoke clearly and were organized in your presentation.
3. whether you addressed the physics in your game.
4. whether you demonstrated relative aspects of your game.
5. whether you were enthusiastic about your game.