CHAPTER 4

■ ■ ■

Chapter 4: Vectors

After completing this chapter, you will be able to:

- Understand that a vector relates two positions to each other,
- Recognize that all points in space are position vectors,
- Comprehend that a vector encapsulates both a distance and a direction,
- Perform basic vector algebra to scale, normalize, add, and subtract vectors,
- Apply vectors to control the motions of game objects,
- Implement simple game object behaviors like aiming and following,
- Design and simulate simple external factors like wind conditions to affect object motion.

Introduction

So far, you have reviewed some of the most elementary and ground laying mathematical concepts used in video game creation. These simple concepts that you have observed and interacted with can be developed further into a powerful and widely used tool set. This approach of introducing a simple concept and expanding it to solve real problems when designing a video game will be continued in this chapter with vectors and the fundamental algebra that accompanies them.

Vectors are entities that encapsulate point-to-point distance and direction. Vector algebra is the mechanism, or rules, for manipulating these two entities. It allows the user to, for example, increase the distance, change the direction, and, combine, or detract both the distance and direction at the same time. Vectors and their associated math concepts allow precise control and accurate prediction of basic game object movements as well as the support for many simple behaviors.

In many video games, object behaviors are often governed by their physical proximity to other objects. For example, non-player characters changing from their predefined wandering pattern, e.g., patrol path, and moving towards the approaching player. To support this simple scenario, you must be able to program the behavior of following a predefined route as well as the ability to detect and move towards the approaching player or character. Vectors, with their encapsulation of both distances and directions, are perfect for representing the motion of objects. Vector algebra complements this encapsulation with the ability to determine the relationships between the in-motion objects. Therefore, with just vectors and their accompanying mathematical operations, you as a game developer, at every moment in your game can determine exactly what game behavior to invoke. Vectors and their associated algebra are one of the most fundamental tools in developing video games.

This chapter introduces vectors as a tool for controlling motion and computing spatial relationships between objects. In general, vectors are important for many, just as significant, applications that are unrelated to object motions. This is especially true for applications of vectors to fields outside of interactive graphic applications or video games. For example, applying vectors in Machine Learning for data cluster analysis. Even within the field of video games, vectors are important for other applications. Some

of these other applications include predicting the exact intersection position between a motion path and a wall, and computing the reflection direction after a collision, both of which will be discussed in future chapters.

This chapter begins by reviewing what you have learned from Chapter 3, but now with a focus on how vectors were used to perform the distance calculations you have experimented with and observed. The chapter then analyzes the details of the vector definition and the algebraic rules that govern the operations on vectors. Through these discussions you will learn that the vector definition is independent of positions, that vectors can be scaled, normalized, and applied to represent velocities that define the motions of objects. The formal definition of vector algebra, the addition and subtraction operations, are presented towards the end of the chapter to conclude and verify the knowledge gained throughout the chapter.

Vectors: Relating Two Points

Vectors have been hinted at thus far in the book and even worked with in the previous chapter when you needed to compute the distance between positions, but now you will finally learn what they are and some of their applications. Please refer to Figure 4-1, which is identical to Figure 3-2 and copied here for convenience.

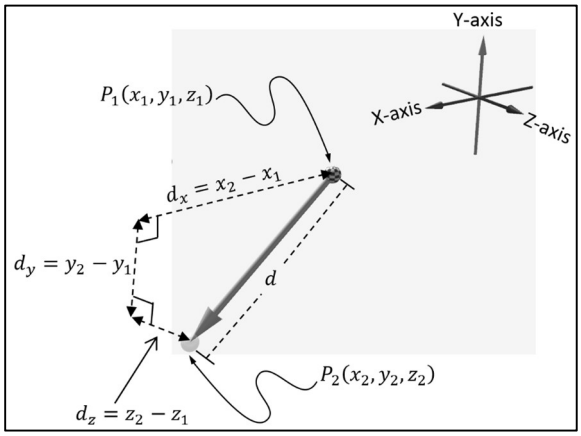

Figure 4-1. *Calculating the distance between any two positions: P_1 and P_2 (same as Figure 3-2)*

Recall that in order to compute the distance between two positions, P_1 and P_2, the distances measured along the major axes must be computed.

- Distance along X-Axis: $d_x = x_2 - x_1$
- Distance along Y-axis: $d_y = y_2 - y_1$
- Distance along Z-axis: $d_z = z_2 - z_1$

You learned that the distance, d, between these positions can be derived by applying the Pythagorean Theorem twice to the two connecting right-angle triangles (see Figure 3-1 if you need a refresher). The derived formula is simply the square root of the summed squared distances measured along the major axes, which is listed as follows.

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

$$d = \sqrt{{d_x}^2 + {d_y}^2 + {d_z}^2}$$

This formula can be interpreted as the distance that is necessary to move an object from position P_1 to P_2. This displacement is defined by the shortest traveling distant, d, along the direction encoded by (d_x, d_y, d_z). This interpretation is reflected closely in the implementation of the $\mathrm{Update}()$ function in $\mathrm{EX\_3\_1\_MyScript}$, as copied and re-listed as follows for reference.

```
void Update() {
    // Update the sphere positions
    Checker.transform.localPosition = CheckerPosition;
    Stripe.transform.localPosition = StripePosition;

    // Apply Pythagorean Theorem to compute distance
    float dx = StripePosition.x - CheckerPosition.x;
    float dy = StripePosition.y - CheckerPosition.y;
    float dz = StripePosition.z - CheckerPosition.z;
    DistanceBetween = Mathf.Sqrt(dx*dx + dy*dy + dz*dz);

    // Compute the magnitude of a Vector3
    Vector3 diff = StripePosition - CheckerPosition;
    MagnitudeOfVector = diff.magnitude;

    #region Display the dx, dy, and dz
}
```

Pay attention to the last two lines of code once more, specifically, the diff variable which is the result of subtracting $\mathrm{CheckerPosition}$ (P_1) from $\mathrm{StripePosition}$ (P_2). As you learned from this example in the last chapter, the magnitude operator returns the distance, d, between the two positions. This same diff variable also defines the direction from P_1 to P_2. This entity, diff, that encodes those two pieces of information, distance and direction, is a **vector**. The line of code that computes diff can be expressed mathematically as follows.

$$\begin{aligned}\vec{V}_d &= P_2 - P_1 \\ &= (x_2 - x_1, y_2 - y_1, z_2 - z_1) \\ &= (d_x, d_y, d_z)\end{aligned}$$

Or simply, vector $\vec{V}_d = (d_x, d_y, d_z)$. There are a few interesting observations that can be made thus far.

- **Symbol**: The symbol for a vector, V, is shown as $\vec{V}$, with an arrow above the character V representing that it's a vector.
- **Definition**: A vector, $\vec{V} = P_2 - P_1$, describes the distance and direction to travel from P_1 to P_2.
- **Notation**: In 3D space, a vector is represented by a tuple of three floating point values, signifying the displacements along each of the corresponding major axes. This notation is identical to that of a position in the Cartesian Coordinate System. In fact, given a tuple with three values, (x, y, z), without any context, it is impossible to differentiate between a position and a vector. This issue will be examined in the next section of this chapter.
- **Representation**: As illustrated in Figure 4-2, graphically, a vector $\vec{V} = (d_x, d_y, d_z)$, is drawn as a line that begins from a position, the **tail**, with an arrow pointing at the end position, the **head**, with the displacements of d_x, d_y, and d_z along the major axes. Note that in this case, d_y is a negative number because the y-displacement is in the negative direction of the Y-axis.

- **Operations**: You have already experienced working with the vector subtraction operator. This operator and others will be explored later in this chapter.

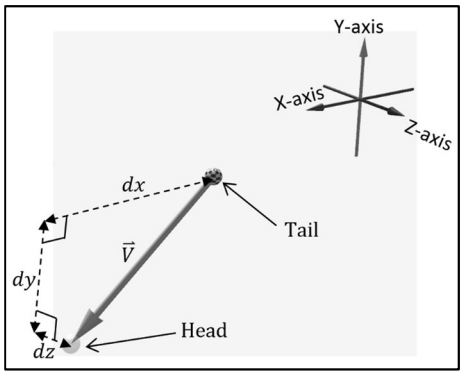

Figure 4-2. *A vector with its head and tail*

Position Vectors

For new learners of vectors, a common point of confusion is the position that defines a vector. For example, since the vector,

$$\vec{V}_d = P_2 - P_1$$

defines the distance and direction from position P_1 to P_2, one may arrive at the wrong assumption that the vector $\vec{V}_d$, is "defined at position P_1." You will begin the exploration of vectors by analyzing this potentially confusing issue head-on and learn that vectors are defined independent of any specific position, and in fact, can be applied to any position.

Notice that the positions that define the vector $\vec{V}_d$, P_1 and P_2, are variables, indicating that this formula is true for any point located at any position. In the special case where P_1' is located at the origin of the Cartesian Coordinate System, $(0,0,0)$, then,

$$\begin{aligned}\vec{V}_d' &= P_2 - P_1' \\ &= (x_2 - x_1', y_2 - y_1', z_2 - z_1') = (x_2 - 0, y_2 - 0, z_2 - 0) \\ &= (d_x, d_y, d_z) = (x_2, y_2, z_2)\end{aligned}$$

Which shows that P_2 can be interpreted as a vector (x_2, y_2, z_2) from the origin. In fact, any position in the Cartesian Coordinate System at (x, y, z) can be interpreted as x-, y-, and z-displacements measured along the three major axes from the origin position and thus all positions in the Cartesian Coordinate System can be interpreted as vectors from the origin. In this way, the position of a point is also referred to as a **Position Vector**. In general, in the absence of a specific context, it is convenient to consider given tuples of three floats, e.g., (x, y, z), as a position vector.

■ ***Note*** *The origin position, (0, 0, 0), is a special position vector and is referred to as the* ***zero vector****.*

Following a Vector

Refer to Figure 4-1 again, recall that the detailed definition of vector $\vec{V}_d$ is as follows.

$$\begin{aligned}\vec{V}_d &= P_2 - P_1 \\ &= (x_2 - x_1, y_2 - y_1, z_2 - z_1) \\ &= (d_x, d_y, d_z)\end{aligned}$$

Remember that $\vec{V}_d$ defines the distance and direction from position P_1 to P_2. A subtle, but logical interpretation of this definition is that position P_2 can be arrived at if an object begins at position P_1 and travels along the X-axis by d_x, the Y-axis by d_y, and the Z-axis by d_z. This interpretation can be described as "following a vector" from P_1 to P_2, and can be verified mathematically as follows.

- P_2 x-position $= x_1 + d_x = x_1 + (x_2 - x_1) = x_2$
- P_2 y-position $= y_1 + d_y = y_1 + (y_2 - y_1) = y_2$
- P_2 z-position $= z_1 + d_z = z_1 + (z_2 - z_1) = z_2$

Not surprisingly, "following a vector" is expressed as:

$$\begin{aligned}P_2 &= P_1 + \vec{V}_d \\ &= \left(x_1 + d_x,\ y_1 + d_y,\ z_1 + d_z\right) \\ &= (x_1 + x_2 - x_1,\ y_1 + y_2 - y_1,\ z_1 + z_2 - z_1) \\ &= (x_2, y_2, z_2)\end{aligned}$$

Graphically, you can imagine placing the tail of $\vec{V}_d$ at location P_1 and "follow the vector" to the head of the vector, to position P_2.This is how you can get from one position to another when you don't know the location of your next position, but you do have the distant and direction ($\vec{V}_d$) to get there.

■ ***Note*** *You have seen the vector subtraction operator where the corresponding coordinate values are subtracted. Here you see vector addition operator, where the corresponding coordinate values are added. The details of vector subtraction and addition will be visited again later in this chapter.*

Following a Vector from Different Positions

Following a vector, $\vec{V}_d$, from a given position, P_1, is also referred to as, "applying the vector $\vec{V}_d$ at P_1." Since both $\vec{V}_d$ and P_1 are variables, the equation,

$$P_2 = P_1 + \vec{V}_d$$

is true and applicable for any vector and any position. This concept is analyzed in detail in this section.

Figure 4-3 illustrates the alternative interpretations of the Cartesian Coordinate position, P_d, and the associated tuple of three floating-point values, (x_d, y_d, z_d).

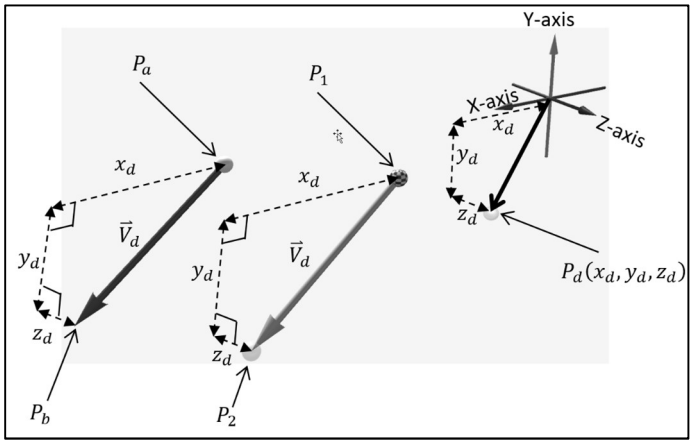

Figure 4-3. *Positions, position vectors, and applying vectors at different positions*

The top right corner of Figure 4-3 illustrates that P_d is a position in 3D space located at distances x_d, y_d, and z_d from the origin. In this way, (x_d, y_d, z_d), is the position vector that identifies the location of the point P_d. The two spheres and the associated arrows on the left side of Figure 4-3 illustrate interpreting the three-float tuple, (x_d, y_d, z_d), as the vector $\vec{V}_d$. If you apply $\vec{V}_d$ to position P_1, you will arrive at position P_2. If you apply $\vec{V}_d$ to position P_a, then you will arrive at P_b. In this case, you know that the Cartesian Coordinate positions for P_1 and P_a are as follows.

$$P_1 = (x_1, y_1, z_1)$$

$$P_a = (x_a, y_a, z_a)$$

Then, the Cartesian Coordinate positions for P_2 and P_b must be as follows.

$$P_2 = P_1 + \vec{V}_d = (x_1 + x_d, y_1 + y_d, z_1 + z_d) = (x_2, y_2, z_2)$$

$$P_b = P_a + \vec{V}_d = (x_a + x_d, y_a + y_d, z_a + z_d) = (x_b, y_b, z_b)$$

These equations are true for any x-, y-, or z-values. This is to say that P_1 (and P_a) can be located at any position in the 3D Cartesian Coordinate System. In this way, a vector can indeed be applied to any position. In all cases, "following a vector" is simply placing the tail of the vector at the starting position, with the head of the vector always being located at the destination position.

Recall that when P_1 is located at the origin, or, when

$$P_1' = (x_1', y_1', z_1') = (0, 0, 0)$$

then,

$$P_2 = P_1' + \vec{V}_d = (0 + x_d, 0 + y_d, 0 + z_d) = (x_d, y_d, z_d) = P_d$$

Observe that when P_1' is located at the origin, then P_d is a coordinate position. This means that the associated tuple of three floating-point numbers, (x_d, y_d, z_d), can be interpreted as the vector $\vec{V}_d$ being applied to the origin, $(0,0,0)$. This is true for any coordinate position. For example, the tuple of three floats, (x_1, y_1, z_1), that defines the position P_1, also describes the vector $\vec{V}_1$ being applied to the origin. The reverse is also true, that a given vector, $\vec{V}$, can be interpreted as the Cartesian Coordinate

position, P, or a position vector. Without sufficient contextual information, such as the tail position, vectors are always depicted and visualized as a line segment with their tail located at the origin.

If you are given a three valued tuple, (x, y, z), without context, you can assume it is a position vector. If you are given a vector, $\vec{V}$, without context, you can assume it is a coordinate position (that it starts from the origin). The next example will cover the details of position vectors and help you understand working with a coordinate position and interpreting that position as a position vector.

