CHAPTER 5

■ ■ ■

Chapter 5: Vector Dot Products

After completing this chapter, you will be able to:

- Understand the vector dot product definition, its properties, and its geometric interpretation,
- Recognize how the vector dot product relates two vectors by their subtended angle and relative projection sizes,
- Comprehend how a vector represents a line segment,
- Apply the dot product to allow the interpretation of a line segment as an interval,
- Perform the simple inside-outside test for a point and an arbitrary interval,
- Apply the vector dot product to determine the shortest distant between a point and a line,
- Apply the vector dot product to compute the closest distance between two lines.

Introduction

In Chapter 4 you learned that a vector is defined by the relationships between two positions in the Cartesian Coordinate System; the direction from one position to another, and the distance between them. Though simple, the vector, or the concept and the associated rules of relating two positions, is demonstrated to be a powerful tool that is capable of representing object velocity and simple environmental effects for video game development. This chapter continues with this theme and introduces the vector dot product to relate two vectors.

Vectors are defined by their direction and magnitude and thus, when relating two vectors, it is essential to include descriptions of how these two quantities are measured with respect to each other. The vector dot product relates vector directions by calculating the cosine of the subtended angle, or the angle between two vectors where their tails are connected, and the vectors' magnitudes by computing the respective projected sizes, or one vector's magnitude when measured along the direction of the other vector. These ways of relating vectors are some of the most fundamental tools in analyzing the proximity and connections between positions and directions in 3D space. The results of applying the vector dot product provide the basis for predicting and controlling object behaviors in almost all video games.

In video games it is often necessary to analyze the spatial relationships, such as distances and intersections, of traveling objects and then predicting what events will occur. For example, detecting and hinting to the player the situation where the pathway of their explorer will pass within a hidden treasure's proximity. To model this situation mathematically, as you have learned from the previous chapters, the pathway of the explorer is a function of her traveling velocity and can be represented as a vector. Then, the hidden treasure can be wrapped by a bounding volume, e.g., bounding sphere. In this way, the problem to solve is to compute the closest distance between the vector and bounding sphere center and determine if that distance is closer than the bounding sphere radius. As you will learn from this chapter, the vector dot product can provide a solution for this situation that is elegant and

straightforward to implement. In fact, the vector dot product is the best tool for determining distances between positions and line segments.

This chapter begins by introducing the vector dot product, what it is, how it is computed, and the rules for working with the operation. The chapter then moves on to explain how to geometrically interpret the dot product results as the angle between vectors and as projected lengths along these vectors. The inside-outside test of a 1D interval along a major axis discussion from Chapter 2 is then cast and generalized as an inside-outside problem based on vector line segments and projections. The two application areas of the vector dot product that are examined specifically are the line to point and the line to line distances. These types of applications play many roles in video game development as well as other interactive graphic applications. Finally, this chapter concludes by reviewing what you have learned about the vector dot product and its many applications.

Vector Dot Product: Relating Two Vectors

Recall that the vector definition is independent of any position. In other words, a vector can have its tail located at any position. This knowledge is important because when you analyze the relationship between two vectors, it is convenient to depict the tails of the vectors at the same location. Figure 5-1 shows a drawing of two arbitrary vectors, $\vec{V}_1$ and $\vec{V}_2$, with the same tail position, P_0. As you can see, the shared tail position allows the two vectors to be in close proximity and facilitates convenient visual comparison. By placing two vectors at the same location, it becomes easier to analyze, understand, and quantify the relationship between these two vectors.

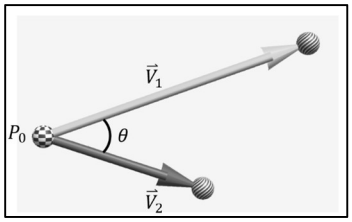

Figure 5-1. *Relationship between two given vectors*

Notice in Figure 5-1 that although the two vectors could be in any direction with any magnitude in 3D Cartesian Coordinate Space, the two vectors together can always be properly depicted on a 2D plane. In fact, the 2D plane that these vectors are depicted on may or may not be parallel to any major axes. In general, it is true that given any two arbitrary vectors in 3D space, as long as the two vectors are not parallel, there is always a 2D plane where both of the vectors in 3D space can be drawn. This observation is what allows two vectors in 3D space to be drawn and analyzed on a 2D plane, as depicted in Figure 5-1.

The second observation from Figure 5-1 is that, recalling that a vector is comprised of a direction and a magnitude, that the relationship between two vectors can be characterized by the angle between the vectors, θ, and by the relative sizes of the two vectors. The vector dot product is the operation that can provide definitive answers to both of these characteristics.

Definition of Vector Dot Product

Given two vectors in 3D space,

$$\vec{V}_1 = (x_1, y_1, z_1)$$

$$\vec{V}_2 = (x_2, y_2, z_2)$$

the **dot product**, or vector dot product, between the two vectors is defined as the sum of the product of the corresponding coordinate components, or,

$$\vec{V}_1 \cdot \vec{V}_2 = x_1x_2 + y_1y_2 + z_1z_2$$

Notice that,

- **Symbol**: the symbol for the dot product operation, "·", is literally a "dot"
- **Operands**: the operation expects two vector operands
- **Result**: the result of the operation is a floating-point number.

It is especially important to pay attention to the last point. Similar to vector addition and subtraction, the dot product operates on two vector operands. However, unlike the other two operations, the result of the dot product is not a vector but a simple floating-point number. It is this floating-point number that encodes the angle between the two operand vectors and the relative sizes of the two operand vectors. How these values are encoded in this single floating-point number and what you can do with it are the topics that will be explored in the following subsections. However, before you begin that journey, you will first need to explore and understand the rules and properties of the dot product.

■ ***Note*** *The dot product is also referred to as the* ***inner product****, or the* ***scalar product*** *in different disciplines of mathematics. This book will refer to the operation as dot product exclusively.*

■ ***Note*** *Do not confuse the dot product symbol, "·" for a multiplication sign. When multiplications are involved in vectors expressions, there will be no symbol between the operands, such as* $s\vec{V}_2$*. Since one cannot multiply two vectors, you will never see* $\vec{V}_1\vec{V}_2$*, and therefore you can safely assume that if you see a "·" between two vectors, the dot product is the operation to perform and not multiplication.*

Properties of Vector Dot Product

The vector dot product properties of commutative, associative, and distributive over a floating-point scaling factor s and other vector operations are summarized in Table 5-1.

Table 5-1*. Properties of vector dot product*

Properties	Vector Dot Product
Commutative	$\vec{V}_1 \cdot \vec{V}_2 = \vec{V}_2 \cdot \vec{V}_1$
Associative	$(\vec{V}_1 \cdot \vec{V}_2) \cdot \vec{V}_3$ [**Undefined!**]
Distributive over vector operation	$(\vec{V}_1 \cdot \vec{V}_2)(\vec{V}_A + \vec{V}_B) = (\vec{V}_1 \cdot \vec{V}_2)\vec{V}_A + (\vec{V}_1 \cdot \vec{V}_2)\vec{V}_B$

Distributive over scale factor, s	$s(\vec{V}_1 \cdot \vec{V}_2) = (s\vec{V}_1) \cdot \vec{V}_2 = \vec{V}_1 \cdot (s\vec{V}_2)$

Take note of the undefined associative property. In this situation it can be helpful to remember that the result of the dot product operation is a floating-point number, so it is possible to let,

$$(\vec{V}_1 \cdot \vec{V}_2) = f$$

then it becomes obvious that,

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

is an undefined operation since the first operand is not a vector but a floating-point number. In general, please pay attention to the subtle differences in the notation. While

$$(\vec{V}_1 \cdot \vec{V}_2)\vec{V}_A = f\vec{V}_A$$

is scaling vector $\vec{V}_A$ by the result of the dot product,

$$(\vec{V}_1 \cdot \vec{V}_2) \cdot \vec{V}_A = f \cdot \vec{V}_A$$

is attempting to perform a dot product between a floating-point number, f, and the vector, $\vec{V}_A$, and is therefore an undefined operation. The only difference is in the single "·" symbol! If you continue to use f to represent the result of $\vec{V}_1$ dot $\vec{V}_2$, then you can rewrite the distributive property over vector addition as,

$$(\vec{V}_1 \cdot \vec{V}_2)(\vec{V}_A + \vec{V}_B) = f(\vec{V}_A + \vec{V}_B) = f\vec{V}_A + f\vec{V}_B$$

which is the distributive property of vector addition over a scaling factor, f. This means that the distributive property also applies over vector subtraction,

$$(\vec{V}_1 \cdot \vec{V}_2)(\vec{V}_A - \vec{V}_B) = (\vec{V}_1 \cdot \vec{V}_2)\vec{V}_A - (\vec{V}_1 \cdot \vec{V}_2)\vec{V}_B$$

or,

$$(\vec{V}_1 \cdot \vec{V}_2)(\vec{V}_A - \vec{V}_B) = f(\vec{V}_A - \vec{V}_B) = f\vec{V}_A - f\vec{V}_B$$

The vector dot product distributive property over a scale factor, s, is worth some special attention. Notice, that the scale factor s is only applied to one of the operands and not both. At first glance this may seem counterintuitive, however, it makes perfect sense if you consider distributive property over a scale factor, s, of a floating-point multiplication between a and b,

$$s \times (a \times b) = (s \times a) \times b = a \times (s \times b)$$

Now, recall that the magnitude of a vector, $\vec{V} = (x,\ y,\ z)$ is,

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

For this reason, a vector dotted with itself is its magnitude squared,

$$\vec{V}_1 \cdot \vec{V}_1 = x_1x_1 + y_1y_1 + z_1z_1 = {x_1}^2 + {y_1}^2 + {z_1}^2 = \|\vec{V}_1\|^2$$

Lastly, the dot product between any vector with the zero vector always result in a zero vector,

$$\vec{V}_1 \cdot ZeroVector = ZeroVector \cdot \vec{V}_1 = ZeroVector$$

The Angle between Two Vectors

This section derives a formula that computes the angle θ between the vectors $\vec{V}_1$ and $\vec{V}_2$ in Figure 5-1. As illustrated in Figure 5-2, this formula derivation begins by subtracting the two given vectors,

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

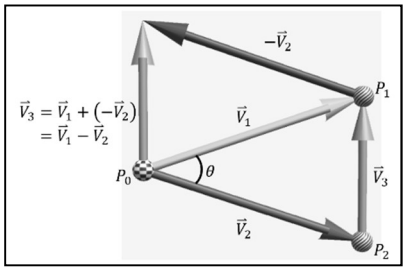

Figure 5-2. *Subtracting the given two vectors*

In Figure 5-2, similar to Figure 5-1, both $\vec{V}_1$ and $\vec{V}_2$ have their tails located in the lower-left corner at position P_0. Notice the $-\vec{V}_2$ vector with its tail at position P_1 and that the vector $\vec{V}_3$ with its tail at P_0 is the result of adding $\vec{V}_1$ with $-\vec{V}_2$, or,

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

Figure 5-2 also depicts vector $\vec{V}_3$ with its tail at P_2 to create triangle $P_0P_1P_2$. Recall that the Laws of Cosine from Trigonometry states that

$$\|\vec{V}_3\|^2 = \|\vec{V}_1\|^2 + \|\vec{V}_2\|^2 - 2\|\vec{V}_1\|\|\vec{V}_2\| \cos\theta$$

In this case, you know that,

$$\vec{V}_1 = (x_1, y_1, z_1)$$

$$\vec{V}_2 = (x_2, y_2, z_2)$$

$$\vec{V}_3 = (x_1 - x_2, y_1 - y_2, z_1 - z_2)$$

With algebraic simplification left as an exercise, you can show that,

$$\cos\theta = \frac{x_1x_2 + y_1y_2 + z_1z_2}{\sqrt{{x_1}^2 + {y_1}^2 + {z_1}^2}\sqrt{{x_2}^2 + {y_2}^2 + {z_2}^2}}$$

this equation says that,

$$x_1x_2 + y_1y_2 + z_1z_2 = \sqrt{{x_1}^2 + {y_1}^2 + {z_1}^2}\sqrt{{x_2}^2 + {y_2}^2 + {z_2}^2} \cos\theta = \|\vec{V}_1\|\|\vec{V}_2\| \cos\theta$$

or simply,

$$\vec{V}_1 \cdot \vec{V}_2 = x_1x_2 + y_1y_2 + z_1z_2 = \|\vec{V}_1\|\|\vec{V}_2\| \cos\theta$$

You have just shown that the dot product definition, the sum of the products of the corresponding coordinate components, actually computes a floating-point number that is equal to the product of the magnitude of the two vectors and the cosine of the angle between these two vectors. By normalizing $\vec{V}_1$ and $\vec{V}_2$, $\|\vec{V}_1\|$ and $\|\vec{V}_2\|$ both becomes 1.0, so that,

$$\hat{V}_1 \cdot \hat{V}_2 = \|\hat{V}_1\|\|\hat{V}_2\| \cos\theta = \cos\theta$$

This formula says that the dot product of two normalized vectors is the cosine of the angle between the vectors.

It is important to note that the angle between two vectors is the one subtended by the two vectors (the smaller angle). As illustrated in Figure 5-3, if θ in Figure 5-1 was $45°$, then the angle between the two vectors is $45°$ and not $315°$. The key to remember is that the angle subtended by two vectors, or two lines, is always between $0°$ and $180°$.

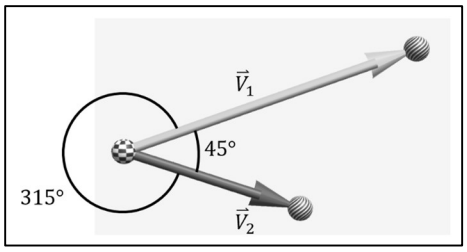

Figure 5-3. *The angle subtended by vectors $\vec{V}_1$ and $\vec{V}_2$*

Figure 5-4 depicts the angle measurements $\theta_2$to θ_6 between vector $\vec{V}_1$ and vectors $\vec{V}_2$ to $\vec{V}_6$ respectively. In this case, $\vec{V}_3$ is perpendicular to $\vec{V}_1$, and $\vec{V}_5$ is in the opposite direction to $\vec{V}_1$, thus $\theta_3 = 90°$, while $\theta_5 = 180°$. Notice the measurement of the angle θ_6, the angle between vectors $\vec{V}_1$ and $\vec{V}_6$, is the angle subtended by these two vectors and is not an accumulation from the angle θ_5. Once again and very importantly, the angle subtended by two vectors is always an angle between $0°$ and $180°$.

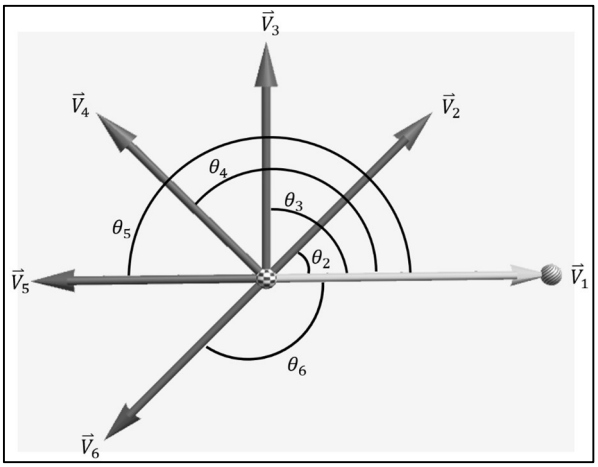

Figure 5-4. *The angles between vectors*

Figure 5-5 is a simple plot and a reminder of the cosine function. Recall that the results of $\cos\theta$ is positive between $0°$ to $90°$ and becomes negative between $90°$and $180°$. With the dot product of two normalized vectors being the cosine of the subtended angle, you can now determine the relative directions of vectors with a simple dot product calculation. In particular, when the subtended angle is less than $90°$, the cosine is positive, and thus you can conclude that the vectors are pointing along a similar direction. Conversely, when the subtended angle is more than $90°$, the cosine is negative, and thus you can conclude that the vectors are pointing away from each other.