The Position Vectors Example

The focus of this example is to allow you to visualize a position vector and then to apply that vector at different locations. This example allows you to adjust, examine, and verify that vectors are defined independent of any given position. Figure 4-4 shows a screenshot of running the EX_4_1_PositionVectors scene from the Chapter-4-Examples project.

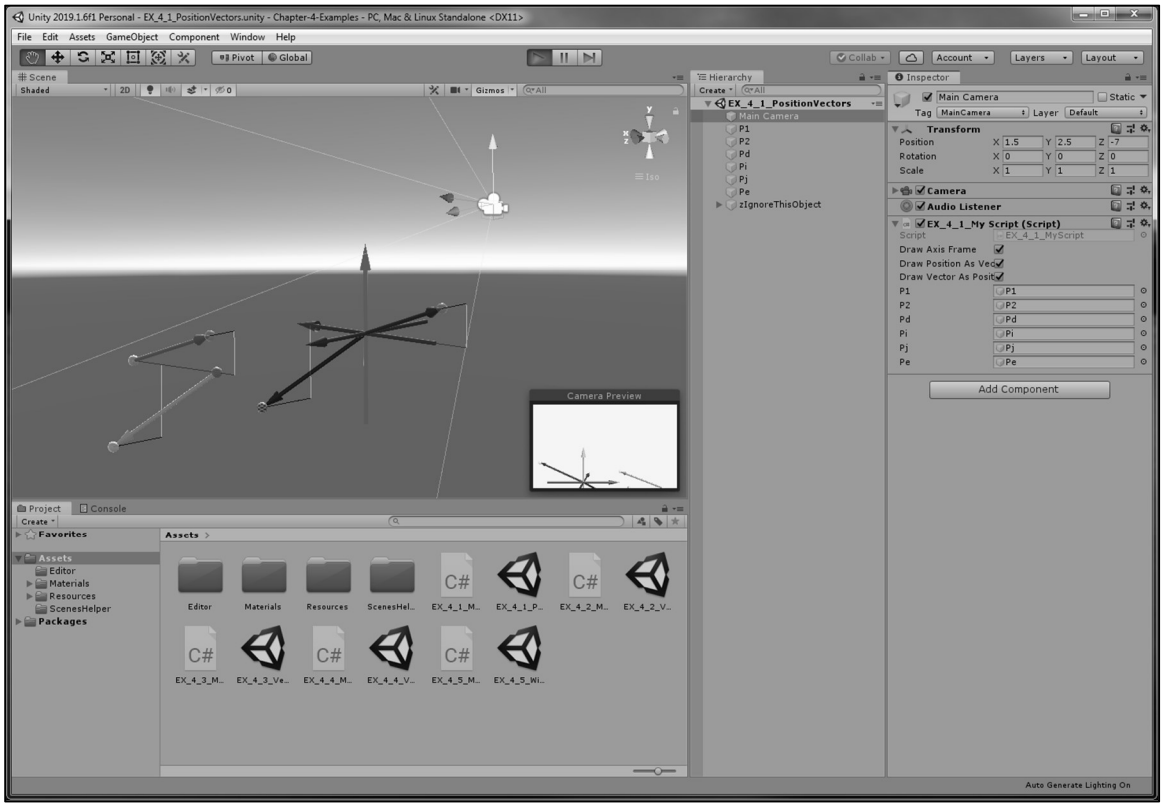

Figure 4-4. *Running the Position Vectors example*

The goals of this example are for you to:

- Understand the relationship between positions, position vectors, and applying vectors at positions

- Manipulate a position and observe the position vector applied at a different location
- Manipulate two positions to define a vector and observe the vector as a position vector
- Examine the implementation and application of vectors
- Increase familiarity with the Vector3 class

Examine the Scene

Take a look at the EX_4_1_PositionVectors scene and observe the predefined game objects in the Hierarchy Window. There you will find MainCamera and six other game objects that will assist in interpreting vectors from two alternative perspectives. These game objects are P1, P2, Pd, Pi, Pj, and Pe. This example will allow you to manipulate the head position of a position vector and to observe how the defined vector can be applied to any position. This example will also allow you to manipulate the positions of two points, observe how those two positions can define a vector, and how the defined vector can be shown as a position vector at the origin.

Analyze MainCamera MyScript Component

The MyScript component on MainCamera presents nine variables you can interact with. Three of these variables are toggle switches to control what you want to show and hide in the scene and the other six variables can be categorizes into two sets of three variables each.

- Position Vector:
  - P1: The reference to the P1 game object
  - P2: The reference to the P2 game object
  - Pd: The reference to the Pd game object
- Vector Defined by Two Points:
  - Pi: The reference to the Pi game object
  - Pj: The reference to the Pj game object
  - Pe: The reference to the Pe game object
- Toggles:
  - DrawAxisFrame: A toggle determining if the axis frame should be drawn
  - DrawPositionAsVector: A toggle determining if a position should be drawn as a vector
  - DrawVectorAsPosition: A toggle determining if a vector should be drawn as a position

---

■ **Note** *For convenience, whenever appropriate, the rest of the examples in this book will assign identical names to the game objects in the scene and the corresponding reference variables in* MyScript.

Interact with the Example

Click on the Play Button to run the example. Notice that, by default the DrawVectorAsPosition toggle is set to off and the corresponding game objects, Pi, Pj, and Pe are not displayed. This is so you can focus on the position vector defined by Pd and apply it at position P1. Select MainCamera and try toggling the DrawAxisFrame on to observe the axis frame in the scene. You only need to show this axis frame when you want to verify the location of the origin and the directions of the major axes. Feel free to hide the axis frame and to show it again whenever you need a reference.

Position Vector

First, verify that Pi, Pj, and Pe are not displayed by selecting these objects in the Hierarchy Window and confirming that they are inactive (the checkbox next to their name in the Inspector Window should be unchecked). Then, select P2 and try to manipulate its position. You will notice that whenever you change a value in P2's transform component in the Inspector Window, it reverts back to its old value. This is because P2's position is under the control of MyScript. Now select and manipulate the position of Pd and verify the following.

- Notice the thin red, green, and blue lines connecting from the origin to position Pd. Switch the DrawAxisFrame on and off to verify that these three lines are parallel to the corresponding X-, Y-, and Z-axis. The lengths of these three lines are x_d ,y_d and z_d, which are the corresponding values of the coordinate position of Pd.
- The position vector is the black vector with its tail at the origin and its head at the current Pd location. This vector represents interpreting the coordinate values of Pd, (x_d, y_d, z_d), as the x-, y-, and z-components of vector $\vec{V}_d$.
- Move Pd to a position close to the origin, e.g., $(0.1, 0.1, 0.1)$, and notice that the black vector is now very small and difficult to observe. When Pd is moved to exactly the origin, the black vector becomes the zero vector and vanishes. The zero vector is a special case that describes a zero displacement. As you will learn, the definition of many vector operations specifically excludes the zero vector. These will be pointed out as you learn about them in future sections and chapters.

You have observed displaying a position as a position vector (a vector from the origin to the position) which demonstrates that all positions in the Cartesian Coordinate System can be interpreted as position vectors. Now, select and manipulate the position of P1 and notice the following.

- Independent of the location of P1, the white vector is always identical to the black position vector where they are parallel and have the same length. The only difference between these vectors is that the white vector has its tail at P1 and not the origin. You can verify this by observing that the thin red, green, and blues lines that connect P1 to P2 are the same length as the thin red, green, and blues lines that connect the origin to Pd.
- Position P2 is always at the head of white vector. In this case, P2 is computed as follows.

$$P_2 = P_1 + \vec{V}_d$$

You have observed that when you apply a position vector at an arbitrary position (P1), that the position vector and the applied vector are indeed identical, and that the only difference between them is that they are located, or applied, at a different position. This illustrates that vectors are independent of positions, meaning that once a vector is defined it can be applied to any position. It also demonstrates that a vector absent of any position information, should be, and are, interpreted as position vectors - vectors originating from the origin. This part of the example has shown that a position in 3D space is simply a vector from the origin to that position.

Vector Defined by Two Points

Now toggle DrawPositionAsVector off and switch DrawVectorAsPosition to on. Now verify that P1, P2, and Pd, are hidden by selecting them in the Hierarchy Window. Next, select and try to change the position of Pe. Note that just like with P2, Pe's position is being set by MyScript and thus cannot be changed from the Inspection Window. Now, select and change the positions of Pi and Pj and notice the following.

- The pink vector, $\vec{V}_e = (x_e, y_e, z_e)$, is defined by the positions Pi, (x_i, y_i, z_i), and Pj (x_j, y_j, z_j), where, $\vec{V}_e = P_j - P_i$, or
  - $x_e = x_j - x_i$, which is the displacement along the X-axis (the length of the thin red line)
  - $y_e = y_j - y_i$, which is the displacement along the Y-axis (the length of the thin green line)
  - $z_e = z_j - z_i$, which is the displacement along the Z-axis (the length of the thin blue line)
- Independent of the locations of Pi and Pj, the pink and purple vectors are practically identical, having the same length and are parallel to each other (they have same direction). The only difference between them is the location of their tail positions. The pink vector has a tail located at position Pi and the purple vector's tail is located at the origin.
- The purple vector's head position is always at, Pe, (x_e, y_e, z_e). Note how the coordinate component values are the same values as that of $\vec{V}_e$, indicating that Pe position is the position vector $\vec{V}_e$.

By confirming the previous points you have observed that any vector, $\vec{V}_e = (x_e, y_e, z_e)$, is equivalent to the coordinate position P_e, (x_e, y_e, z_e) and can be displayed as a position vector with tail at the origin.

Details of MyScript

Open MyScript and examine the source code in the IDE. The instance variables are as follows.

```
// For visualizing the two vectors
public bool DrawAxisFrame = true; // Draw or Hide The AxisFrame
public bool DrawPositionAsVector = true;
public bool DrawVectorAsPosition = true;

private MyVector ShowVd;        // From Origin to Pd
private MyVector ShowVdAtP1;    // Show Vd at P1
private MyVector ShowVe;        // From Origin to Pe
private MyVector ShowVeAtPi;    // Ve from Pi to Pj

// Support position Pd as a vector from P1 to P2
public GameObject P1;    // Position P1
public GameObject P2;    // Position P2
public GameObject Pd;    // Position vector: Pd

// Support vector defined by Pi to Pj, and show as Pe
public GameObject Pi;    // Position Pi
public GameObject Pj;    // Position Pj
public GameObject Pe;    // Position vector: Pe
```

All of the public variables for MyScript have been discussed when analyzing the MainCamera's MyScript component. The four private variables of MyVector data type are defined to support the visualization of the vectors you have observed previously.

- ShowVd: Used for visualizing the position vector of Pd (the black vector)

- ShowVdAtP1: Used for visualizing the vector at position P1 (the white vector)
- ShowVe: Used for visualizing the position vector of Pe (the purple vector)
- ShowVeAtPi: Used for visualizing the vector at position Pi (the pink vector)

As in the case of the previous custom classes such as MyBoxBound and MySphereBound, MyVector is defined specifically for visualizing a vector and is irrelevant for understanding the math being discussed in this book. For example, you can always run the examples with all code concerning the MyVector data type removed, but the visualization of these vectors (black, white, pink, etc.) will no longer exist. You can see a screenshot of the MyVector class in Figure 4-5, which shows that MyVector is indeed defined for the drawing of a vector.

```
public class MyVector
{
    private functionality for drawing support

    public MyVector()...          // Constructor

    public float Magnitude...     // Size of the vector
    public Vector3 Direction...   // Direction of the vector
    public Vector3 VectorAt...    // The location to draw the vector

    // Drawing Support
    public bool DrawVector...       // Draw or Hide the interval
    public bool DrawVectorComponents ...
    public Color VectorColor...     // Color to draw

    // A vector from src to dst
    public void VectorFromTo(Vector3 src, Vector3 dst)...

    // A vector at src, with direction: dir and magnitude: len
    public void VectorAtDirLength(Vector3 pos, Vector3 dir, float len)...
}
```

Figure 4-5. *The MyVector class*

The Start() function for MyScript is listed as follows.

```
void Start() {
    Debug.Assert(P1 != null);    // Verify proper setting in the editor
    Debug.Assert(P2 != null);
    Debug.Assert(Pd != null);
    Debug.Assert(Pi != null);
    Debug.Assert(Pj != null);
    Debug.Assert(Pe != null);

    // To support show position and vector at P1
    ShowVd = new MyVector {
        VectorColor = Color.black,
        VectorAt = Vector3.zero       // Always draw Vd from the origin
    };
    ShowVdAtP1 = new MyVector {
        VectorColor = new Color(0.9f, 0.9f, 0.9f)
    };

    // To support show vector from Pi to Pj as position vector
    ShowVe = new MyVector {
```

```
            VectorColor = new Color(0.2f, 0.0f, 0.2f),
            VectorAt = Vector3.zero     // Always draw Vv from the origin
        };
        ShowVeAtPi = new MyVector() {
            VectorColor = new Color(0.9f, 0.2f, 0.9f)
        };
}
```

As seen, the Start() function verifies proper public variable setup in the Hierarchy Window, and instantiates and initializes the private MyVector variables to their respective colors. Note that ShowVd and ShowVe are defined to display position vectors and are therefore initialized to show the vectors starting from the origin (Vector3.zero). The Update() function is listed as follows.

```
void Update()
{
    Visualization on/off: show or hide to avoid clustering

    Position Vector: Show Pd as a vector at P1

    Vector from two points: Show Ve as the position Pe
}
```

The Update() function is divided into three separate #region areas according to the logic they perform and for readability. The details of these regions are explained in the next three sections.

Region: Visualization on/off

The code in this region, listed as follows, simply sets the active flag on the relevant game objects for displaying or hiding whichever game objects the user toggles via the MyScript component on MainCamera.

```
#region  Visualization on/off: show or hide to avoid clutering
AxisFrame.ShowAxisFrame = DrawAxisFrame;      // Draw or Hide Axis Frame
P1.SetActive(DrawPositionAsVector);           // Support for position as vector
P2.SetActive(DrawPositionAsVector);
Pd.SetActive(DrawPositionAsVector);
Pi.SetActive(DrawVectorAsPosition);           // Support for vector as position
Pj.SetActive(DrawVectorAsPosition);
Pe.SetActive(DrawVectorAsPosition);
ShowVdAtP1.DrawVector = DrawPositionAsVector;     // Display or hide the vectors
ShowVd.DrawVector = DrawPositionAsVector;
ShowVeAtPi.DrawVector = DrawVectorAsPosition;
ShowVe.DrawVector = DrawVectorAsPosition;
#endregion
```

Region: Position Vector

The code in this region, listed as follows, is only active when the DrawPositionAsVector toggle is set to true.

```
#region Position Vector: Show Pd as a vector at P1
if (DrawPositionAsVector) {
    // Use position of Pd as position vector
    Vector3 vectorVd = Pd.transform.localPosition;

    // Step 1: take care of visualization
    //           for Vd
```

```
        ShowVd.Direction = vectorVd;
        ShowVd.Magnitude = vectorVd.magnitude;

        //          apply Vd at P1
        ShowVdAtP1.VectorAt = P1.transform.localPosition; // Always draw at P1
        ShowVdAtP1.Magnitude = vectorVd.magnitude;          // get from vectorVd
        ShowVdAtP1.Direction = vectorVd;

        // Step 2: demonstrate P2 is indeed Vd away from P1
        P2.transform.localPosition = P1.transform.localPosition + vectorVd;
}
#endregion
```

In this case, as illustrated by the bolded font in the code listing, the position of Pd, Pd.transform.localPosition, is interpreted as a vector, vectorVd, or $\vec{V}_d$. In Step 1, vectorVd is drawn via the ShowVd variable. Recall that ShowVd is initialized to be drawn at the origin. For this reason, ShowVd is simply drawing vectorVd, or the coordinate values of Vd, as a position vector. In order to show the same vector at position P1, the magnitude (length) and direction of ShowVdAtP1 are assigned the corresponding values from vectorVd and is then displayed at the location of P1, P1.transform.localPosition, instead of the origin like that of vectorVd. In Step 2, once again shown in bolded font, P2's position is set as $P_2 = P_1 + \vec{V}_d$ which will always place P2 at the head of $\vec{V}_d$. This repeated updating of P2's position is the reason why when you interacted with this example, you were not able to move the P2 game object.