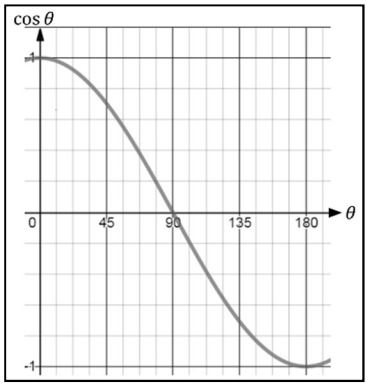

Figure 5-5. *Simple plot of the $y = cos\ \theta$ function*

In the cases of Figure 5-4, you know,

$$\hat{V}_1 \cdot \hat{V}_2 = \cos\theta_2 = \text{a positive number}; \quad \text{because } \theta_2 < 90°$$
$$\hat{V}_1 \cdot \hat{V}_3 = \cos\theta_3 = 0 \quad \text{because } \theta_3 = 90°$$
$$\hat{V}_1 \cdot \hat{V}_4 = \cos\theta_4 = \text{a negative number}; \quad \text{because } \theta_4 > 90°$$
$$\hat{V}_1 \cdot \hat{V}_5 = \cos\theta_5 = -1 \quad \text{because } \theta_5 = 180°$$
$$\hat{V}_1 \cdot \hat{V}_6 = \cos\theta_6 = \text{a negative number} \quad \text{because } \theta_6 > 90°$$

These observations can be summarized in Table 5-2 for any given vectors, $\vec{V}_1$ and $\vec{V}_2$.

Table 5-2*. Properties of dot product between*

Dot Product Results	The Angle θ	Conclusions
$\hat{V}_1 \cdot \hat{V}_2 = \cos\theta = 1$	$\theta = 0°$	The vectors are in the exact same direction, $\hat{V}_1 == \hat{V}_2$
$\hat{V}_1 \cdot \hat{V}_2 = \cos\theta = 0$	$\theta = 90°$	The vectors directions are perpendicular to each other
$\hat{V}_1 \cdot \hat{V}_2 = \cos\theta > 0$	$\theta < 90°$	The vectors are pointing along similar directions
$\hat{V}_1 \cdot \hat{V}_2 = \cos\theta < 0$	$\theta > 90°$	The vectors are pointing along similar, but opposite directions
$\hat{V}_1 \cdot \hat{V}_2 = \cos\theta = -1$	$\theta = 180°$	The vectors are in the exact opposite direction, $\hat{V}_1 == -\hat{V}_2$

The Angle Between Vectors Example

This example allows you to manipulate three positions that define two vectors. The example computes and displays the angle between the two vectors and enables you to verify the conclusions gathered from Table 5-2. Additionally, this example demonstrates that as long as the two given vectors are not parallel, a 2D plane can always be found for drawing the two vectors. Figure 5-6 shows a screenshot of running the EX_5_1_AngleBetweenVectors example from the Chapter-5-Examples project.

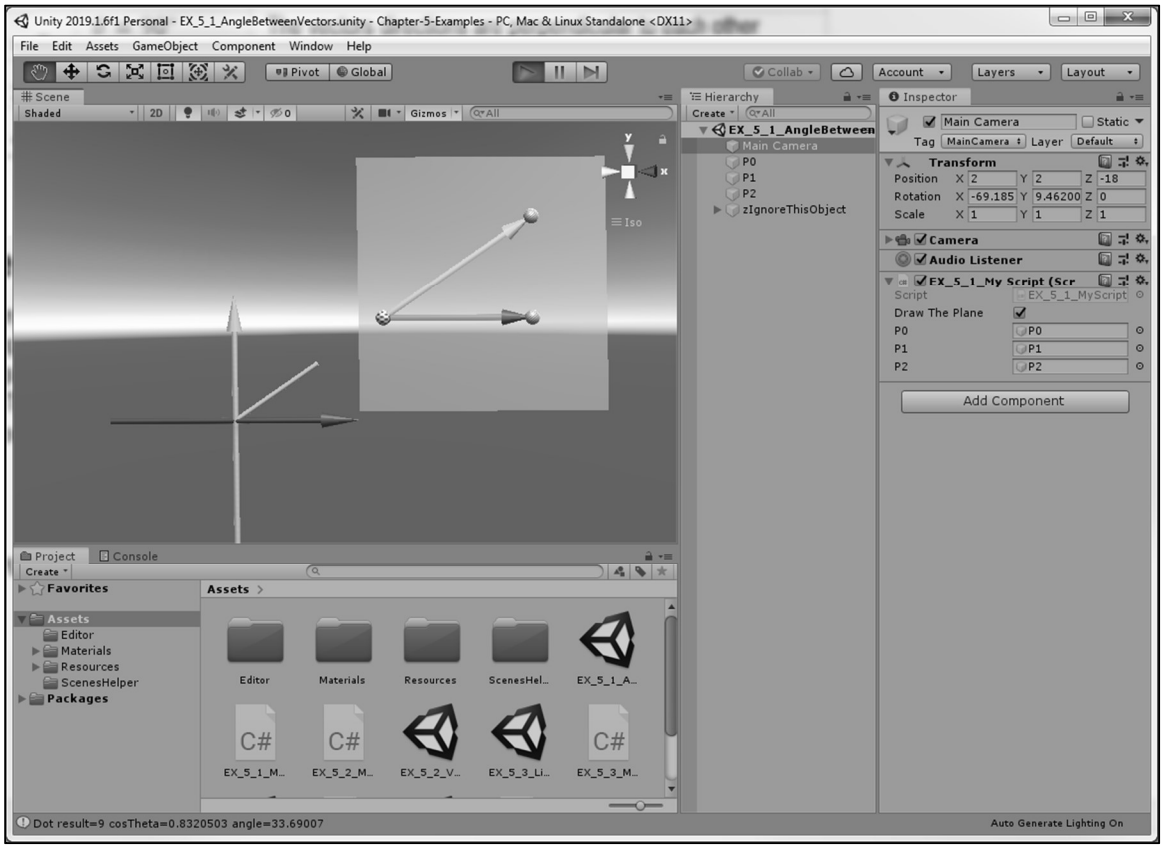

Figure 5-6. *Running the Angle Between Vectors example*

The goals of this example are for you to:

- Experience manipulating the angle subtended by two vectors and observe the results of the dot product
- Verify that a 2D plane can always be found for drawing two non-parallel vectors
- Examine the implementation of and appreciate the subtleties of vector normalization when computing dot products

Examine the Scene

Take a look at the Example_5_1_AngleBetweenVectors scene and observe the predefined game objects in the Hierarchy Window. In addition to the MainCamera, there are three objects in this scene, the checkered sphere (P0), and the stripped spheres (P1 and P2). These three game objects, with their corresponding transform.localPosition, will be referenced to define the two vectors for performing dot product calculations.