In the Cartesian Coordinate System, positions are defined by three-float tuples. So far, this example shows that the same three-float tuple can be interpreted as a vector. This alternative interpretation allows vectors to be used as a tool for describing physical behaviors, like object movements. This topic will be covered in detailed in a later section of this chapter.

Region: Vector from Two Points

The code in this region, listed as follows, is only active when the DrawVectorAsPosition toggle is set to true.

```
#region Vector from two points: Show Ve as the position Pe
if (DrawVectorAsPosition) {
    // Use from Pi to Pj as vector for Ve
    Vector3 vectorVe = Pj.transform.localPosition - Pi.transform.localPosition;

    // Step 1: Take care of visualization
    //          for Ve: from Pi to Pj
    ShowVeAtPi.VectorFromTo(Pi.transform.localPosition,
                            Pj.transform.localPosition);
    //          Show as Ve at the origin
    ShowVe.Direction = vectorVe;
    ShowVe.Magnitude = vectorVe.magnitude;

    // Step 2: demonstrate Pe is indeed Ve away from the origin
    Pe.transform.localPosition = vectorVe;
}
#endregion
```

As illustrated by the bolded font in the code listing, the vector vectorVe or $\vec{V}_e$, is computed based on the positions of Pi and Pj according to the formula,

$$\vec{V}_e = P_j - P_i$$

In Step 1, ShowVeAtPi is set to be drawn as a vector between Pi and Pj's positions. ShowVe's direction and magnitude are assigned by the corresponding values of vectorVe. Recall that the draw position of ShowVe was initialized to the origin, and thus ShowVe is showing vectorVe as a position vector. In Step 2, again shown in bolded font, the position of Pe is set to the corresponding x-, y-, and z-component values of vectorVe, literary showing vectorVe as a coordinate position. Similar to the case of P2's position, in this case, Pe is continuously updated by the script and thus the user has no control over the position of Pe while the scene is running.

In general, the ability to interpret a given vector as a position allows all vectors to be plotted as position vectors from the origin, supporting straightforward visualization comparisons across multiple vectors. You have completed the cycle of interpreting positions as vectors and now vectors as positions. This entire discussion is designed to demonstrate that, once defined, a vector as an entity can be analyzed and applied at any position because its definition is independent of any specific position.

■ **Note** *The vector from Pi to Pj is computed by subtracting Pi from Pj,*

$$\vec{V}_e = P_j - P_i$$

The order of subtraction is important. Reversing the subtraction order, $P_i - P_j$, computes a vector from Pj to Pi. Vector subtraction will be discussed in detailed later in this chapter.

Take Away from This Example

This example presents you with two ways to define, manipulate, and interpret a vector. The first method is based on initializing a starting point (e.g., the origin) and then selecting the ending position. The second method is based on defining a vector between two explicitly controlled positions. In all interactions, all four vectors describe how to move from one position to another: from origin to Pd (black), from P1 to P2 (white), from Pi to Pj (pink), and from origin to Pe (purple).

You have seen that it does not matter where a vector is applied (or drawn), if the encoded distances and direction information are the same, the underlying vectors are the same. You have also witnessed that a vector can be treated as a position, and a position can be treated as a vector.

Relevant mathematical concepts covered include:

- A vector describes the movement from one position to another.
- The vector between two given positions is defined by the differences between the corresponding coordinate values in the x-, y-, and z-components.
- The Cartesian Coordinate values for any position P, (x, y, z), describes the displacements from the origin to the position P. For this reason, the (x, y, z) values of any position can be interpreted as a vector between the origin and the position. This interpretation of the coordinate position is referred to as position vector.
- All positions in the Cartesian Coordinate system can be interpreted as position vectors.

- The zero vector is the position vector of the origin. This vector describes a displacement with zero distance, or a position moving back onto itself. This is a special case vector; thus many vector operations cannot operate or do not work on the zero vector.
- Vectors are independent of positions, thus, once defined, a vector can be applied to any position.
- In the absence of position information, vectors are often drawn as a position vector, a line segment from the origin to the coordinate position defined by the x-, y-, and z-component values of that vector.

Unity Tools:

- MyVector: A custom defined class to support the visualization of vectors
- AxisFrame.ShowAxisFrame: A Boolean flag to control the showing of the Cartesian Coordinate origin and axes directions

---

■ ***Note*** *The Unity* Vector3 *data type closely encapsulates the concept of a vector. From the code listing in the* Update() *function you can observe the power and convenience of working with proper data abstraction. With the Unity* Vector3 *abstraction you can avoid the nuisance of re-typing similar code for individual values of each major axis when computing distances between positions, or when following a vector. For the rest of this book, with very few exceptions, such as when analyzing the detailed definitions of vector operations, you will work with the* Vector3 *class and will not work with the values of the individual coordinate axes.*

EXERCISES

Contrast the Creation of $\vec{V}_d$ and $\vec{V}_e$

Note that $\vec{V}_d$ *is created via a single position being interpreted as a position vector, while* $\vec{V}_e$ *is created by subtracting two positions explicitly. Nevertheless, both methods can accomplish the creation of the same vector. For example, move the position of* Pi *to overlap* P1*. This can be accomplished by running the game, selecting* P1 *in the Hierarchy Window, taking note of the position values of the* Transform *component of* P1*, and copying these values to be the position*

values of Pi's Transform component. You can now adjust Pj, or Pd to try to align $\vec{V}_e$ with $\vec{V}_d$.

Switch Vector Creation Methods

You can take advantage of the observation that both position vector and the difference between two points can create the same vector. Edit MyScript and remove Pe, Pi, and Pj variables. Instead, include a new Boolean flag CreateWithPositionVector which will allow P1, P2, and Pd to behave as Pe, Pi, and Pj did.

- when CreateWithPositionVector is true, let the user manipulate Pd to create the vector and show the vector at P1. In this case, P2 is computed based on the vector defined and the user will not be able to adjust P2.
- when CreateWithPositionVector is false, let the user manipulate both P1 and P2 and use the difference between these two points to compute the position vector to Pd. In this case, Pd is computed based on the vector defined and the user will not be able to adjust Pd.

Note the "two ways to define a vector" logic is similar to that of the "two ways to define a bounding box." You can refer to the Update() function of the EX_2_2_BoxBounds_IntervalsIn3D scene of Chapter-2-Examples project for a template of the control logic required for this exercise.

Verify Vector Size, or Length, or Magnitude

A vector describes the movement from one position to another, it encapsulates both the distance and the direction to travel. You have seen the distance being referred to as "magnitude," it is also commonly referred to as the "size" or "length" of the vector. Edit MyScript to print the size of each of the vectors, either via public float variables or via Debug.Log() function calls. Verify that both ShowVd and ShowVdAtP1, and ShowVe and ShowVeAtPi are indeed two sets of vectors with identical length.

Manipulate Vector Lengths

Manipulate the two vectors in this example such that $\vec{V}_d = (2, 0, 0)$ and $\vec{V}_e = (0, 2, 0)$. Notice that in this case, $\vec{V}_d$ and $\vec{V}_e$ have the same lengths of 2.0. However, the two vectors are pointing towards drastically different directions: towards positive X-axis, and Y-axis. Notice that it is possible to define two vectors with identical length but with very different directions.

Verify Vector Directions

You can verify two vectors are the same by printing out the values of the x-, y-, and z-components. Edit MyScript to print the coordinate values of ShowVe and ShowVeAtPi to verify that these two vectors are indeed exactly the same. With previous exercises on vector size, the obvious question is, "is it possible to manipulate the two vectors such that they are pointing in the same direction but with different lengths?" The short answer is yes. For example, consider vectors, $(1, 0, 0)$ and $(2, 0, 0)$. Both are pointing towards the positive x-direction, but the lengths are 1 and 2. The general consideration for this question is slightly more involved and is the topic for the next section.

Vector Algebra: Scaling

A vector encodes both a distance and a direction, describing how an object can move from position $P_1\ (x_1, y_1, z_1)$, in a straight line, and arrive at $P_2\ (x_2, y_2, z_2)$. You know that a vector, $\vec{V}_a$, that describes this movement can be defined as follows.

$$\begin{aligned}\vec{V}_a &= P_2 - P_1 \\ &= (x_2 - x_1, y_2 - y_1, z_2 - z_1) \\ &= (x_a, y_a, z_a)\end{aligned}$$

The distance, d, between the two points is referred to as the size (or magnitude, or length) of the vector, and is labeled with the symbol $\|\vec{V}_a\|$. The size of a vector is defined as follows.

$$d = \|\vec{V}_a\| = \sqrt{{x_a}^2 + {y_a}^2 + {z_a}^2}$$

The size of a vector can be scaled. For example, if there is a vector $\vec{V}_b = (x_b, y_b, z_b) = (5x_a, 5y_a, 5z_a)$ then,

$$\begin{aligned}\|\vec{V}_b\| &= \sqrt{{x_b}^2 + {y_b}^2 + {z_b}^2} \\ &= \sqrt{(5x_a)^2 + (5y_a)^2 + (5z_a)^2} \\ &= \sqrt{25({x_a}^2 + {y_a}^2 + {z_a}^2)} \\ &= 5\sqrt{{x_a}^2 + {y_a}^2 + {z_a}^2} \\ &= 5\|\vec{V}_a\|\end{aligned}$$

Note that in general, the observed relationship is true for any floating-point number, s. That is, if

$$\vec{V}_a = (x_a, y_a, z_a)$$

and

$$\vec{V}_b = (sx_a, sy_a, sz_a)$$

then,

$$\|\vec{V}_b\| = s\|\vec{V}_a\|$$

The length or magnitude of $\vec{V}_b$ is s times that of $\vec{V}_a$. In this case, $\vec{V}_b$ is described as, "scaling $\vec{V}_a$ by a factor s," or simply, "scaling $\vec{V}_a$ by s" and is expressed as,

$$\vec{V}_b = s\vec{V}_a$$

■ **Note** *While it is always true that if $\vec{V}_b = s\vec{V}_a$, then, $\|\vec{V}_b\| = s\|\vec{V}_a\|$. The reverse is not always true. For example, if $\vec{V}_a = (1, 0, 0)$ and $\vec{V}_b = (0, s, 0)$, then, in this case, it is true that, $\|\vec{V}_b\| = s\|\vec{V}_a\|$, but $\vec{V}_b = s\vec{V}_a$ is certainly not true.*

Figure 4-6 illustrates an example where $\vec{V}_a = (x_a, 0, 0)$, $\vec{V}_b = 1.5\vec{V}_a$, and $\vec{V}_c = \frac{1}{x_a}\vec{V}_a$.

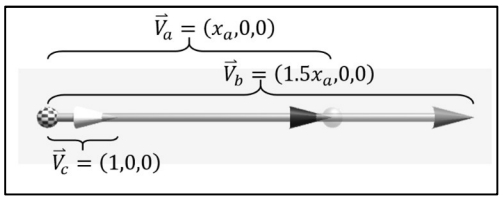

Figure 4-6. *Scaling of a vector that is in the x-direction*

Referring to Figure 4-6, you now know that,

- $\vec{V}_b = 1.5\vec{V}_a = (1.5x_a, 0, 0)$
- $\vec{V}_c = \frac{1}{x_a}\vec{V}_a = (\frac{1}{x_a}x_a, 0, 0) = (1, 0, 0)$

Additionally, you know when x_a is a positive number, the lengths of the three vectors in Figure 4-6 are as follows.

$$\|\vec{V}_a\| = \sqrt{{x_a}^2 + 0^2 + 0^2} = x_a$$

$$\|\vec{V}_b\| = 1.5\|\vec{V}_a\| = 1.5x_a$$

$$\|\vec{V}_c\| = \frac{1}{x_a}\|\vec{V}_a\| = 1$$

Lastly, and very importantly, based on your knowledge of the Cartesian Coordinate System and so far in this chapter, you know that although the vectors in Figure 4-6 have different lengths, the three vectors overlap perfectly and are all pointing in the positive X-axis direction. This overlap shows that scaling a vector only changes the distance that it encodes and does not affect the direction. It turns out, as illustrated in Figure 4-7, this statement is true for any direction.

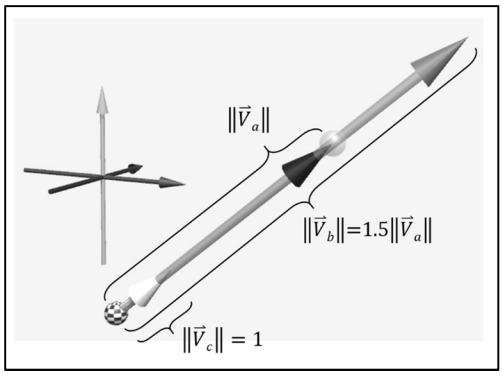

Figure 4-7. *Scaling of an arbitrary vector*

Figure 4-7 shows three vectors with the same lengths as of those in Figure 4-6.

- Vector $\vec{V}_a$ with magnitude $\|\vec{V}_a\|$
- Vector $\vec{V}_b = 1.5\vec{V}_a$ with magnitude $1.5\|\vec{V}_a\|$
- Vector $\vec{V}_c = \frac{1}{\|\vec{V}_a\|}\vec{V}_a$ with magnitude of 1.0

Notice that, in exactly the same manner as the vectors in the X-axis direction (Figure 4-6), these three vectors all point in the same direction as each other. In all cases, scaling a vector only affects its size and not the direction. In general, scaling a vector by any positive number will result in a vector that is in the same direction, while scaling by a negative number will flip the direction of that vector. This means when a positive x-direction vector is scaled by a negative value, the resulting vector will point in the negative x-direction. Scaling by a negative number is left as an exercise for you to complete in the next example.

Similar to how multiplying scaling factors to the number zero will produce a result of zero, scaling a zero vector has no effect and will result in the same zero vector.

Normalization of Vectors

Vector $\vec{V}_c$ in Figures 4-7 is the result of scaling an existing vector by the inverse of the length of that vector. This is interesting because with such a specific scaling factor, the magnitude of $\vec{V}_c$ is guaranteed to be 1. As you will see frequently in the rest of this book, and is true in general, vectors with a magnitude of 1 are important as they enable convenient computations in many situations.

A vector with a magnitude of 1 is so important that it has its own symbol, $\hat{V}$, which is the same as the original symbol for a vector, but replaces the arrow above the 'V' with a cap. This vector has a special name, **normalized vector** or **unit vector**. The process of computing a normalized vector is called **vector normalization**. In general, it is always the case that for any non-zero vector, $\vec{V} = (x, y, z)$,

- Magnitude of Vector $\vec{V}$

$$\|\vec{V}\| = \sqrt{x^2 + y^2 + z^2}$$

- Normalization of Vector $\vec{V}$

$$\begin{aligned}\hat{V} &= \frac{1}{\|\vec{V}\|}\vec{V} \\ &= \frac{1}{\sqrt{x^2+y^2+z^2}}\vec{V} \\ &= \left(\frac{x}{\sqrt{x^2+y^2+z^2}}, \frac{y}{\sqrt{x^2+y^2+z^2}}, \frac{z}{\sqrt{x^2+y^2+z^2}}\right)\end{aligned}$$

Notice that normalization is a division by length. Recall that a zero vector has a length of zero, and from basic algebra that, division by zero is an undefined operation. This means that the zero vector cannot be normalized. This is the first case you encounter, but certainly not the last, that a vector operation is not applicable to the zero vector.