Analyze MainCamera MyScript Component

The MyScript component on MainCamera shows four variables: P0, P1, P2, and DrawThePlane toggle. The toggle is for showing or hiding the 2D plane where the two vectors are drawn, while the other three variables are defined for accessing the game objects with their corresponding names. In this example, you will manipulate the positions of the three game objects and examine the dot product resulting from the vectors, $\vec{V}_1$ and $\vec{V}_2$, defined accordingly,

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

$$\vec{V}_2 = P_2 - P_0$$

Interact with the Example

Click on the Play Button to run the example. In the Scene View window, you will observe two vectors with tail positions located at the checkered sphere, P0, and a greenish plane where the two vectors are drawn. The two vectors are the $\vec{V}_1$ and $\vec{V}_2$, and are defined by the positions of P0, P1, and P2 game objects. Also visible in the Scene View window is the 2D axis frame with the red X-axis and green Y-axis vectors. On the axis frame, extending from the origin is a black line segment. The angle subtended by this black line segment and the red X-axis is the same angle subtended by vectors $\vec{V}_1$ and $\vec{V}_2$, and the length of this black line is proportional to the cosine of that angle, scaled by 1.5 times for easier visual analysis. Lastly, take a look at the Console Window to observe the text output reporting the computed angle between vectors $\vec{V}_1$ and $\vec{V}_2$.

Now that you have looked over the scene, you will manipulate and observe the cosine of the angle subtended by the two vectors and how the angle itself changes. Please switch off the DrawThePlane toggle as the 2D plane can be distracting. Next, select P1 and change its x- and y-component values to vary the angle between the two vectors. In the Console Window, you can verify the values of the subtended angle and the cosine of this angle. Observe how the black line segment, with its length changing, rotates towards or away from the red X-axis direction, corresponding to the angle changes you are making.

Since the length of the black line segment is proportional to the cosine of the subtended angle, from the plot in Figure 5-5 and Table 5-2, you can verify that when the subtended angle increases, up to $90°$, the cosine of the angle decreases, and thus the length of the line also decreases. The opposite is also true, as the angle decreases (between $90°$ and $0°$), the cosine of the angle, and thus the length of the black line, increases. In fact, you should notice that the length of the black line is maximized when the subtended angle approaches zero, and that the length of the line approaches zero when the two vectors are approximately perpendicular. You can observe this behavior by decreasing the P1 x-component value such that the subtended angle approaches $90°$. When doing so, notice how the length of black line segment also approaches zero, corresponding to $\cos 90° = 0$.

When you increase the subtended angle beyond $90°$, you will notice the color of the black line segment change to red, indicating that the sign of the dot product result has turned into a negative number. Now, decrease the y-component value of P1 to continue to increase the subtended angle and notice that the red line segment continues to grow in length once more as it rotates away from the positive X-axis direction. When $\vec{V}_1$ and $\vec{V}_2$ are approximately in the opposite direction, the red line segment will achieve maximum length and should be on top of the negative red X-axis line, indicating the angle between the two vectors is about $180°$ and that $\cos 180° = -1$. Now, notice that any attempt to increase the subtended angle beyond $180°$ will cause the red line segment to rotate back towards the positive X-axis direction. This is similar to cases of vectors that are between $\vec{V}_5$ and $\vec{V}_6$ in

Figure 5-4. This exercise is to reaffirm that subtended angles are always between $0°$ and $180°$ and to visually demonstrate what Table 5-2 showcases.

Next, you will verify that a 2D plane can always be defined to draw two vectors that are not parallel. Please switch on the DrawThePlane toggle and rotate the camera to see that the two vectors are indeed drawn on the greenish plane by examining that the plane slices through the two arrows representing $\vec{V}_1$ and $\vec{V}_2$ respectively. You can manipulate any of the P0, P1, or P2 positions to observe the vectors change accordingly and more importantly, observe that the green plane also changes accordingly, always cutting through both of the vectors. Now, adjust P1 to the exact location of P2. One way you can do this is by copying the values from P2's transform components in the Inspector window and pasting them onto that of P1's corresponding transform components. Once done, notice that the 2D plane disappears. In this case, since the two vectors are pointing in the exact same direction, there are an infinite number of 2D planes that can cut through the vectors and thus none are shown.

Details of MyScript

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

```
// Three positions to define two vectors: P0->P1, and P0->P2
public GameObject P0 = null;    // Position P0
public GameObject P1 = null;    // Position P1
public GameObject P2 = null;    // Position P2

public bool DrawThePlane = true;

#region For visualizing the vectors
#endregion
```

All the public variables for MyScript have been discussed when analyzing the MainCamera's MyScript component. The code in the "For visualizing the vectors" region is specific to drawing the vectors and as usual does not pertain to the math being discussed in this section.

■ **Note** *By now, you have observed and may even have worked with some of the visualization code. From here on, the visualization portion of MyScript will become increasingly complex and involved. To avoid unnecessary distractions, beginning from this example, the code for visualization will be separated into collapsed hidden regions. The details of these regions will not be explained or brought up as they can be tedious and in all cases, are irrelevant to the concepts being discussed. You are very welcome to explore these at your leisure.*

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

```
void Start() {
    Debug.Assert(P0 != null);    // Verify proper setting in the editor
    Debug.Assert(P1 != null);
    Debug.Assert(P2 != null);

    #region For visualizing the vectors
```

```
    #endregion
}
```

As in all previous examples, the Debug.Assert() calls ensure proper setup regarding referencing the appropriate game objects via the Inspector Window. The region "For visualizing the vectors", which contains the details of initializing the visualization variables for the vectors in the scene are, once again, irrelevant to the math being discussed and can be distracting. Therefore, this region will not be discussed. The Update() function is listed as follows.

```
void Update() {
    float cosTheta = float.NaN;
    float theta = float.NaN;
    Vector3 v1 = P1.transform.localPosition - P0.transform.localPosition;
    Vector3 v2 = P2.transform.localPosition - P0.transform.localPosition;
    float dot = Vector3.Dot(v1, v2);
    if ((v1.magnitude > float.Epsilon) && (v2.magnitude > float.Epsilon)) {
        cosTheta = dot / (v1.magnitude * v2.magnitude);
        // Alternatively,
        //    costTheta = Vector3.Dot(v1.normalize, v2.normalize)
        theta = Mathf.Acos(cosTheta) * Mathf.Rad2Deg;
    }
    Debug.Log("Dot result=" + dot + " cosTheta=" + cosTheta + " angle=" + theta);

    #region For visualizing the vectors
    #endregion
```

The first three lines of the Update() function compute,

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

$$\vec{V}_2 = P_2 - P_0$$

$$dot = \vec{V}_1 \cdot \vec{V}_2$$

The if condition ensures that neither of the vectors are the zero vector, which, as you have learned, does not have a length, cannot be normalized, and thus, cannot subtend angles. When both of the vectors are properly defined, the cosine of the angle between them can be computed by recognizing that,

$$dot = \vec{V}_1 \cdot \vec{V}_2 = \|\vec{V}_1\|\|\vec{V}_2\| \cos\theta$$

Which means that the cosine of the subtended angle is simply,

$$\cos\theta = \frac{dot}{\|\vec{V}_1\|\|\vec{V}_2\|}$$

Finally, theta, the subtended angle value, can be derived by the arccosine function. Note that alternatively, $\cos\theta$ can be computed by performing the dot operation with the normalized version of the two vectors. The dot products between vectors that are normalized will be examined in more details in the following sections.

Take Away from This Example

This example verifies that when given two non-parallel vectors, a 2D plane can always be derived to draw the two vectors. This fact allows the examination of the two arbitrary vectors, which may not be aligned with any major axes, to be drawn, examined, and analyzed. You have interacted with and closely examined the angle subtended by two vectors, and that, that angle is always between $0°$ and $180°$. Finally, you have observed that the cosine of a subtended angle can be computed by dividing the dot product of the two vectors with their magnitudes, or, alternatively, from the dot product of the two vectors after they have been normalized.

$$\cos\theta = \frac{dot}{\|\vec{V}_1\|\|\vec{V}_2\|} = \hat{V}_1 \cdot \hat{V}_2$$

Relevant mathematical concepts covered include:

- The dot product of normalized vectors is the cosine of their subtended angle
- The value of the dot product provides insights into the relative directions of the operand vectors, (see Table 5-2)
- A unique 2D plane can be derived from two non-parallel vectors such that both vectors can be drawn on the plane

Unity Tools:

- The Mathf library can be used for mathematical functions
- Rad2Deg: the scale factor for radian to degree conversion
- Acos can be used to compute arccosine
- The Mathf.Acos function returns the angle in units of radian and not degree

EXERCISES

Derive the Dot Product Formula

Given that,

$$\|\vec{V}_3\|^2 = \|\vec{V}_1\|^2 + \|\vec{V}_2\|^2 - 2\|\vec{V}_1\|\|\vec{V}_2\|\cos\theta$$

and that,

$$\vec{V}_1 = (x_1, y_1, z_1)$$

$$\vec{V}_2 = (x_2, y_2, z_2)$$

$$\vec{V}_3 = (x_1 - x_2, y_1 - y_2, z_1 - z_2)$$

Show that,

$$\cos\theta = \frac{x_1x_2 + y_1y_2 + z_1z_2}{\sqrt{{x_1}^2 + {y_1}^2 + {z_1}^2}\sqrt{{x_2}^2 + {y_2}^2 + {z_2}^2}}$$

Verify the Need for Normalization

When computing theta in MyScript,

```
cosTheta = dot / (v1.magnitude * v2.magnitude);
theta = Mathf.Acos(cosTheta) * Mathf.Rad2Deg;
```

replace these two lines of code with the non-normalized vectors version,

```
theta = Mathf.Acos(Vector3.Dot(v1, v2)) * Mathf.Rad2Deg;
```

Try running your game and observe the error messages. Now, include proper normalization,

```
theta = Mathf.Acos(Vector3.Dot(v1.normalize, v2.normalize)) * Mathf.Rad2Deg;
```

Try running this latest version and observe the same results as the original. This simple exercise shows that it is vital to normalize your vectors when computing the angle between them.

Verify the dot product formula

When computing theta in MyScript,

```
cosTheta = dot / (v1.magnitude * v2.magnitude);
```

replace this line of code with the explicit dot product computation,

```
cosTheta = (v1.x*v2.x + v1.y*v2.y + v1.z*v2.z) / (v1.magnitude * v2.magnitude);
```

Verify that runtime results are identical.

Vector Projections

You have learned that the dot product between two vectors, $\vec{V}_1$ and $\vec{V}_2$, computes the product of the vector magnitudes and the cosine of the angle subtended by the two vectors,

$$\vec{V}_1 \cdot \vec{V}_2 = \|\vec{V}_1\|\|\vec{V}_2\| \cos\theta$$

In the previous example, you have verified that by normalizing both of the vectors beforehand, ensuring that,

$$\|\hat{V}_1\| = \|\hat{V}_2\| = 1.0$$

That the dot product now simply computes the cosine of the angle between the given vectors,

$$\hat{V}_1 \cdot \hat{V}_2 = \cos\theta$$

Now, you can examine the two remaining ways of computing the dot product between two given vectors--with only one of the vectors being normalized, or,

$$\hat{V}_1 \cdot \vec{V}_2 = \|\vec{V}_2\| \cos\theta$$

$$\vec{V}_1 \cdot \hat{V}_2 = \|\vec{V}_1\| \cos\theta$$

Figure 5-7 depicts the geometric interpretation of these two dot product computations.

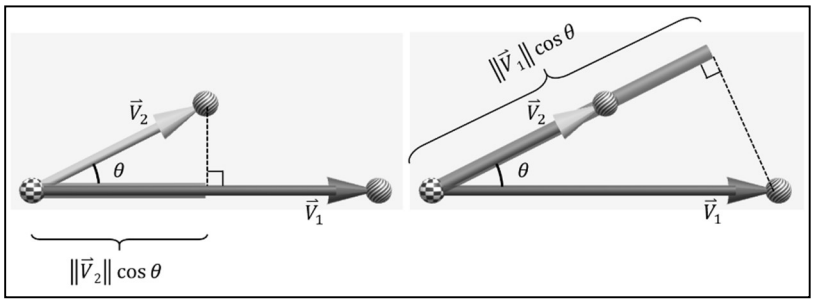

Figure 5-7*. Computing dot products between two vectors with one being normalized*

It is important to note that the left and right images of Figure 5-7 are both based on exactly the same two vectors, $\vec{V}_1$ and $\vec{V}_2$. The left image of Figure 5-7 shows that, with vector $\hat{V}_1$ normalized, the dot product computed,

$$\hat{V}_1 \cdot \vec{V}_2 = \|\vec{V}_2\| \cos\theta$$

is the length of $\vec{V}_2$ when measured along direction of the $\vec{V}_1$ vector. This is also referred to as the projected length of $\vec{V}_2$ on the $\vec{V}_1$ vector. Notice that with the tails of the two vectors located at the same position, the head of $\vec{V}_2$ is projected perpendicular to and onto the $\vec{V}_1$ vector, as evident by the dotted line with the right-angle indicator. This projected length can also be interpreted through trigonometry. You can treat $\vec{V}_2$ as the hypotenuse that subtends the angle, θ, with the base direction being $\hat{V}_1$ and the last side being the dotted line, thus forming a right-angle triangle. In this case, the length of the base of the right-angle triangle is $\|\vec{V}_2\| \cos\theta$.

The right image of Figure 5-7 shows the same two vectors, $\vec{V}_1$ and $\vec{V}_2$, where the dot product is computed with vector $\hat{V}_2$ being normalized instead of $\hat{V}_1$,

$$\vec{V}_1 \cdot \hat{V}_2 = \|\vec{V}_1\| \cos\theta$$

In this case, the dot product computes the exact complement of the previous case--the length of $\vec{V}_1$ when measured along the direction of $\vec{V}_2$, or the projected length of $\vec{V}_1$ on the $\vec{V}_2$ vector; or the length of the base of the right-angle triangle that is in the $\vec{V}_2$ direction and subtends an angle, θ, with the hypotenuse $\vec{V}_1$, and the dotted line as its final side. The right image of Figure 5-7 also illustrates a case where the length of the base of the right-angle triangle extends beyond the head of the vector $\vec{V}_2$. This example shows that the projected size can be larger than the magnitude of the vector that it is being projected onto, or,

$$\|\vec{V}_1\| \cos\theta > \|\vec{V}_2\|$$

Finally, remember that $\cos\theta$ is negative for $\theta > 90°$, and therefore, $\|\vec{V}\|\cos\theta$, or the projected size of a vector can actually be a negative value. In such cases, you know that the vector being projected onto is more than $90°$ away from the vector being projected. This turns out to be important information with many applications, some of which will be elaborated on in later subsections.

This discussion shows that, with the appropriate vector normalized, the dot product can compute the projection of the length of one vector onto the direction of the other vector and can provide a way to relate the lengths of these two vectors. In other words, the dot product allows you the capability to project one vector onto another. Observe that the normalized operand is the vector being projected onto. These projections are examined in the next example. The actual applications of the vector dot product will be discussed after the next section.

The Vector Projections Example

This example allows you to interact with and examine the results of vector projections. You will manipulate the definition of two vectors and examine the results of projecting these two vectors onto each other. Figure 5-8 shows a screenshot of running the EX_5_2_VectorProjections scene from the Chapter-5-Examples project.