■ **Note** The vector normalization process involves a division by a square-root. Though with modern computers this computation cost is becoming less of a concern, it is still a good practice to pay attention to the need for normalization in general. For example, the Unity `Vector3` class defines the `sqrMagnitude` property to return the squared of a vector length, $\|\vec{V}\|^2$, which can be used when information

on vector length is needed, but not normalization. For example, when performing size comparisons, e.g., determining which vector is longer.

Direction of Vectors

The magnitude of a vector can be simply and effectively conveyed by a number. In contrast, the direction of a vector must be expressed in relation to a "frame of reference." For example, "in the x-direction" uses the X-axis as the frame of reference. In the 3D Cartesian Coordinate System, a direction can be described by using the X-, Y-, and Z-axes as the references. Such a description involves a reference direction and a rotation. For example, a direction that is defined by a rotation of the Y-axis about the Z-axis in the X-axis direction by 15 degrees. If you find that description difficult to follow, you are not alone. Fortunately, there are alternatives to describing the direction of a vector.

Recall that, as illustrated in Figure 4-7, the direction of a vector does not change when the vector is scaled. This means that a unit vector uniquely identifies the direction of all vectors with different lengths in that direction. For simplicity, both representationally and computationally, this book choses to identify the direction of a vector by referring to its unit vector. For example, for a given vector, $\vec{V}$, this book refers to its magnitude as, $\|\vec{V}\|$, and its direction as, $\hat{V}$. In the rest of this book, you will encounter phrases like, "the direction of $\vec{V}$", or "the direction of $\hat{V}$", both refer to the direction of the vector, $\hat{V}$.

Since the normalized zero vector is undefined, a zero vector has no direction.

The Vector Scaling and Normalization Example

This example demonstrates the results of scaling a vector and defining a vector with separate input for magnitude and direction. It allows you to adjust and examine the effects of changing the vector scaling factor, as well as control the creation of a vector via specifying its magnitude and direction. Figure 4-8 shows a screenshot of running the `EX_4_2_VectorScaling` scene from the `Chapter-4-Examples` project.

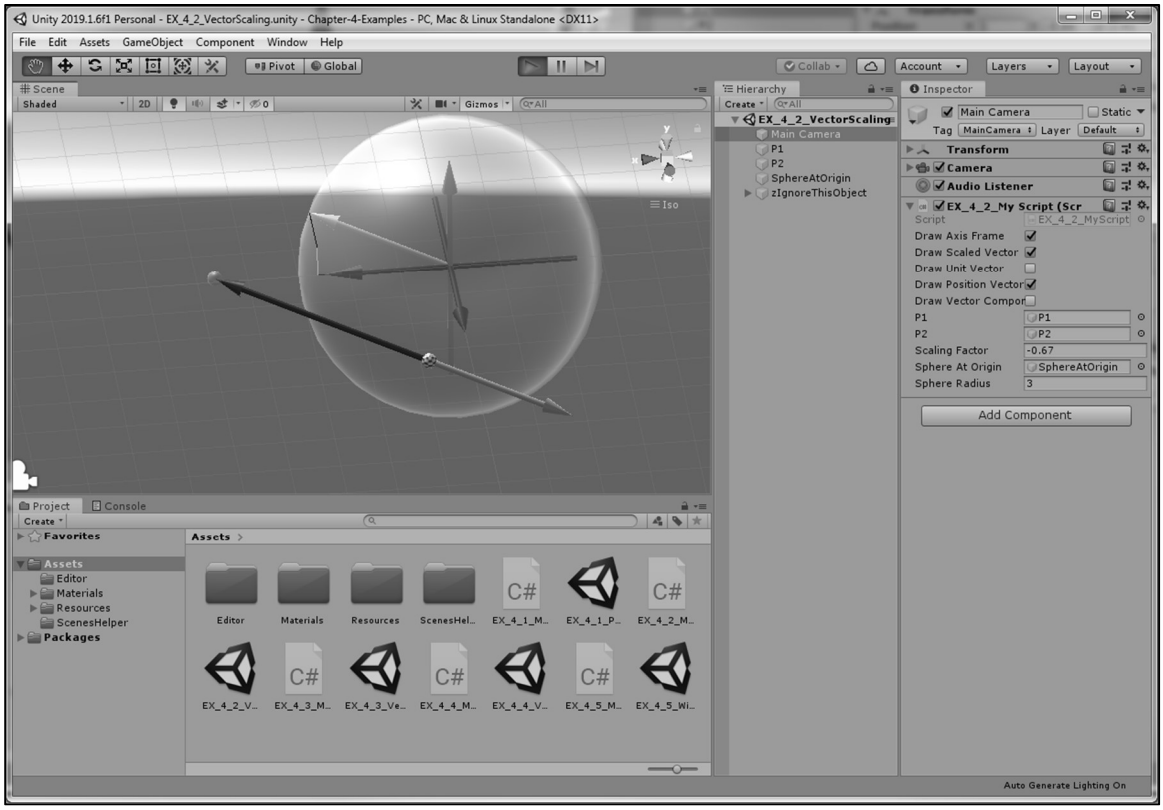

Figure 4-8. *Running the Vector Scaling example*

The goals of this example are for you to:

- Interact with and examine the effects of scaling vectors
- Experience defining vectors based on specifying their magnitude and direction
- Understand the effects of separately changing the magnitude and direction of a vector
- Examine the implementation of working with vectors

Examine the Scene

Take a look at the Example_4_2_VectorScaling scene and observe, besides MainCamera, the three predefined game objects in the Hierarchy Window: P1, P2, and SphereAtOrigin. As in the previous example, P1 and P2 together will allow you to define a vector, $\vec{V}_a$. The SphereAtOrigin is a transparent sphere located at the origin, where, you will create a position vector in the same direction as $\hat{V}_a$, with a magnitude that just touches the surface of this transparent sphere.

Analyze MainCamera MyScript Component

The MyScript component on MainCamera shows ten variables that can be categorized into three groups.

- Drawing control: Allows you to show or hide different vector functionality.
  - DrawAxisFrame: Show or hide the Cartesian Coordinate origin and reference axis frame
  - DrawScaledVector: Show or hide the scaled version of $\vec{V}_a$
  - DrawUnitVector: Show or hide the unit vector $\hat{V}_a$
  - DrawPositionVector: Show or hide the position vector that touches the SphereAtOrigin surface
  - DrawVectorComponents: Show or hide the x-, y-, and z-displacements of each vector. Notice that for clarity, when displayed, the position vector always draws its vector components.
- Definition of $\vec{V}_a$: Defines and allows manipulation of the vector $\vec{V}_a$
  - P1: The reference to the P1 game object
  - P2: The reference to the P2 game object
  - ScalingFactor: The factor to scale the vector $\vec{V}_a$ by
- Definition of a position vector: Defines and allows manipulation of the position vector
  - SphereAtOrigin: The reference to the SphereAtOrigin game object
  - SphereRadius: The radius of the SphereAtOrigin sphere and consequently the length of the position vector that will be parallel to $\hat{V}_a$

Interact with the Example

Click on the Play Button to run the example. Notice that by default, all vector drawing toggles are off so you should only be observing vector $\vec{V}_a$, the vector being drawn between positions P1 and P2. Now select the MainCamera and get ready to toggle drawing options and observe the following.

Scaled Vector

Toggle on the drawing option for DrawScaledVector to observe a slightly shorter pink vector in the same direction as $\vec{V}_a$. Now adjust the ScalingFactor variable and watch as the pink vector changes size. This pink vector is displaying the vector $\vec{V}_s$

$$\vec{V}_s = ScalingFactor \times \vec{V}_a$$

Notice three interesting intervals:

- $0 < ScalingFactor < 1$: $\vec{V}_s$ has a length shorter than $\vec{V}_a$ and is thus displayed as a vector embedded in $\vec{V}_a$.
- $ScalingFa \quad > 1$: $\vec{V}_s$ has a magnitude larger than $\vec{V}_a$ and is thus a vector that extends beyond $\vec{V}_a$.
- $ScalingFactor < 0$: $\vec{V}_s$ points in the reversed direction of $\vec{V}_a$. Note that the two vectors are drawn at the same position, P1 and that the two vectors do indeed extend in the exact opposite directions.

Normalized or Unit Vector

Toggle on the drawing option for DrawUnitVector to observe a short white vector embedded in $\vec{V}_a$. This is $\vec{V}_a$ normalized, or $\hat{V}_a$. Recall that $\hat{V}_a$ is computed by scaling $\vec{V}_a$ by the inverse of its magnitude, $\frac{1}{\|\vec{V}_a\|}$. Initially, $\vec{V}_a$ has a magnitude of 5, so if you adjust ScalingFactor to the value of $\frac{1}{5} = 0.2$, you will observe that the pink ($\vec{V}_s$) and white vectors overlap exactly. This overlap will stop once you adjust the ScalingFactor. Remember, $\vec{V}_s$ has a length that is ScalingFactor times the current $\|\vec{V}_a\|$, yet, the size of $\hat{V}_a$ is always 1.

Manipulate and set the positions of P1 and P2 to be identical, e.g., by copying values of P1's Transform component to that of P2. Now, notice error messages in the Console Window about NaN and that the normalized white vector now points downwards in the negative Y-axis direction. When positions of P1 and P2 are identical, $\vec{V}_a$ becomes the zero vector and $\hat{V}_a$ should be undefined. Later when you examine the implementation, you will notice that the zero vector condition is not checked. Here, you are observing the results of a common coding error: performing a vector operation without verifying if the operation is defined for the given vector. A responsible developer should always invoke pre-condition checking before performing the corresponding vector operations.

Position Vector from Direction and Magnitude

Toggle on the drawing option for DrawPositionVector to observe a navy-blue position vector, $\vec{V}_p$, that is parallel to $\hat{V}_a$ and has a magnitude that is defined by the SphereRadius variable.

$$\vec{V}_p = SphereRadius \times \hat{V}_a$$

You can verify this by adjusting SphereRadius and noting that the SphereAtOrigin game object (the transparent sphere) changes size and $\vec{V}_p$, while maintaining the direction of $\hat{V}_a$, adjusts its magnitude such that its tip touches the sphere surface. You can toggle off and hide the axis frame via DrawAxisFrame to observe the thin red, green, and blue vector components of $\vec{V}_p$, verifying that this vector does indeed just touch the sphere surface, indicating that the length of the vector is indeed the radius of the sphere.

This interaction shows that you can create a direction and a magnitude separately and combine them to create a desired vector. Note that since $\hat{V}_a$ is a unit vector, the size of $\vec{V}_p$, or $\|\vec{V}_p\|$, is simply SphereRadius. An important observation is that if a vector is defined by a size and a unit vector, then this size is the magnitude property of that vector. In the next section, you will see how this simple observation can be applied to implement the behavior of an object following a target.

Summary of interaction

Four vectors are created and examined in this example:

- $\vec{V}_a$: Vector between two user control positions, P1 and P2
- $\vec{V}_s = ScalingFactor \times \vec{V}_a$: A vector in the same or opposite direction as $\vec{V}_a$
- $\hat{V}_a = \frac{1}{\|\vec{V}_a\|} \times \vec{V}_a$: The normalized vector of $\vec{V}_a$, since this vector is always scaled by the inverse of its magnitude, it has a constant size of 1
- $\vec{V}_p = SphereRadius \times \hat{V}_a$: A constructed vector based on a size and a direction

Details of MyScript

Open MyScript and examine the source code in the IDE. The instance variables are as follows.

```
// Toggle of what to draw
public bool DrawAxisFrame = false;
public bool DrawScaledVector = false;
public bool DrawUnitVector = false;
public bool DrawPositionVector = false;
public bool DrawVectorComponents = false;

// For defining Va and Vs (ScaledVector)
public GameObject P1 = null;    // Position P1
public GameObject P2 = null;    // Position P2
public float ScalingFactor = 0.8f;

// For defining Vp (PositionVector)
public GameObject SphereAtOrigin = null;    // Transparent sphere at origin
public float SphereRadius = 3.0f;

// For visualizing all vectors
private MyVector ShowVa;                    // Vector Va
private MyVector ShowVaScaled;              // Scaled Va
private MyVector ShowNorm;                  // Normalized Va
private MyVector ShowPositionVector;  // Position vector at the origin
```

All the public variables for MyScript have been discussed when analyzing the MainCamera's MyScript component. The four private variables of the MyVector data type are for visualizing the four vectors: $\vec{V}_a$, $\vec{V}_s$, $\hat{V}_a$, and $\vec{V}_p$ respectively. The Start() function for MyScript is listed as follows.

```
void Start(){
    Debug.Assert(P1 != null);    // Check for proper setup in the editor
    Debug.Assert(P2 != null);
    Debug.Assert(SphereAtOrigin != null);

    // To support visualizing the vectors
    ShowVa = new MyVector {
        VectorColor = Color.black };
    ShowNorm = new MyVector {
        VectorColor = new Color(0.9f, 0.9f, 0.9f)};
    ShowVaScaled = new MyVector {
        VectorColor = new Color(0.9f, 0.4f, 0.9f) };
    ShowPositionVector = new MyVector {
        VectorColor = new Color(0.4f, 0.9f, 0.9f),
        VectorAt = Vector3.zero       // Position Vector at the origin
     };
}
```

The Debug.Assert() calls ensure proper setup regarding referencing the appropriate game objects via the Inspector Window, while the MyVector variables are instantiated and initialized with the proper colors. The Update() function is listed as follows.

```
void Update()
{
```

Visualization on/off: show or hide to avoid clutering

Vector Va: Compute Va and setup the drawing for Va

```
    if (DrawScaledVector) …

    if (DrawUnitVector) …

    if (DrawPositionVector) …
}
```

The $\mathrm{Update}()$ function is logically structured into five steps: handling the drawing toggles, and then computing and showing $\vec{V}_a$, $\vec{V}_s$, $\hat{V}_a$, and $\vec{V}_p$ respectively. The details in each step are presented next in separate subsections. While reading the code, note the exact one-to-one match between the derived formula to compute each vector and the corresponding listed code. This is an important and elegant characteristic of vector-based game object behavior; the implementation often closely resemblances the underlying mathematic derivation.

Visualization on/off

The code in this region sets the game object's active state for displaying or hiding according to user's toggle settings. This code is listed as follows.

```
#region  Visualization on/off: show or hide to avoid clutering
AxisFrame.ShowAxisFrame = DrawAxisFrame;      // Draw or Hide Axis Frame
ShowVaScaled.DrawVector = DrawScaledVector; // Display or hide the vectors
ShowNorm.DrawVector = DrawUnitVector;
ShowVa.DrawVectorComponents = DrawVectorComponents;
ShowVaScaled.DrawVectorComponents = DrawVectorComponents;
ShowNorm.DrawVectorComponents = DrawVectorComponents;
ShowPositionVector.DrawVector = DrawPositionVector;
SphereAtOrigin.SetActive(DrawPositionVector);
#endregion
```

Vector Va

The code in this region computes $\vec{V}_a$ based on the current $\mathrm{P1}$ and $\mathrm{P2}$ positions and sets up the ShowVa variable for visualizing the vector. This code is listed as follows.

```
#region Vector Va: Compute Va and setup the drawing for Va
Vector3 vectorVa = P2.transform.localPosition - P1.transform.localPosition;

// Show the Va vector at P1
ShowVa.Direction = vectorVa;
ShowVa.Magnitude = vectorVa.magnitude;
ShowVa.VectorAt = P1.transform.localPosition;
#endregion
```

The variable $\mathrm{vectorVa}$ is $\vec{V}_a = P_2 - P_1$. The ShowVa variable receives the corresponding direction and size values from $\mathrm{vectorVa}$ and is set to display the vector at position $\mathrm{P1}$.

DrawScaledVector

When this toggle is set to true, $\vec{V}_s$ is computed and shown. The code to accomplish this is listed as follows.

```
if (DrawScaledVector) {
    Vector3 vectorVs = ScalingFactor * vectorVa;
    ShowVaScaled.Direction = vectorVs;
    ShowVaScaled.Magnitude = vectorVs.magnitude;
    ShowVaScaled.VectorAt = P1.transform.localPosition;
}
```

The variable vectorVs is $\vec{V}_s = ScalingFactor \times \vec{V}_a$. The ShowVaScaled is properly setup to display vectorVs at P1.

DrawUnitVector

When this toggle is set to true, $\hat{V}_a$ is computed and shown. The code to accomplish this is listed as follows.

```
if (DrawUnitVector) {
    Vector3 unitVa = (1.0f / vectorVa.magnitude) * vectorVa;  // scale Va by its inversed size
    // Vector3 dirVa = vectorVa.normalized;  // Another way to normalized Va
    ShowNorm.Direction = unitVa;
    ShowNorm.Magnitude = unitVa.magnitude;
    ShowNorm.VectorAt = P1.transform.localPosition;
}
```

The variable unitVa is $\hat{V}_a = \frac{1}{\|\vec{V}_a\|} \times \vec{V}_a$. Notice the alternative way commented out below this line of code, Vector3.normalized, to compute a unit vector.

Here you can observe a coding error, where, vectorVa.magnitude is used as the denominator in the normalization computation without first being verified that its value is not zero. Once again, a zero vector will have a length of zero and therefore cannot be normalized. In this case, the logic should check if vectorVa is equal to the zero vector and if so, simply skip the drawing of ShowNorm.

■ ***Note*** *In general, it is not advisable to compare computation results to floating point constants. For example, it is un-wise to attempt to detect the zero vector condition by performing:*

```
if (vectorVa.magnitude == 0.0f)
```

The chances of the results of the floating-point computation being exactly zero is almost non-existent. In this case, you should check for the condition of smaller than a "very small" number. The C# programming language defines the float.Epsilon *for this purpose. In this case, the condition to check for zero vector should be:*

```
if (vectorVa.magnitude < float.Epsilon)
    // vectorVa is, for all practical purposes, a zero vector
```

DrawPositionVector

When this toggle is set to true, $\vec{V}_p$ is computed and shown. The code to accomplish this is listed as follows.

```
if (DrawPositionVector) {
    Vector3 vectorVp = SphereRadius * vectorVa.normalized;
    ShowPositionVector.Direction = vectorVp;
    ShowPositionVector.Magnitude = vectorVp.magnitude;
    ShowPositionVector.VectorAt = SphereAtOrigin.transform.localPosition;

    // Set the radius of the sphere at the origin
    SphereAtOrigin.transform.localScale = new Vector3(2.0f*SphereRadius,
                                    2.0f * SphereRadius, 2.0f * SphereRadius);
}
```

The variable vectorVp is $\vec{V}_p = SphereRadius \times \hat{V}_a$. Note that in this case, $\hat{V}_a$ is computed based on the Unity Vector3.normalized utility. The last line of code scales the sphere by setting the Unity Transform.localScale. Notice that the scaling factor for the sphere is its diameter, or 2 times the radius. This is because localScale adjusts the scale of a sphere based on its diameter, not its radius.

Take Away from This Example

Note that the entire implementation for this example, the code in the Update() function that performs useful computation, is actually just four lines; one line for each of the vectors, $\vec{V}_a$, $\vec{V}_s$, $\hat{V}_a$, and $\vec{V}_p$ respectively. The rest of the code is there to support user interaction and to setup the four toggle variables for visualizing the vectors. This example shows that when working with vector-based logic, the code can be rather compact with the implementation closely resembling the actual math involved to compute such results.

Relevant mathematical concepts covered include:

- All scaled vectors are along exactly the same direction as their reference vector.
- The unit vector, or normalized vector, is a special case of the scaled vector; it is a vector scaled by the inverse of the size of its reference vector.
- The normalized vector, or unit vector, always has a length of one and does indeed uniquely and consistently represent the direction of vectors with different scaling factors.
- The zero vector cannot be normalized. Proper coding should include specific conditional checks before invoking the normalization computation.
- A vector can be defined based on a magnitude and a direction. An interesting implication of this fact is that any vector can be decomposed into a unit vector with a scale.

Unity Tools:

- Transform.localScale: to change the size of game objects
- Sphere primitive: the scale value is the diameter of the sphere

EXERCISES

Verify the Directions of vectorVa and vectorVp

Make sure that $\vec{V}_a$, $\vec{V}_s$, and $\hat{V}_a$, are in the exact same direction by setting ScalingFactor to a positive value. Next, verify the $\vec{V}_p$ vector is also in the same direction by moving P1 to the origin. Interestingly, you can also move the position of the SphereAtOrigin to P1 by changing the value of SphereAtOrigin.Transform.localPosition.

Properly Handle the Zero Vector

Implement the detection and handling of the zero vector condition to avoid the normalization process accordingly.

Work with Unit Vector and MyVector

A unit vector always has a size of 1 and can be a convenient reference for defining vectors of different lengths. For example, edit MyScript to display 5 different vectors with lengths of 1, 2, 3, 4, and 5 in the $\hat{V}_a$ direction. Display these vectors at the x-axis locations that correspond to their length, length 1 at $(1,0,0)$, length 2 at $(2,0,0)$, etc. The easiest solution to this problem would be to compute $\hat{V}_a$ and loop from 1 to 5, scaling each vector accordingly and working with MyVector to display the vectors at their proper positions.

Application of Vector: Velocity

When riding in a traveling car, you move at the speed and direction of that car. On a per-unit time basis, you will cover the "speed" amount of distance in the direction of the car. For example, during rush hours, a taxi traveling at 1.4 miles per hour towards the northeast will cover 1.4 miles in the northeast direction each hour. In this way, a velocity is speed in a specific direction, or, simply, a vector. Figure 4-9 illustrates the example of that taxi ride.

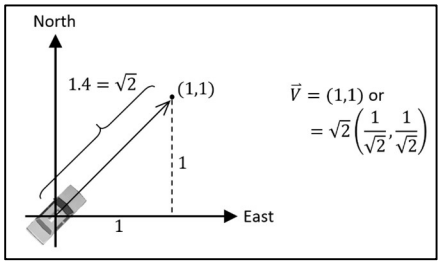

Figure 4-9. *Driving at* 1.4 *miles per hour towards the northeast*

As illustrated in Figure 4-9, the 1.4 miles per hour speed of the taxi describes the total distance covered per hour and is actually the magnitude of the vector. In this case, a velocity of

$$\vec{V}_t = (1,\ 1)\ miles/hou$$

Will, in an hour, cover a distance of,

$$\|\vec{V}_t\| = \sqrt{1^2 + 1^2} = \sqrt{2} \approx 1.4 \text{ miles}$$

and the traveling direction is indeed towards the northeast (assuming north is the positive y-direction and east is the positive x-direction). Notice in this description how the distance covered is separated from the direction of movement in the description of the taxi ride. When discussing velocities, it is important to identify the speed and the direction of travel. In terms of implementation, this means it is convenient to express a velocity, $\vec{V}_t$, as

$$\vec{V}_t = Spe \quad \times \hat{V}_t$$

In the case of Figure 4-9,

- Speed = 1.4
- $\hat{V}_t = (\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$

Recall that you have worked with vectors in this format in the `DrawPositionVector` portion of the previous example, `EX_4_2_VectorScaling`. Representing vectors in this way supports independent adjustments to the magnitude and the direction. In the context of velocity, this representation supports the independent adjustments to the speed (Figure 4-10) and the traveling direction (Figure 4-11).

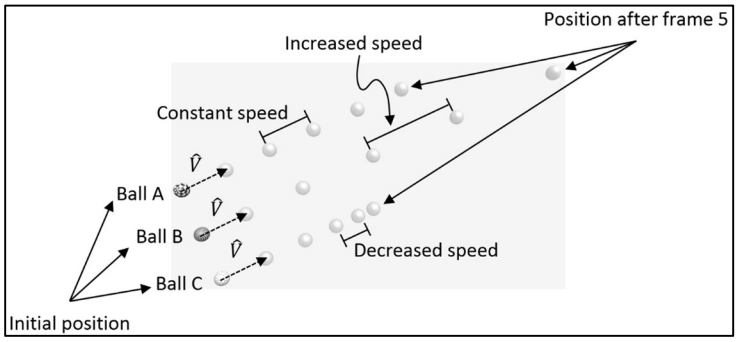

Figure 4-10. *Adjusting the speed while maintaining the direction of travel*

Figure 4-10 shows three balls, A, B, and C, traveling in the same direction, $\hat{V}$, at constant, increasing, and decreasing speeds respectively. Notice how the balls continue to travel parallel to each other but end up at very different locations along their parallel paths after a few updates.

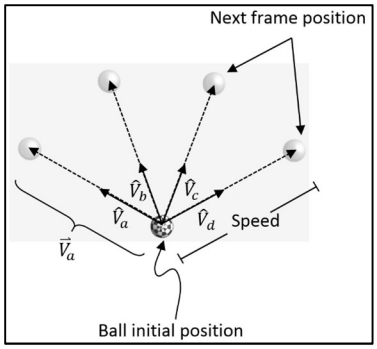

Figure 4-11. *Adjusting the direction of travel while maintaining a constant speed*

In contrast to Figure 4-10, Figure 4-11 shows how the traveling direction of an object can be adjusted without altering its speed. In this case, after subsequent updates, the objects would travel a constant distance from the original position but will end up at very different locations. In all cases, mathematically, the position of an object will change or "travel" by "following the velocity vector," $\vec{V}_t$. If,

P_{init}: Initial Position

then, at the end of the time unit, the object would travel " following the vector $\vec{V}_t$" and arrive at:

$$P_{final} = P_{init} + (\vec{V}_t \times elapsedTime)$$

This further illustrates the fact that velocity can be perfectly represented as a vector where the vector's magnitude is speed and direction is the direction of travel. This representation of velocity as a vector is convenient for game development and will be showcased in the next example.

The Velocity and Aiming Example

This example demonstrates the manipulation of object velocity and simple aiming functionality based on the vector concepts you have learned in the previous sections. The example allows you to adjust the speed, direction, and the traveling distance of an object separately. This example also allows you to examine the implementation of these factors. Figure 4-12 shows a screenshot of running the EX_4_3_VelocityAndAiming scene from the Chapter-4-Examples project.

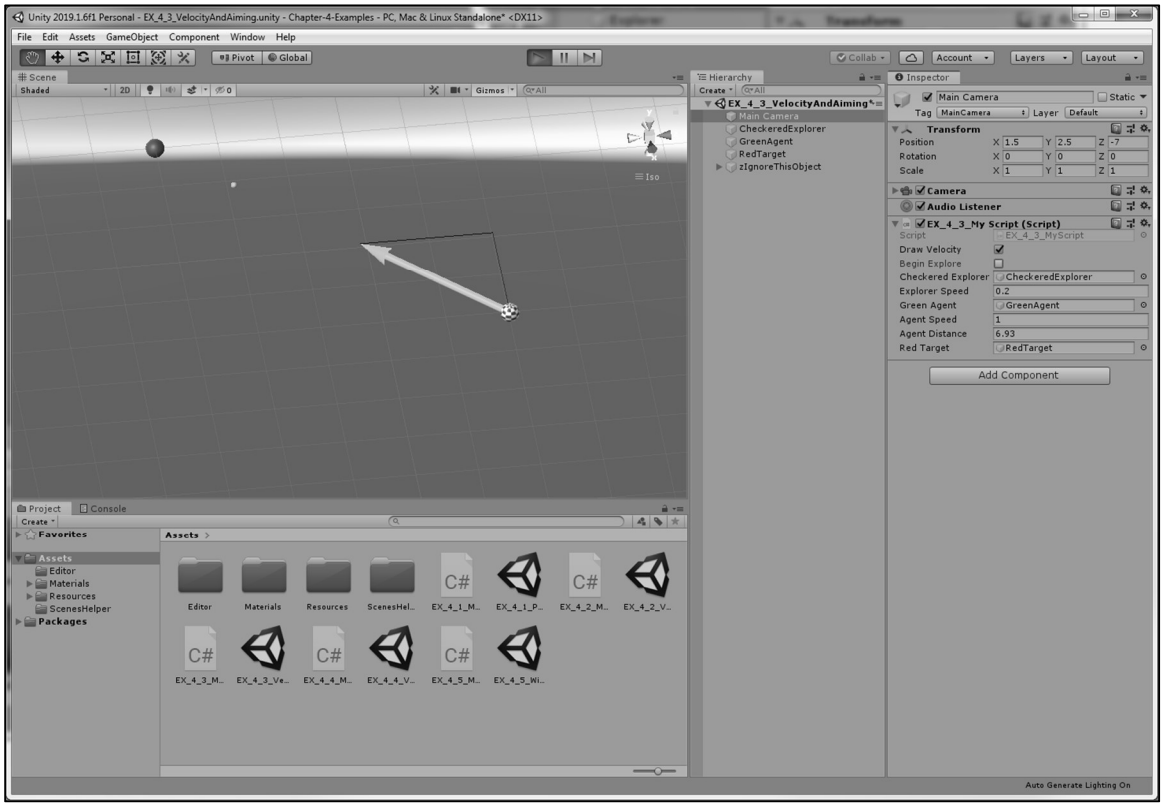

Figure 4-12. *Running the Velocity and Aiming example*

The goals of this example are for you to:

- Understand the distinction between speed and direction of a velocity
- Experience controlling a velocity by manipulating its speed and direction separately
- Examine a simple aiming behavior
- Examine the implementation of vector-based motion control

Examine the Scene

Take a look at the Example_4_3_VelocityAndAiming scene and observe the predefined game objects in the Hierarchy Window. In addition to the MainCamera, there are three objects in this scene: CheckeredExplorer, GreenAgent, and RedTarget. Select these objects in the Hierarchy Window to note that the CheckeredExplorer is the checkered sphere, the GreenAgent is the small green sphere, and the RedTarget is the red sphere. As in all previous examples, these sphere game objects represent positions where only their transform.localPosition are referenced. When the game begins to run, the CheckeredExplorer position will

move slowly towards the position of the RedTarget while continuously sending out the GreenAgent towards the RedTarget as well, but at a faster speed.

Analyze MainCamera MyScript Component

The MyScript component on MainCamera shows four sets of variables.

- Control Toggles: Toggles drawing on or off, or allows object movement
  - DrawVelocity: Show or hide the velocity of the CheckeredExplorer
  - BeginExplore: Enable the movement of the CheckeredExplorer and the GreenAgent
- Support for the CheckeredExplorer:
  - CheckeredExplorer: The reference to the CheckeredExplorer game object
  - ExplorerSpeed: The traveling speed of the CheckeredExplorer
- Support for the GreenAgent:
  - GreenAgent: The reference to the GreenAgent game object
  - AgentSpeed: The traveling speed of the GreenAgent
  - AgentDistance: The distance that the GreenAgent should travel before returning to base and restarting the exploration
- Support for the RedTarget:
  - RedTarget: The reference to the RedTarget game object

The velocity direction for both the CheckeredExplorer and the GreenAgent are implicitly defined by their relative position to the RedTarget because that is the target position that both the CheckeredExplorer and GreenAgent are moving towards.

Interact with the Example

Click on the Play Button to run the example. Initially the BeginExplore toggle is set to false and there will thus be no movement in the scene. The green vector you observe extending from the CheckeredExplorer represents the velocity of the CheckeredExplorer object. Since you know the vector from the CheckeredExplorer to the RedTarget is, $\vec{V}_{ET}$, then assuming the CheckeredExplorer object is located at $P_{Explorer}$ and the RedTarget object is located at P_{Target}, then

$$\vec{V}_{ET} = P_{Target} - P_{Explorer}$$

Both the CheckeredExplorer and the GreenAgent will be traveling, with their respective speeds of ExplorerSpeed and AgentSpeed, towards the RedTarget. The velocities of these two objects, $\vec{V}_{Explorer}$ and $\vec{V}_{Agent}$, are defined as,

$$\vec{V}_{Explorer} = ExplorerSpeed \times \hat{V}_{ET}$$

$$\vec{V}_{Agent} = AgentSpeed \times \hat{V}_{ET}$$

Note that the two velocities are in the same direction, unit vector $\hat{V}_{ET}$, but with different magnitudes, or speeds. Additionally, in both cases, the speeds are under user control and yet the velocity direction is implicitly defined by the RedTarget position.