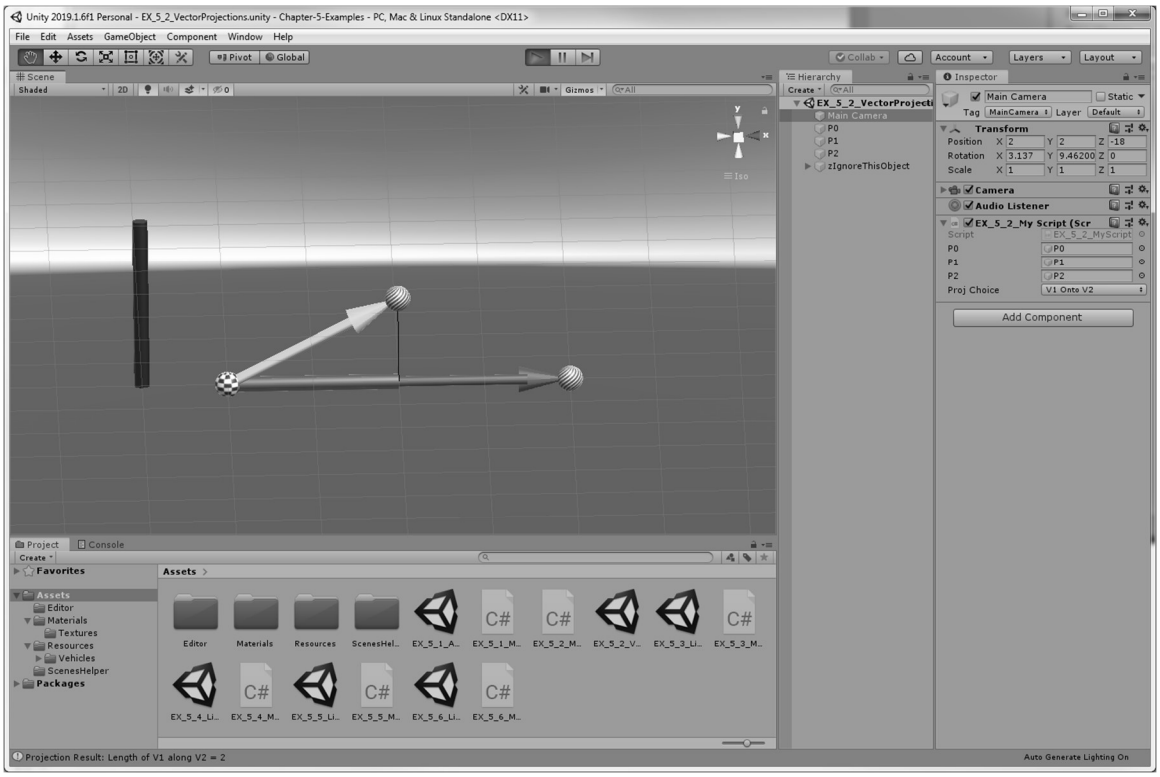

Figure 5-8. *Running the Vector Projections example*

The goals of this example are for you to:

- Appreciate the results of normalizing one of the vectors in the dot product operation
- Experience and understand the results of projecting vectors onto each other
- Observe and interact with negative projected distances
- Examine the code that performs vector projection

Examine the Scene

Take a look at the Example_5_2_VectorProjections scene and observe the predefined game objects in the Hierarchy Window. In addition to the MainCamera, there are three objects in this scene: P0, P1, and P2. As with the previous example in this chapter, you will manipulate the positions of the three game objects to define two vectors, $\vec{V}_1$ and $\vec{V}_2$,

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

$$\vec{V}_2 = P_2 - P_0$$

and examine the results of projecting these two vectors onto each other.

Analyze MainCamera MyScript Component

The MyScript component on MainCamera shows the references to the three game objects, the checkered sphere P0; two stripped spheres, P1, and P2; and a dropdown menu, ProjectionChoice. The dropdown menu allows the following options,

- V1 Onto V2: Project $\vec{V}_1$ vector onto $\vec{V}_2$
- V2 Onto V1: Project $\vec{V}_2$ vector onto $\vec{V}_1$
- Projection Off: do not perform any projection

Interact with the Example

Click on the Play Button to run the example. Take note that by default, the ProjectionChoice is set to V1OntoV2 and therefore MyScript is computing and displaying the results of projecting $\vec{V}_1$ onto $\vec{V}_2$.

Observe the two vectors, $\vec{V}_1$ and $\vec{V}_2$, that are defined by three positions. $\vec{V}_1$ is cyan and initially is above $\vec{V}_2$, which is magenta. Notice a light, semi-transparent cylinder along the $\vec{V}_2$ vector that is connected with a thin black line to the head of $\vec{V}_1$. The thin black line depicts the projection from the head of $\vec{V}_1$ perpendicularly onto $\vec{V}_2$, where the line intersects $\vec{V}_2$. The semi-transparent cylinder on $\vec{V}_2$ shows the projected length of $\vec{V}_1$ on $\vec{V}_2$. To emphasize the fact that the result of a dot product, or the projected length in this case, is just a floating-point number, this value is used to scale the height of the black bar to the side of the checkered sphere (P0). The length of the black bar is always the same as the semi-transparent cylinder. This length is the result of the calculated dot product, and in this scenario is,

$$v1LengthOnV2 = \vec{V}_1 \cdot \hat{V}_2 = \left\|\vec{V}_1\right\| \cos\theta$$

Now, select P1 and manipulate its x-component to change the length of $\vec{V}_1$. Notice that as $\vec{V}_1$ increases in length, the projected length on $\vec{V}_2$, the semi-transparent cylinder, also increases in length resulting in the black bar growing taller. This observation can be explained by the fact that the length of $\vec{V}_1$ is $\left\|\vec{V}_1\right\|$, and as $\left\|\vec{V}_1\right\|$ increases, so does $\left\|\vec{V}_1\right\| \cos\theta$, or v1LengthOnV2.

Now, select P2 and decrease the y-component value to increase the subtended angle. Observe that as the angle increases, the projected length of $\vec{V}_1$ decreases and when $\vec{V}_1$ and $\vec{V}_2$ become almost perpendicular, the length approaches zero. This observation can be explained by the fact that, as the angle θ increases, $\cos\theta$ decreases, and thus v1LengthOnV2 also decreases. When the two are perpendicular, $\cos\theta$ returns a value of 0, forcing $\|\vec{V}_1\|\cos\theta$ to be 0 as well, which is why no projection is visible when $\vec{V}_1$ and $\vec{V}_2$ are perpendicular. Beyond $90°$ and to $180°$, $\cos\theta$ is negative and thus the dot product result is negative. When this occurs, you will observe the black bar turning red and growing in the negative y-direction. Notice how the semi-transparent projection cylinder is no longer on $\vec{V}_2$, but extending in the opposite direction of $\vec{V}_2$. There are three important observations to make about the value of v1LengthOnV2,

- It is a simple floating-point number, this number is a measurement of the length of the projecting vector, $\vec{V}_1$, along the vector being projected onto, $\hat{V}_2$
- It is the sign of the number that indicates whether $\vec{V}_1$ and $\vec{V}_2$ are within $90°$ of each other
- Its magnitude is directly proportional to the length of the projecting vector, $\vec{V}_1$, and the cosine of the subtended angle with $\hat{V}_2$

It is important to remember the characteristics of the cosine function, that, its result decreases when the angle increases from $0°$ to $90°$. This means, as you have experienced and observed, that the magnitude of v1LengthOnV2 is actually inversely proportional to the angle θ for $0° < \theta < 90°$.