The green vector you observed represents $\vec{V}_{Explorer}$. Now, adjust ExplorerSpeed in the MyScript component of the MainCamera object and notice the green vector's length change accordingly. Since this vector's length is determined by ExplorerSpeed you can expect the CheckeredExplorer object to move quicker when the green vector is long and slower when it is short. Now, enable the BeginExplore toggle and observe the following.

- The CheckeredExplorer follows slowly behind the repeating and faster traveling GreenAgent. You can adjust the speed of the CheckeredExplorer via the ExplorerSpeed variable and observe, as mentioned previously, that the speed corresponds to the length of the green vector.
- The GreenAgent continuously repeats the quick motion of traveling from the CheckeredExplorer towards the RedTarget. Try adjusting the AgentSpeed variable and observe how the GreenAgent's speed changes.
- The AgentDistance variable dictates how far the GreenAgent can travel from the CheckeredExplorer before its position is reset and it starts over. If $\vec{V}_{EA}$ is the vector from GreenAgent to the CheckeredExplorer then,

  $$\vec{V}_{EA} = P_{Agent} - P_{Explorer}$$

  the current distance between the two is simply the magnitude of this vector, $\|\vec{V}_{EA}\|$. Now, try altering the value of AgentDistance to observe the green sphere traveling that corresponding distance from the checkered sphere before restarting.
- The RedTarget is stationary but you can manipulate its position via its transform components, and since,

  $$\vec{V}_{ET} = P_{Target} - P_{Explorer}$$

  when the RedTarget position, P_{Target}, is changed, the vector $\vec{V}_{ET}$ is updated accordingly. The velocity direction, $\hat{V}_{ET}$, of both the CheckeredExplorer and GreenAgent are also updated. In this way, both of these objects are always aiming at and moving towards the RedTarget.

Notice that when the CheckeredExplorer arrives at a location that is very close to the RedTarget, the green vector that represents its velocity will rapidly flip back and forth. As you will find out when analyzing the implementation, there is no logic involved for checking the stop condition of the CheckeredExplorer. Therefore, you are observing the CheckeredExplorer continuously moving pass the RedTarget, flipping its velocity, and then moving pass the RedTarget again. The logic to stop the CheckeredExplorer's motion is left as an exercise at the end of this example.

Details of MyScript

Open MyScript and examine the source code in the IDE. The instance variables are as follows.

```
// Drawing control
public bool DrawVelocity = true;
public bool BeginExplore = false;

public GameObject CheckeredExplorer = null;// The CheckeredExplorer
public float ExplorerSpeed = 0.05f;          // units per second

public GameObject GreenAgent = null;          // The GreenAgent
public float AgentSpeed = 1.0f;               // units per second
public float AgentDistance = 3.0f;            // Distance to explore before returning to base
```

```
public GameObject RedTarget = null;          // RedTarget

private MyVector ShowVelocity = null;        // For visualizing Explorer velocity

private const float kSpeedScaleForDrawing = 15f;
```

All public variable for MyScript have been discussed when analyzing the MainCamera's MyScript component. The private variable ShowVelocity is to support the visualization of the CheckeredExplorer velocity where the kSpeedScaleForDrawing is a constant value meant to scale this vector such that it is visible. The Start() function for MyScript is listed as follows.

```
void Start() {
    Debug.Assert(CheckeredExplorer != null);
    Debug.Assert(RedTarget != null);
    Debug.Assert(GreenAgent != null);

    ShowVelocity =  new MyVector() {
        VectorColor = Color.green;
    }

    // initially Agent is resting insdie the Explorer
    GreenAgent.transform.localPosition =
        CheckeredExplorer.transform.localPosition;
}
```

As in all previous examples, the Debug.Assert() calls ensure proper setup regarding referencing the appropriate game objects via the Inspector Window, while the ShowVelocity variable is properly instantiated. Lastly, the initial position of GreenAgent is set to that of the CheckeredExplorer. The Update() function is listed as follows.

```
void Update() {
    Vector3 vET = RedTarget.transform.localPosition - CheckeredExplorer.transform.localPosition;

    ShowVelocity.VectorAt = CheckeredExplorer.transform.localPosition;
    ShowVelocity.Magnitude = ExplorerSpeed * kSpeedScaleForDrawing;
    ShowVelocity.Direction = vET;
    ShowVelocity.DrawVector = DrawVelocity;

    if (BeginExplore) {
        float dToTarget = vET.magnitude;  // Distance to target
        if (dToTarget < float.Epsilon)
            return; // Avoid normlaizing a zero vector
        Vector3 vETn = vET.normalized;

        Process the Explorer (checkered sphere)

        Process the Agent (small green sphere)
    }
}
```

The first line of the Update() function computes $\vec{V}_{ET} = P_{Target} - P_{Explorer}$, and the next four lines set up the ShowVelocity variable for visualizing the CheckeredExplorer's velocity as a vector with its tail located at the position of CheckeredExplorer. Note that because of CheckeredExplorer's slow speed (ExplorerSpeed's value), the ShowVelocity.Magnitude is scaled by kSpeedScaleForDrawing in order to properly visualize the vector.

When BeginExplore is enabled, the magnitude of $\vec{V}_{ET}$, or $\|\vec{V}_{ET}\|$, is checked to avoid the normalization of a zero vector. Next, $\hat{V}_{ET}$ is computed and stored in the variable, vETn. The two regions that process the CheckeredExplorer and the GreenAgent are explained in the following subsections.

Process the Explorer

The code in this region, listed as follows, computes the velocity of the explorer,

$$\vec{V}_{Explorer} = ExplorerSpeed \times \hat{V}_{ET}$$

and updates CheckeredExplorer.transform.localPosition accordingly.

```
#region Process the Explorer (checkered sphere)
Vector3 explorerVelocity = ExplorerSpeed * vETn; // define velocity
CheckeredExplorer.transform.localPosition += explorerVelocity * Time.deltaTime; // update position
#endregion
```

Remember that displacement, or distance, is velocity traveled over time, or, $\text{Velocity} \times \text{elapsedTime}$. In Unity, the per-update elapsed time is recorded in the Time.deltaTime property. The very last line in this region computes the total displacement over time and updates CheckeredExplorer's position with the computed displacement, ensuring smooth movement.

Process the Agent

As illustrated in the following code, similar to processing the movement of CheckeredExplorer, the first two lines of code compute the velocity of the agent,

$$\vec{V}_{Agent} = AgentSpeed \times \hat{V}_{ET}$$

and update GreenAgent.transform.localPosition accordingly. Note that, as mentioned previously, because $\vec{V}_{Explorer}$ and $\vec{V}_{Agent}$ are both computed based on scaling the same unit vector, the CheckeredExplorer and GreenAgent are traveling in the exact same direction, $\hat{V}_{ET}$, with different speeds, ExplorerSpeed and AgentSpeed.

```
#region Process the Agent (small green sphere)
Vector3 agentVelocity = AgentSpeed * vETn; // define velocity
GreenAgent.transform.localPosition += agentVelocity * Time.deltaTime; ;    // update position
Vector3 vEA = GreenAgent.transform.localPosition - CheckeredExplorer.transform.localPosition;
if (vEA.magnitude > AgentDistance)
    GreenAgent.transform.localPosition = CheckeredExplorer.transform.localPosition;
#endregion
```

The last three lines of code compute the vector between the explorer and the agent,

$$\vec{V}_{EA} = P_{Agent} - P_{Explorer}$$

compares the magnitude of this vector, $\|\vec{V}_{EA}\|$, to the user specified AgentDistance, and then resets the agent's position when it is too far away from the explorer, or when $\|\vec{V}_{EA}\| > AgentDistance$.

Take Away from This Example

This example demonstrates the application of vector concepts learned in modeling the simple object behaviors of aiming at and moving towards a target position. You have observed that the velocity of objects can be described by scaling a unit vector with speed, and that, velocities computed based on the same unit vector will move objects in exactly the same direction. Lastly, you have experienced once again that the distance between two objects can be easily computed as the magnitude of the vector defined between these two objects.

Relevant mathematical concepts covered include:

- The velocity of an object can be represented by a vector
- A velocity can be composed by scaling a direction, or unit vector, with speed
- The distance between two objects is the magnitude of the vector that is defined by those two objects

EXERCISES

Stop the CheckeredExplorer When It Reaches the RedTarget

Recall that the motion of CheckeredExplorer never terminates and that it tends to overshoot the RedTarget followed by turning around and overshooting it again. This cycle continues, causing the CheckeredExplorer to swing back and forth around the RedTarget. Modify MyScript to define a bounding box around the RedTarget and stop the CheckeredExplorer when it is inside the bounding box. Notice that in this case, it is actually easier and more accurate to treat the RedTarget as a bounding sphere and to stop the motion of the CheckeredExplorer when it is inside the bounds of the sphere.

Reset the GreenAgent When It Reaches the RedTarget

Run the game and increase the AgentDistance to some large value, e.g., 15. Now set BeginExplore to true and observe how the GreenAgent passes through the RedTarget and continues to move forward until its position is more than 15 units from the CheckeredExplorer, in which case it finally resets. With the bound you defined in the previous exercise, modify MyScript to reset the GreenAgent's position as soon as it is inside the RedTarget's bounds.

Invert the GreenAgent's Velocity Direction

Modify MyScript such that when the GreenAgent is too far away from the CheckeredExplorer, instead of resetting the position, the GreenAgent would simply move towards the CheckeredExplorer as though it is now the target. In this way, the GreenAgent would move continuously between the CheckeredExplorer and the RedTarget. This example allows you to gain experience with reversing the direction of a given vector.

Vector Algebra: Addition and Subtraction

Although it has not yet been formally defined, based on observing the relative positions in the Cartesian Coordinate System, you have worked with vector addition and subtraction for quite a while now. For example, you have learned that the statement "position P_1 can be reached by following a vector $\vec{V}_1$ at position P_0" is expressed mathematically as,

$$P_1 = P_0 + \vec{V}_1$$

In this case, by interpreting P_0 and P_1 as position vectors, the "+" operator has two vector operands and produces a position vector as the result of the operation. You have also learned that the statement "the vector $\vec{V}_1$ is a vector with its tail at position P_0 and head at position P_1" is expressed mathematically as,

$$\vec{V}_1 = P_1 - P_0$$

Once again, with P_0 and P_1 interpreted as position vectors, the "−" operation also has two vector operands and produces a vector as the result of the operation.

Rules of Vector Addition and Subtraction

You have learned and experienced that in both vector addition and subtraction, the resulting vectors are simply the addition and subtraction of the corresponding x-, y-, and z-component values. These observations are summarized in Table 4-1.

Table 4-1. *Vector addition and subtraction*

Operation	Operand 1	Operand 2	Result
+: Addition	$\vec{V}_1 = (x_1, y_1, z_1)$	$\vec{V}_2 = (x_2, y_2, z_2)$	$\vec{V}_1 + \vec{V}_2 = (x_1 + x_2, y_1 + y_2, z_1 + z_2)$
−: Subtraction	$\vec{V}_1 = (x_1, y_1, z_1)$	$\vec{V}_2 = (x_2, y_2, z_2)$	$\vec{V}_1 - \vec{V}_2 = (x_1 - x_2, y_1 - y_2, z_1 - z_2)$

Note that the given definition in Table 4-1 states that following is always true,

$$\vec{V} + \vec{V} = 2\vec{V}$$

$$\vec{V} - \vec{V} = ZeroVector$$

Because the operators add and subtract the corresponding coordinate component values, the familiar floating-point arithmetic addition and subtraction properties are obeyed. The properties of commutative, associative, and distributive with a floating-point scaling factor s, are summarized in Table 4-2.

Table 4-2. *Properties of vector addition and subtraction*

Properties	Vector Addition	Vector Subtraction
Commutative	$\vec{V}_1 + \vec{V}_2 = \vec{V}_2 + \vec{V}_1$	$\vec{V}_1 - \vec{V}_2 \neq \vec{V}_2 - \vec{V}_1$ [**not a property**]
Associative	$(\vec{V}_1 + \vec{V}_2) + \vec{V}_3 = \vec{V}_1 + (\vec{V}_2 + \vec{V}_3)$	$(\vec{V}_1 - \vec{V}_2) - \vec{V}_3 = \vec{V}_1 - (\vec{V}_2 - \vec{V}_3)$
Distributive	$s(\vec{V}_1 + \vec{V}_2) = s\vec{V}_1 + s\vec{V}_2$	$s(\vec{V}_1 - \vec{V}_2) = s\vec{V}_1 - s\vec{V}_2$

As illustrated in the first-row, right column of Table 4-2, just as with floating-point subtraction, vector subtraction is not commutative. In fact, similar to floating-point subtraction, vector subtraction is anti-commutative, or,

$$\vec{V}_1 - \vec{V}_2 = -1 \times (\vec{V}_2 - \vec{V}_1) = -\vec{V}_2 + \vec{V}_1$$
$$= \vec{V}_1 - \vec{V}_2$$

Addition and Subtraction with the Zero Vector

As in the case of floating-point arithmetic, vector addition and subtraction with the zero vector behave as expected.

$$\vec{V}_1 + ZeroVector = ZeroVector + \vec{V}_1 = \vec{V}_1$$
$$\vec{V}_1 - ZeroVector = \vec{V}_1$$
$$ZeroVector - \vec{V}_1 = -\vec{V}_1$$

Vectors in an Equation

Vectors behave just like floating-point values in an equation. For example, if

$$\vec{V}_3 = \vec{V}_1 + \vec{V}_2,$$

then adding a $-\vec{V}_2$ to both sides of the equation:

$$\vec{V}_3 + (-\vec{V}_2) = \vec{V}_1 + \vec{V}_2 + (-\vec{V}_2)$$
$$\vec{V}_3 - \vec{V}_2 = \vec{V}_1$$
$$\vec{V}_1 = \vec{V}_3 - \vec{V}_2.$$

This little example helps demonstrate that vector algebra obeys the basic algebraic equation rule that a term can be moved across the equality by flipping its sign.

Geometric Interpretation of Vector Addition and Subtraction

Fortunately, there are intuitive diagrammatic interpretations for the essential rules of vector addition and subtraction. Please refer to Figure 4-13, where vectors $\vec{V}_1$ and $\vec{V}_2$ are defined by the three given positions, P_0, P_1, and P_2. These two vectors are defined as,

$$\vec{V}_1 = P_1 - P_0$$

$$\vec{V}_2 = P_2 - P_1$$

Figure 4-13 shows vector $\vec{V}_1$ with its tail at P_0, and vector $\vec{V}_2$ with its tail at P_1.

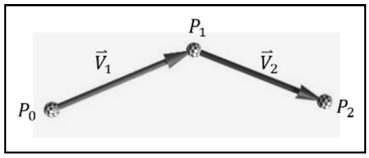

Figure 4-13. *Two vectors defined by three positions*

Vector Addition

Figure 4-14 shows the result of vector addition geometrically. Notice that the result of adding the two vectors,

$$\vec{V}_{sum} = \vec{V}_1 + \vec{V}_2$$

is a vector with its tail located at the tail of $\vec{V}_1$, P_0, and its head located at the head of $\vec{V}_2$, P_2. This can be interpreted geometrically as, $\vec{V}_{sum}$ is the combined results of "following $\vec{V}_1$ then $\vec{V}_2$." Except, that in case this, instead of following the two vectors sequentially, the summed vector, $\vec{V}_{sum}$, will take you directly from the beginning to the end along the shortest path. This observation is true in general, the result of summing vectors is always a vector that combines the results of following all of the operand vectors sequentially and is then the shortest path from the beginning location to the final destination location.

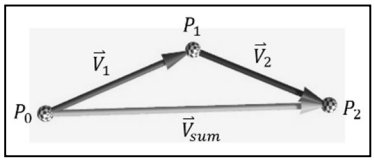

Figure 4-14. *Vector addition*

Commutative Property of Vector Addition

Figure 4-15 illustrates the commutative property of vector addition,

$$\vec{V}_{sum} = \vec{V}_1 + \vec{V}_2 = \vec{V}_2 + \vec{V}_1$$

Note the difference in the order of operations, the top half of Figure 4-14 applies $\vec{V}_1$ at P_0 followed by applying $\vec{V}_2$ at the head of $\vec{V}_1$, while the latter applies $\vec{V}_2$ at P_0 followed by applying $\vec{V}_1$ at the head of $\vec{V}_2$. Observe that in both cases, the result is identical, $\vec{V}_{sum}$ has its tail located at P_0 and its head at P_2.

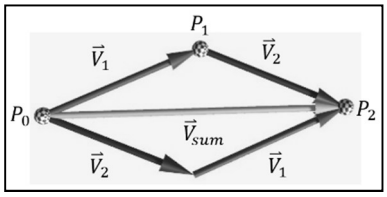

Figure 4-15. *The commutative property of vector addition*

Figure 4-14 shows that, geometrically, vector addition depicts a triangle where the first two edges are the operands and the third is the resulting sum. In Figure 4-15, the two $\vec{V}_1$ are of the same length and are parallel, and so are the two $\vec{V}_2$ vectors. For this reason, the depiction in Figure 4-15 is a parallelogram. These observations are true in general, that vector addition and the commutative property always depict a triangle and parallelogram respectively. Though these observations do not result in direct applications in video games, they provide insights into relationships between different fields of mathematics, in this case, linear algebra and geometry.

Vector Subtraction

Figure 4-16 shows the result of vector subtraction geometrically. The two vectors with tails at position P_1 are $\vec{V}_2$ and a scaling of $\vec{V}_2$ by a factor of -1 resulting in $-\vec{V}_2$, or $\vec{V}_{n2}$, a vector with same length in the opposite direction to $\vec{V}_2$. This figure shows that subtracting a vector is essentially the same as using the opposite direction of that vector in a vector addition. In this case, $\vec{V}_1 - \vec{V}_2$, can be understood as travel along $\vec{V}_1$, followed by traveling along the opposite direction of $\vec{V}_2$. This interpretation can be verified mathematically as follows. Notice that just as floating-point algebra, the subtraction of the two vectors,

$$\vec{V}_{sub} = \vec{V}_1 - \vec{V}_2$$

can be written as an addition,

$$\vec{V}_{sub} = \vec{V}_1 + \vec{V}_{n2}$$

where

$$\vec{V}_{n2} = -\vec{V}_2$$

or simply

$$\vec{V}_{sub} = \vec{V}_1 - \vec{V}_2 = \vec{V}_1 + (-\vec{V}_2)$$

Notice the perfect correspondence between the expression, $\vec{V}_1 + (-\vec{V}_2)$, and the description, " travel along $\vec{V}_1$, followed by traveling along the opposite direction of $\vec{V}_2$."

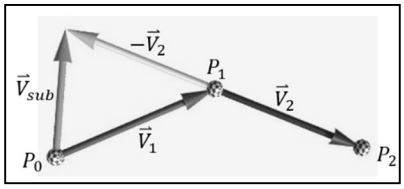

Figure 4-16. *Vector Subtraction*

The Vector Add and Sub Example

This example demonstrates the results of and allows you to interact with the vector addition and subtraction operations. This example also serves as a review and reaffirmation that vectors can be located at any position as their definition does not link them to a specific position. Figure 4-17 shows a screenshot of running the EX_4_4_VectorAddandSub example from the Chapter-4-Examples project.

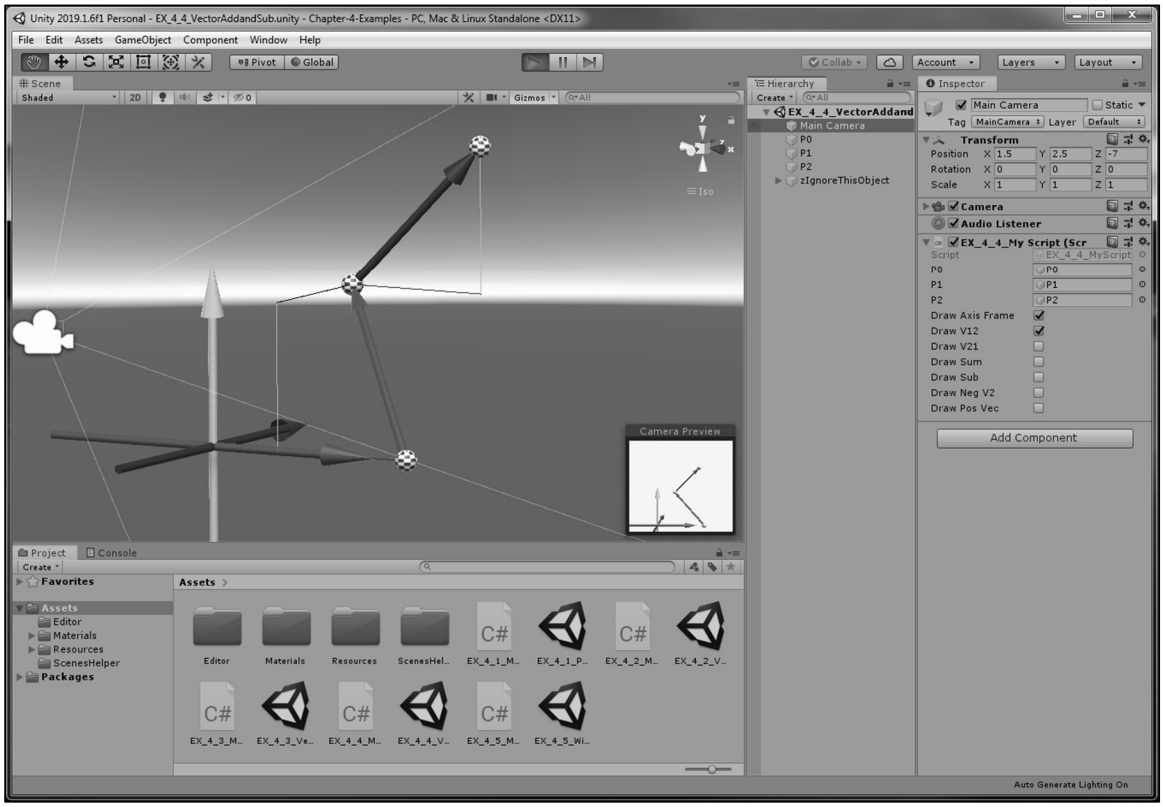

Figure 4-17. *Running the Vector Add and Sub example*

The goals of this example are for you to:

- Examine and gain understanding of vector addition and subtraction
- Understand that vector subtraction is simply vector addition with a negative vector as the second operand
- Review that all vectors are defined independent of any position

Examine the Scene

Look at the Example_4_4_VectorAddandSub scene and observe the predefined game objects in the Hierarchy Window. In addition to MainCamera, there are three objects in this scene: P0, P1, and P2. Each of these objects reference one of the spheres in the scene which in turn represent a position in the Cartesian Coordinate System. In this example you can manipulate these three positions to define two vectors, where the results of adding and subtracting these two vectors are shown at those positions and at the origin as position vectors.

Analyze MainCamera MyScript Component

The MyScript component on MainCamera shows two sets of variables.

- The three positions:
  - P0: The reference to the P0 game object.
  - P1: The reference to the P1 game object.
  - P2: The reference to the P2 game object.

  The transform.localPosition of these objects will provide the positions defining the two vectors:

  $\vec{V}_1 = P_1 - P_0$

  $\vec{V}_2 = P_2 - P_1$
- Draw control: There are seven toggles for showing or hiding the following.
  - DrawAxisFrame: Show or hide the axis frame; the axis frame serves as a reference for showing position vectors.
  - DrawV12: Show or hide vector $\vec{V}_1$ at position P_0, and $\vec{V}_2$ at the head of $\vec{V}_1$. This is convenient for examining $\vec{V}_1 + \vec{V}_2$.
  - DrawV21: Show or hide vector $\vec{V}_2$ at position P_0, and $\vec{V}_1$ at the head of $\vec{V}_2$. This is convenient for examining $\vec{V}_2 + \vec{V}_1$.
  - DrawSum: Show or hide the vectors $\vec{V}_{sum} = \vec{V}_1 + \vec{V}_2$ and $\vec{V}_{sum} = \vec{V}_2 + \vec{V}_1$.
  - DrawSub: Show or hide the vector $\vec{V}_{sub} = \vec{V}_1 - \vec{V}_2$.
  - DrawNegV2: Show or hide the vector $-\vec{V}_2$.
  - DrawPosVec: Show or hide currently visible vector(s) as position vector(s).

The purpose of this example is for you to manipulate the P0, P1, and P2 positions and toggle each of the above drawing options to closely examine each of the corresponding vectors.

Interact with the Example

Click on the Play Button to run the example. Initially, both DrawAxisFrame and DrawV12 are enabled so you should observe the axis frame and the two vectors $\vec{V}_1$ (in red) and $\vec{V}_2$ (in blue) connecting the checkered spheres P0, P1, and P2. Now, enable DrawPosVec to observe vectors $\vec{V}_1$ and $\vec{V}_2$ drawn at the origin as position vectors. At any point in the following interaction, feel free to toggle on DrawAxisFrame for referencing. For now, please toggle it off to avoid cluttering up the scene.

Vector Addition and the Commutative Property

With DrawPosVec on, switch on both DrawV12 and DrawV21 toggles to witness these two sets of vectors being drawn. Select and manipulate position P1 to observe how the two sets of vectors change. Now toggle DrawSum on and continue with the manipulation of position P1. Observe that since $\vec{V}_{sum} = \vec{V}_1 + \vec{V}_2 = \vec{V}_2 + \vec{V}_1$ is a vector from P0 to P2, changing P1 has absolutely no effect on $\vec{V}_{sum}$. Next, select and manipulate P0 to observe how the red $\vec{V}_1$ and green $\vec{V}_{sum}$ vectors change together while the blue $\vec{V}_2$ remains constant. Repeat the manipulation for P2 and observe $\vec{V}_2$ and $\vec{V}_{sum}$ altering while $\vec{V}_1$ remains constant.

Through these interactions, you have verified that vector addition is indeed accumulating the results of individual operands and that the operation does indeed obey the commutative property. You were also reminded, through turning on the DrawPosVec toggle, that vectors are independent of positions as all three vectors were identical to their corresponding color partner except for their tail location.

Vector Subtraction

Reset all toggles to off and switch on DrawPosVec, DrawV12, and DrawNegV2. You should observe three sets of vectors, $\vec{V}_1$ (in red), $\vec{V}_2$ (in blue), and $-\vec{V}_2$ (in yellow). Manipulate the Scene View camera to observe that the yellow vectors are indeed the same length and in opposite directions as the blue vectors. Select and manipulate P1 to observe the three sets of vectors changing in sync. If you manipulate P2, it will only affect $\vec{V}_2$ (in blue), and $-\vec{V}_2$ (in yellow) vectors. Now switch on the DrawSub toggle to observe the gray $\vec{V}_{sub}$ vector as the sum of the red and yellow vector, $\vec{V}_{sub} = \vec{V}_1 + (-\vec{V}_2)$.

Through these interactions, you have verified that vector subtraction is indeed the same as vector addition with the second operand being negated. In fact, every operand after the first operand, if originally being subtracted, can instead be added after its been negated, just like with floating-point arithmetic.

Position Vector

With DrawPosVec toggle on, every computed vector is displayed at the origin as a position vector. For example, while $\vec{V}_{sum}$ was computed by $\vec{V}_1 + \vec{V}_2$ and the geometric depiction suggests that $\vec{V}_{sum}$ must always have its tail at P0, this is not the case. Once again, a vector is a length and a direction, this definition holds true independent of any specific position, even when a position is used initially to define that vector.

Details of MyScript

Open MyScript and examine the source code in the IDE. The instance variables are as follows.

```
public GameObject P0, P1, P2;                 // V1=P1-P0 and V2=P2-p1

// For visualizing the vectors
private MyVector ShowV1atP0, ShowV2atV1,      // Show V1 at P0 and V2 at head of V1
                 ShowV2atP0, ShowV1atV2,      // Show V2 at P0 and V1 at head of V2
                 ShowSumV12, ShowSumV21,  // V1+V2, and V2+V1
                 ShowSubV12,                  // V1-V2
                 ShowNegV2;                   // -V2
private MyVector PosV1, PosV2, PosSum, PosSub, PosNegV2; // Show as position vectors

// Toggles for drawing/hiding corresponding vectors
public bool DrawAxisFrame = true;
public bool DrawV12 = false, DrawV21 = false;
public bool DrawSum = false;
public bool DrawSub = false, DrawNegV2 = false;
```

```
public bool DrawPosVec = false;
```

All public variables for MyScript have been discussed when analyzing the MainCamera's MyScript component. The large number of private MyVector variables are for visualizing the corresponding vectors. The Start() function for MyScript is listed as follows.

```
void Start() {
    Debug.Assert(P0 != null);
    Debug.Assert(P1 != null);
    Debug.Assert(P2 != null);

    ShowV1atP0 = new MyVector() { // Show V1 vectors
        VectorColor = Color.red      };
    ShowV1atV2 = new MyVector() {
        VectorColor = Color.red      };
    PosV1 = new MyVector()       { // Show V1 as position vector
        VectorAt = Vector3.zero,  // always show at the origin
        VectorColor = Color.red      };

    ShowV2atP0 = new MyVector() { // Show V2 vectors
       VectorColor = Color.blue      };
    ShowV2atV1 = new MyVector() {
        VectorColor = Color.blue     };
    PosV2 = new MyVector()       { // Show V2 as position vector
        VectorAt = Vector3.zero,
        VectorColor = Color.blue     };

    ShowSumV12 = new MyVector() { // Show V1 + V2
        VectorColor = Color.green     };
    ShowSumV21 = new MyVector() { // Show V2 + V1
        VectorColor = Color.green     };
    PosSum = new MyVector()       { // Show sum as position vector
        VectorAt = Vector3.zero,
        VectorColor = Color.green     };

    ShowSubV12 = new MyVector() { // Show V1 - V2
        VectorColor = Color.gray      };
    PosSub = new MyVector()       { // Show as position vector
        VectorAt = Vector3.zero,
        VectorColor = Color.gray      };

    ShowNegV2 = new MyVector()   { // Show -V2
        VectorColor = new Color(0.9f, 0.9f, 0.2f, 1.0f) };
    PosNegV2 = new MyVector()     {
        VectorAt = Vector3.zero,
        VectorColor = new Color(0.9f, 0.9f, 0.2f, 1.0f) };
}
```

As in all previous examples, the Debug.Assert() calls ensure proper setup regarding referencing the appropriate game objects via the Inspector Window. The rest of the Start() function instantiates the many MyVector variables for visualization, setting their colors and display positions. The Update() function is listed as follows.

```
void Update() {
    Vector3 V1 = P1.transform.localPosition - P0.transform.localPosition;
    Vector3 V2 = P2.transform.localPosition - P1.transform.localPosition;
    Vector3 sumV12 = V1 + V2;
    Vector3 sumV21 = V2 + V1;
```

```
    Vector3 negV2 = -V2;
    Vector3 subV12 = V1 + negV2;
```

Draw control: switch on/off what to show

V1: show V1 at P0 and head of V2

V2: show V2 at P0 and head of V1

Sum: show V1+V2 and V2+V1

Sub: show V1-V2

Negative vector: show -V2

```
}
```

The $\mathrm{Update}()$ function first computes all the relevant vectors,

$$\vec{V}_1 = P_1 - P_0$$
$$\vec{V}_2 = P_2 - P_1$$
$$\vec{V}_{sum12} = \vec{V}_1 + \vec{V}_2$$
$$\vec{V}_{sum21} = \vec{V}_2 + \vec{V}_1$$
$$\vec{V}_{n2} = -\vec{V}_2$$
$$\vec{V}_{sub12} = \vec{V}_1 - \vec{V}_2$$

Then it sets up the corresponding $\mathrm{MyVector}$ variables for display based upon their values and if their toggle switch is true. The details of this visualization code are independent of the vector operations being studied and are therefore not discussed here. You can explore the code in these regions at your own leisure.