Feel free to choose the V2OntoV1 option for the ProjectionChoice variable and to examine and verify the complementary observations for,

$$v2LengthOnV1 = \hat{V}_1 \cdot \vec{V}_2 = \|\vec{V}_2\|\cos\theta$$

Details of MyScript

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

```
public enum ProjectionChoice {
    V1OntoV2,
    V2OntoV1,
    ProjectionOff
};

// Three positions to define two vectors: P0->P1, and P0->P2
public GameObject P0 = null;    // Position P0
public GameObject P1 = null;    // Position P1
public GameObject P2 = null;    // Position P2

public ProjectionChoice ProjChoice = ProjectionChoice.V1OntoV2;

#region For visualizing the vectors
#endregion
```

All the public variables for MyScript have been discussed when analyzing the MainCamera's MyScript component. Take note that variables with the enumerated datatype show up in the Hierarchy Window as options for a dropdown menu. The Start() function for MyScript is listed as follows.

```
void Start() {
    Debug.Assert(P0 != null);    // Verify proper setting in the editor
    Debug.Assert(P1 != null);
```

```
    Debug.Assert(P2 != null);

    #region For visualizing the vectors
    #endregion
}
```

As in all previous examples, the Debug.Assert() calls ensure proper setup regarding referencing the appropriate game objects via the Inspector Window. The Update() function is listed as follows.

```
void Update() {
    Vector3 v1 = P1.transform.localPosition - P0.transform.localPosition;
    Vector3 v2 = P2.transform.localPosition - P0.transform.localPosition;

    if ((v1.magnitude > float.Epsilon) &&
        (v2.magnitude > float.Epsilon))  {
        // make sure v1 and v2 are not zero vectors
         switch (ProjChoice) {
              case ProjectionChoice.V1OntoV2:
                   float v1LengthonV2 = Vector3.Dot(v1, v2.normalized);
                   Debug.Log("Projection Result: Length of V1 along V2 = " + v1LengthonV2);
                   break;
              case ProjectionChoice.V2OntoV1:
                   float v2LengthonV1 = Vector3.Dot(v1.normalized, v2);
                   Debug.Log("Projection Result: Length of V2 along V1 = " + v2LengthonV1);
                   break;
              default:
                   Debug.Log("Projection Result: no projection, dot=" + Vector3.Dot(v1, v2));
                   break;
         }
    }
    #region  For visualizing the vectors
    #endregion
}
```

The first two lines of the Update() function compute,

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

$$\vec{V}_2 = P_2 - P_0$$

The if condition checks and ensures that the normalization operation will not be performed on zero vectors. When conditions are favorable, the switch statement checks the user's projection choice and simply computes and prints the results of one of the followings,

$$v1LengthOnV2 = \vec{V}_1 \cdot \hat{V}_2$$

$$v2LengthOnV1 = \hat{V}_1 \cdot \vec{V}_2$$

Take Away from This Example

This example demonstrates the results of projecting vectors onto each other. Vector projection is computed when one of the two operands of a dot product operation is normalized. Remember, the normalized vector is the one being projected onto. It is important to remember that projection is a simple dot product operation and the result is a signed floating-point number.

Relevant mathematical concepts covered include:

- Calculating the dot product with a normalized vector can be interpreted as projecting the length of a vector onto another vector
- The sign of the projection result indicates if the subtended angle is less than, when positive, or more than, when negative, $90°$
- The projection result is directly proportional to the length of the projecting vector and inversely proportional to the subtended angle when the angle is between $0°$ and $90°$

Unity Tools:

- Enum datatype appears as dropdown menu options in the Hierarchy Window

EXERCISE

Verify the Vector Projection Formula

When computing v1LengthonV2 in MyScript,

```
float v1LengthonV2 = Vector3.Dot(v1, v2.normalized);
```

Verify the projection formula,

$$\vec{V}_1 \cdot \hat{V}_2 = \|\vec{V}_1\| \cos\theta$$

and replace that line with,

```
float cosTheta = Vector3.Dot(v1.normalize, v2.normalized);
float v1LengthonV2 = v1.magnitude * cosTheta;
```

verify that the runtime results are identical.

Representation of a Line Segment

Figure 5-9 shows two checkered sphere positions, P_0 and P_1, defining a vector, $\vec{V}_1$,

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

Notice that the region bounded by P_0 to P_1 is a segment of a straight line. In this case, the position P_a, when measured along the $\vec{V}_1$ direction, is located before the line segment, and position P_b is located after the line segment. In Figure 5-9, positions in between both P_0 and P_1 are described as inside the line segment and thus both P_a and P_b are both outside of the line segment.

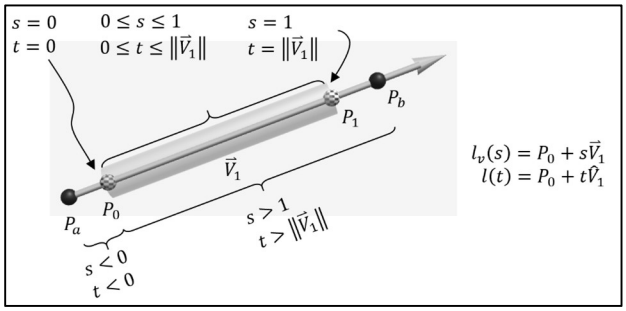

Figure 5-9. *Representing a line segment with a vector*

As you will see in later examples, in many applications it is critical to determine if a position is within the bounds of a line segment defined by two positions. By referencing the vector defined by the two positions, i.e., the $\vec{V}_1$ in Figure 5-9, there are two convenient ways to identify a line segment region. The first way is to represent a line segment based on parameterizing the vector $\vec{V}_1$,

$$l_v(s) = P_0 + s\vec{V}_1$$

As illustrated in Figure 5-9, the value of s identifies a position along the P_0 and P_1 line segment. For example,

$$l_v(0) = P_0 + 0\vec{V}_1 = P_0$$

$$l_v(0.5) = P_0 + 0.5\vec{V}_1 = \text{midpoint of the line segment}$$

$$l_v(1) = P_0 + 1\vec{V}_1 = P_0 + (P_1 - P_0) = P_1$$

In this way, the value of s is the portion, or percentage, of the line segment covered as measured from P_0 towards P_1, or, the portion of the line segment covered along the $\vec{V}_1$ direction starting from P_0. When there is no coverage, or when $s = 0$, the position identified is the beginning position of the line segment, P_0. A complete coverage, or when $s = 1$, is the position identified as the end position of the line segment, P_1. In general, as illustrated in Figure 5-9, a position is within the line segment boundaries when $0 \le s \le 1$. When $s < 0$, e.g., P_a, the position is before the beginning position, P_0, and when $s > 1$, e.g., P_b, the position is after the end position of the line segment, P_1.

The second way to represent the line segment region bounded by the positions P_0 and P_1 is by parameterizing the normalized $\vec{V}_1$, or $\hat{V}_1$,

$$l(t) = P_0 + t\hat{V}_1$$

In this case, because the vector is normalized, t is the measurement of the actual distance traveled from the beginning position, P_0, towards the end position of the line segment, P_1, or, the distance traveled along the $\hat{V}_1$ direction starting at P_0. For this reason, when $t = 0$, or $l(0)$, it signifies that no distance was traveled, and thus the identified position is the beginning of the line segment, P_0. The end position of the line segment is reached when $t = \|\vec{V}_1\|$, or the length of the vector $\vec{V}_1$,

$$l(\|\vec{V}_1\|) = P_0 + \|\vec{V}_1\|\hat{V}_1 = P_0 + \vec{V}_1 = P_1$$

As illustrated in Figure 5-9, only the range $0 \leq t \leq \|\vec{V}_1\|$ identifies a position within the line segment boundaries. $t < 0$ and $t > \|\vec{V}_1\|$ identify positions that are before the beginning position and after the end position of the line segment as measured along the $\hat{V}_1$ direction.