Take Away from This Example

This example demonstrates the details of vector addition and subtraction where the commutative property of vector addition is verified, and vector subtraction is presented as vector addition with a negated vector. Equally important is the review of a vector's independence of positions.

Relevant mathematical concepts covered include:

- Vector addition results in a vector that accumulates the operand vectors
- Vector addition is indeed commutative
- Vector subtraction is simply an addition with the second operand being negated
- Reviewed that vectors are independent of any particular position

EXERCISES

Verify Vector Addition Accumulates in General

Modify the scene and MyScript to include a fourth position, P3, and a vector, $\vec{V}_3$.

$$\vec{V}_3 = P_3 - P_2$$

Now, define $\vec{V}_{sum}$,

$$\vec{V}_{sum} = \vec{V}_1 + \vec{V}_2 + \vec{V}_3$$

Verify that if the tail of $\vec{V}_{sum}$ is located at P0 then its head will be located at P3.

Verify the Associative Property of Addition and Subtraction

With the fourth position, P3, and vector $\vec{V}_3$, verify

$$(\vec{V}_1 + \vec{V}_2) + \vec{V}_3 = \vec{V}_1 + (\vec{V}_2 + \vec{V}_3)$$

and

$$(\vec{V}_1 - \vec{V}_2) - \vec{V}_3 = \vec{V}_1 - (\vec{V}_2 - \vec{V}_3)$$

by computing and displaying each as a different MyVector object.

Application of Vector Algebra

Although seldom applied directly, the indirect applications of vector algebra in video games are ubiquitous and vital. For example, you have already experienced working with vector subtraction in defining a vector between two positions for distance computation, and vector addition in computing movements when applying a velocity to an object.

A straightforward application of vector addition is in simulating velocity under a constant external factor, e.g., an airplane flying or a ship sailing under a constant wind condition. Please refer to Figure 4-18 where a traveling ball is progressing towards a target with a velocity of $\vec{V}_T$. Under the window condition, $\vec{V}_{wind}$, the effective velocity experienced by the ball then becomes $\vec{V}_A$,

$$\vec{V}_A = \vec{V}_T + \vec{V}_{wind}$$

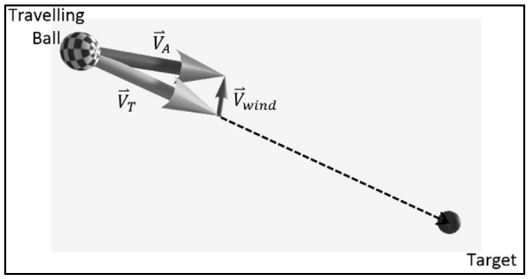

Figure 4-18. *Traveling under constant wind condition*

With your knowledge of vectors and vector addition, this wind condition is straightforward to simulate and is examined in the next example.

The Windy Condition Example

This example uses vector addition to simulate an object traveling under a constant wind condition. The example allows you to adjust all the parameters of this simulation, including the speed of the traveling object and the wind, the direction of the wind, and if the wind condition should affect the traveling object. Figure 4-19 shows a screenshot of running the EX_4_5_WindyCondition example from the Chapter-4-Examples project.

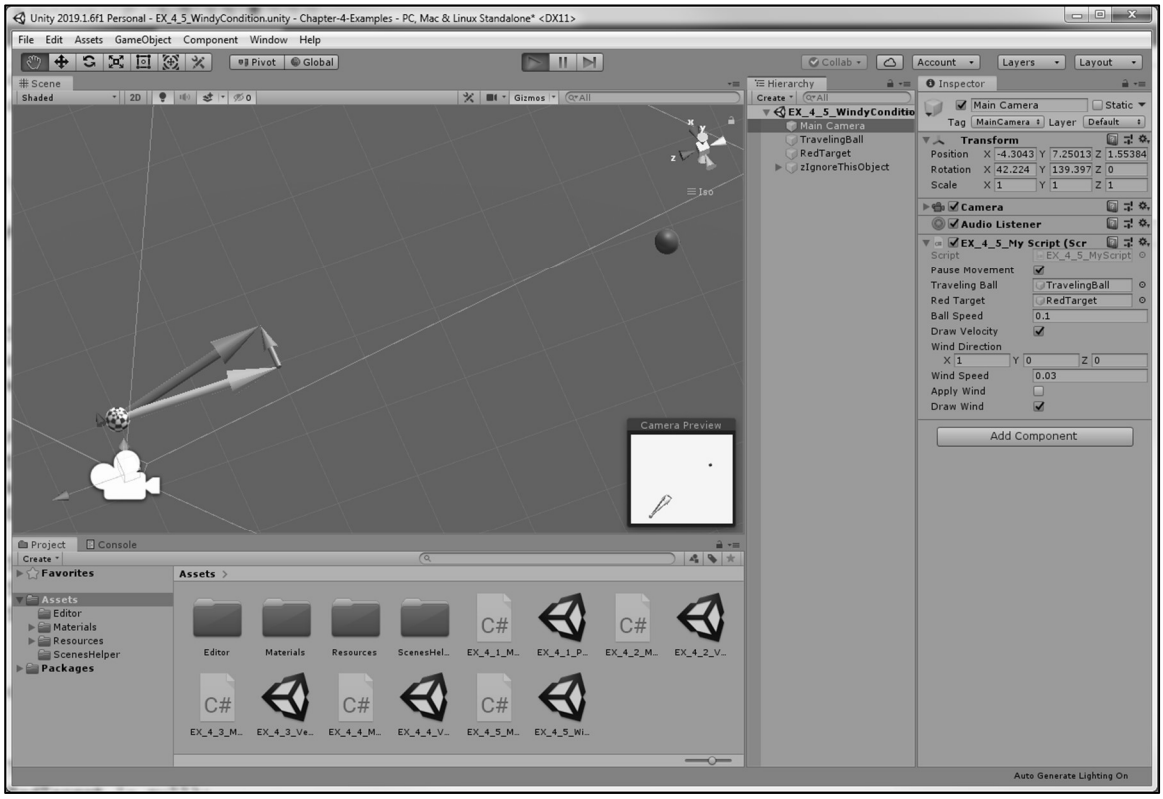

Figure 4-19. *Running the Windy Condition example*

The goals of this example are for you to:

- Experience a straightforward example of applying vector addition to affect object behavior
- Examine and understand the simple implementation of how velocity can be affected under a constant wind condition

Examine the Scene

Take a look at the Example_4_5_WindyCondition scene and observe the predefined game objects in the Hierarchy Window. In addition to the MainCamera, there are two objects in this scene: TravelingBall and RedTarget. This example simulates the TravelingBall progressing towards the RedTarget under a constant wind condition that affects its velocity.

Analyze MainCamera MyScript Component

The MyScript component on MainCamera shows four sets of variables.

- Simulation Control: Variables that control the simulation

  - PauseMovement: The toggle that stops the simulation and the movements of the objects in the scene, allowing for careful examination of the scene.

- The objects: The objects in the scene that you can interact with.
  - TravelingBall: The reference to the TravelingBall game object
  - RedTarget: The reference to the RedTarget game object
- Traveling ball speed: Variables that affect the speed of the traveling ball
  - BallSpeed: The speed at which the ball is traveling without any wind. Note that the direction of ball's velocity is along the vector defined by the ball and the target positions. Assuming P_B is the position of the ball, and P_T is the position of the target, then,

$$\vec{V}_T = BallSpeed \times (P_T - P_B).Normalized$$

  - DrawVelocity: A toggle to hide or show the ball's velocity vector, $\vec{V}_T$
- Wind condition: The variables that control the wind condition in the simulation
  - WindDirection: Determines the direction of the wind velocity, $\vec{V}_{wind}$
  - WindSpeed: Determines the speed of the wind velocity, $\vec{V}_{wind}$
  - ApplyWind: Toggles the effect of the wind on or off
  - DrawWind: A toggle to hide or show the wind's velocity vector

Interact with the Example

Click on the Play Button to run the example. Note that initially PauseMovement is enabled and the traveling ball does not move. The three vectors you observe are explained as follows. The green vector pointing from the TravelingBall towards the RedTarget is the ball's current velocity, $\vec{V}_T$. The red vector is the wind's velocity, $\vec{V}_{wind}$. Lastly, the blue vector is the path that the ball will take, the resulting vector, $\vec{V}_A$, where

$$\vec{V}_A = \vec{V}_T + \vec{V}_{wind}$$

Increase the BallSpeed and WindSpeed to observe the corresponding green and red vectors increase in length. Select and move the RedTarget to verify that the direction of the green vector, $\vec{V}_T$, always points towards the RedTarget. Next, select and change the components of the WindDirection variable to verify that the direction of the red vector changes accordingly.

Now, switch off PauseMovement toggle to allow the simulation to proceed. Try increasing WindSpeed, e.g., to 0.05, and observe $\vec{V}_T$ being compensated and adjusted while the TravelingBall proceeds and drifts towards the RedTarget. Note that when WindSpeed and WindDirection are unfavorable, e.g., a speed of 0.15 in the direction of $(1, 0, 0)$, the TravelingBall will drift away from and never reach the RedTarget.

Details of MyScript

Open MyScript and examine the source code in the IDE. The instance variables are as follows.

```
public bool PauseMovement = true;

public GameObject TravelingBall = null;
```

```
public GameObject RedTarget = null;

public float BallSpeed = 0.01f;            // units per second
public bool DrawVelocity = false;
private float VelocityDrawFactor = 20f;    // So that we can see the vector drawn

public Vector3 WindDirection = Vector3.zero;
public float WindSpeed = 0.01f;
public bool ApplyWind = false;
public bool DrawWind = false;

private MyVector ShowVelocity = null;
private MyVector ShowWindVector = null;
private MyVector ShowActualVelocity = null;
```

All public variables for MyScript have been discussed when analyzing MainCamera's MyScript component. The private variable VelocityDrawFactor is for scaling the small magnitude velocity vectors such that they can be visible. The MyVector data type private variables are to visualize the three vectors, $\vec{V}_T$, $\vec{V}_{wind}$, and $\vec{V}_A$. The Start() function for MyScript is listed as follows.

```
void Start() {
    Debug.Assert(TravelingBall != null);
    Debug.Assert(RedTarget != null);

    ShowVelocity = new MyVector() {
        VectorColor = Color.green,
        DrawVectorComponents = false };
    ShowWindVector = new MyVector() {
        VectorColor = new Color(0.8f, 0.3f, 0.3f, 1.0f),
        DrawVectorComponents = false  };
    ShowActualVelocity = new MyVector() {
        VectorColor = new Color(0.3f, 0.3f, 0.8f, 1.0f),
        DrawVectorComponents = false     };
}
```

As in all previous examples, the Debug.Assert() calls ensure proper setup regarding referencing the appropriate game objects via the Inspector Window, while the rest of the function instantiates the MyVector variables for proper visualization of the vectors. The Update() function is listed as follows.

```
void Update() {
    Vector3 vDir = RedTarget.transform.localPosition - TravelingBall.transform.localPosition;
    float distance = vDir.magnitude;

    if (distance > float.Epsilon) { // if not already at the target
        vDir.Normalize();
        WindDirection.Normalize();

        Vector3 vT = BallSpeed * vDir;
        Vector3 vWind = WindSpeed * WindDirection;
        Vector3 vA = vT + vWind;

        // Display the vectors

        if (PauseMovement)
            return;

        if (ApplyWind)
```

```
            TravelingBall.transform.localPosition += vA * Time.deltaTime;
        else
            TravelingBall.transform.localPosition += vT * Time.deltaTime;
    } // if (ditance < float.Epsilon)
}
```

The Update() function first computes the vector from TravelingBall towards the RedTarget, $\vec{V}_{dir}$. Next, the magnitude of $\vec{V}_{dir}$, distance, is computed and checked to ensure that this is not a very small number. This checking accomplishes two important objectives. First, a small distance value means that the TravelingBall object is closed to or has reached the RedTarget object and further simulation is no longer required. Second, when distance is approximately zero, $\vec{V}_{dir}$ is approximately a zero vector and thus cannot be normalized. When distance is larger than approximately zero, the following velocity vectors are computed:

$$\vec{V}_T = BallSpeed \times \hat{V}_{dir}$$

$$\vec{V}_{wind} = WindSpeed \times \widehat{WindDirect}$$

$$\vec{V}_A = \vec{V}_T + \vec{V}_{wind}$$

When the simulation condition is true, depending on if the user wants to observe the effects of the wind, the TravelingBall position is updated by either $\vec{V}_T \times elapsedTime$, or $\vec{V}_A \times elapsedTime$.

Take Away from This Example

This example demonstrates the straightforward application of vector addition by simulating traveling under a constant, external effect, like a wind condition. You have observed that such a condition can be simulated as a velocity vector being added to the traveling velocity.

Relevant mathematical concepts covered include:

- Model constant wind breeze as a velocity
- Changing an object's velocity by the addition of an object's own velocity with that of external velocities

EXERCISES

Compensate for the Wind Conditions

Note that if the wind velocity, $\vec{V}_{wind}$, is available during the computation of an object's velocity, $\vec{V}_T$, then it is possible to compensate for the wind condition. Instead of moving towards the target, $\hat{V}_{dir}$, the traveling velocity should point towards the target only after $\hat{V}_{dir}$ is affected by the wind condition, or,

$$BallSpeed \times \hat{V}_{dir} = \vec{V}_T + \vec{V}_{wind}$$

So

$$\vec{V}_T = BallSpeed \times \hat{V}_{dir} - \vec{V}_{wind}$$

Implement this compensation and observe a smoother TravelingBall movement. You have observed that it is possible to compensate and largely remove the external wind factor by not traveling directly towards the final destination.

Travel Under Multiple External Factors

Support a strong wind gust which occurs probabilistically (or pseudo-randomly). In addition to speed and direction allow your user to adjust the occurrence frequency and duration of this wind gust. Now, as the TravelingBall moves towards its target, it may get blown off course some of the times. You now know how to add simple environmental factors into a game.

Summary

This chapter introduces vectors by relating to your understanding of measurement and distance computations in the Cartesian Coordinate System. You have learned:

- a vector is a size and a direction that can relate two positions,
- the vector definition is independent of any particular position,
- all positions in the Cartesian Coordinate System can be considered as position vectors,
- scaling a vector by a floating-point number changes its size but not its direction,
- a normalized or unit vector has a size of 1 and is convenient for representing the direction of a vector,
- vectors are ideal for representing the velocities of objects,
- it is convenient to represent a velocity by separately storing its speed and direction of movement,
- vector addition and subtraction rules follow closely to those of floating-point algebra.

The examples presented in this chapter allowed you to interact with and examine the details of vectors and their operations. Based on vector concepts, you have examined the simple object behaviors of following, or aiming, at a target, and the environmental affects you can create by disturbing an object's motion with an external velocity.

Through this chapter, you have gained the basic knowledge of what a vector is, its basic rules, and how it can be used to model simple object behaviors and environmental effects. You are now ready to examine the more advanced operations of vectors, like the dot product, which determines the relationship of two given vectors.

Before you continue, it is important to remember that the applications of vector related concepts go far beyond interactive graphical applications like video games. In fact, in many cases it is impossible to depict or visualize the vectors being used in different applications. For example, a vector in n-dimensional space where n is significantly large than 100! It is important to remember that you are learning one flavor of vector usage: applications in interactive graphics. In general, vectors can be applied to solve problems in a wide variety of disciplines.