The only difference between the two line segments' representations is the normalization of the $\vec{V}_1$ vector,

$$l_v(s) = P_0 + s\vec{V}_1$$

$$l(t) = P_0 + t\hat{V}_1$$

When comparing these two representations, the 0 to 1 range of the s parameter in $l_v(s)$ is convenient for determining if a position is within the line segment bounds, and the distance measurement of the t parameter in $l(t)$ is advantageous when an actual distance traveled is required in the computation. Note that the s and t parameters are related by a simple scaling factor, $\|\vec{V}_1\|$,

$$t = s \times \|\vec{V}_1\|$$

In practice, when serving as part of more elaborate algorithmic computations, line segments are selfdom explicitly represented. In these situations, the $l_v(s)$ or $l(t)$ parameterizations are often used interchangeably depending on the needs of the computations.

When represented explicitly, a line segment is often referred to as a **Ray.** Rays are always parameterized as $l(t)$ with a normalized direction vector. For this reason, $l(t)$ is often referred to the **vector line equation**, or the ray equation, and is used often in video game development. For example, the Unity Ray class, https://docs.unity3d.com/ScriptReference/Ray.html, is a straightforward implementation of the line equation.

Inside-Outside Test of a General 1D Interval

Recall from Chapter 2 that a 1D interval is a region that is bounded by a minimum and maximum position along one of the major axes of the Cartesian Coordinate System. With the knowledge of vectors, the definition of an interval can now be relaxed. In general, a 1D interval, or a line segment, is defined as the region bounded by two positions along a direction (instead of just a major axis). In this way, the line segment in Figure 5-9 can be described as a 1D interval with its minimum position at $P_0$and its maximum position at P_1 along the $\vec{V}_1$ direction.

Figure 5-10 shows that the inside-outside test for an interval can be based on the comparison of coordinate values or the comparison of distances. Recall that given an interval defined along the Y-axis with min and max positions, a given y-value, v, is inside the interval when,

$$min \leq v \leq max$$

If the value min is subtracted from all sides of the equation,

$$min - min \leq v - min \leq max - min$$

then,

$$0 \leq (v - min) \leq (max - min)$$

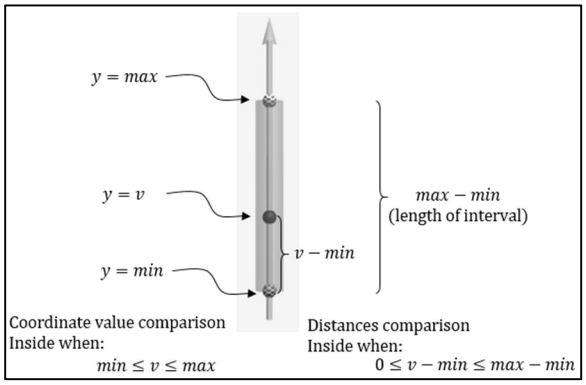

Figure 5-10. *Inside-Outside test based on coordinate values and distances*

This inequality equation says that the inside-outside test can also be determined by examining the distance from the minimum and maximum positions of the interval. For example, a given y-value, v, is inside the Y-axis interval when the distance between v to the minimum position is greater than zero and less than that of the maximum to minimum distance. With this understanding, Figure 5-11 illustrates the case for a general interval, with a direction that may not be aligned with a major axis of the Cartesian Coordinate System, like the Y-axis of Figure 5-10.

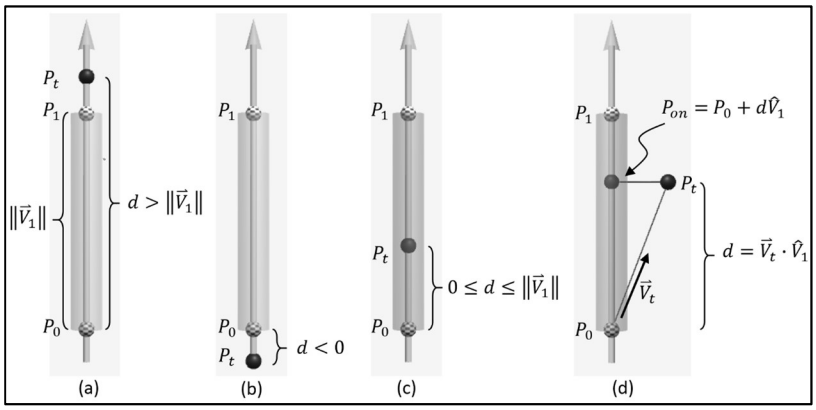

Figure 5-11. *An interval bounded by P_0 and P_1, or a line segment along the $\vec{V}_1$ direction*

With the knowledge of vectors, you can now define a vector, $\vec{V}_1$, with tail position at P_0, to represent the interval in Figure 5-11, where,

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

In this way, the interval is simply the line segment,

$$l(t) = P_0 + t\hat{V}_1$$

With the interval being described as a line segment, it should not be surprising that Figure 5-11 (a), (b), and (c), are similar to that of Figure 5-10. Figure 5-11 (a) and (b) illustrate the situation when the position to be tested, P_t, is outside of the line segment interval. Figure 5-11 (a) shows that, d, the symbol representing the distance between P_t and P_0 along the $\vec{V}_1$ direction, is larger than the line segment's magnitude, $d > \|\vec{V}_1\|$, and is thus beyond P_1. Figure 5-11 (b) shows the case when $d < 0$, or when P_t is before P_0. It is obvious that in both Figure 5-11 (a) and (b), P_t is outside of the interval. Figure 5-11 (c) shows that P_t is within the bounds of the interval when $0 \leq d \leq \|\vec{V}_1\|$. Note the similarities between these three cases with those of Figure 2-2, except, instead of the coordinate value comparisons, the inside-outside conditions are restated in Figure 5-11 based on distance measurements.

Figure 5-11 (d) is a more interesting case, here the position of interest, P_t, does not lie on the same line as the interval. You have addressed this type of situation in Chapter 2. You may recall that, when working with intervals along the Y-axis, the x- and z-component values are irrelevant when it comes to determining if a position is within a given Y-interval. For example, a given position, $(-3, 2, 5)$, is inside of the Y-axis interval with a bound of $min = -1$ and $max = 4$ because the y-component value of the position, 2, is bounded by the values of min, -1, and max, 4. In this case, the position $(-3, 2, 5)$ does not lie on the same line as the interval, the Y-axis, and only the coordinate value along the axis-direction of interest, the y-axis value of 2, is considered.

Figure 5-11 (d) translates the interval test knowledge from Chapter 2 using the vector projections you have learned. In this case, $\vec{V}_1$ is the vector from P_0 to P_1, and is the direction that corresponds to the Y-axis where the interval is defined. Given a position of interest, P_t, you can define the vector $\vec{V}_t$ as,

$$\vec{V}_t = P_t - P_0$$

then, the distance, d in all cases of Figure 5-11, is simply the projected distance of vector $\vec{V}_t$, in the $\vec{V}_1$ direction, or,

$$d = \vec{V}_t \cdot \hat{V}_1$$

Note that since $\vec{V}_t$ is projected onto the $\hat{V}_1$ direction, the vector $\vec{V}_1$ must be normalized. Finally, you know that the position, P_{on}, the projection of the P_t position onto $\vec{V}_1$ is, $t = d$ along the $l(t)$ line, or, d distance away from P_0 in the $\vec{V}_1$ direction,

$$P_{on} = l(d) = P_0 + d\hat{V}_1$$

■ **Note** *You can refer back to the initial discussion of vector projection in Figure 5-7. In this case, $\vec{V}_t$ is simply $\vec{V}_2$, and the projected length, d, is $\|\vec{V}_2\| \cos\theta$. When $d > \|\vec{V}_1\|$, the projected length is greater than the size of the vector being projected onto, and when $d < 0$, the subtended angle, θ, is more than $90°$.*

The Line Interval Bound Example

This example demonstrates the results of the inside-outside test for a general 1D interval (non-axis aligned interval). This example allows you to interactively define a general 1D interval and manipulate a test position to examine the results of performing the inside-outside test. Figure 5-12 shows a screenshot of running the EX_5_3_LineIntervalBound scene from the Chapter-5-Examples project.

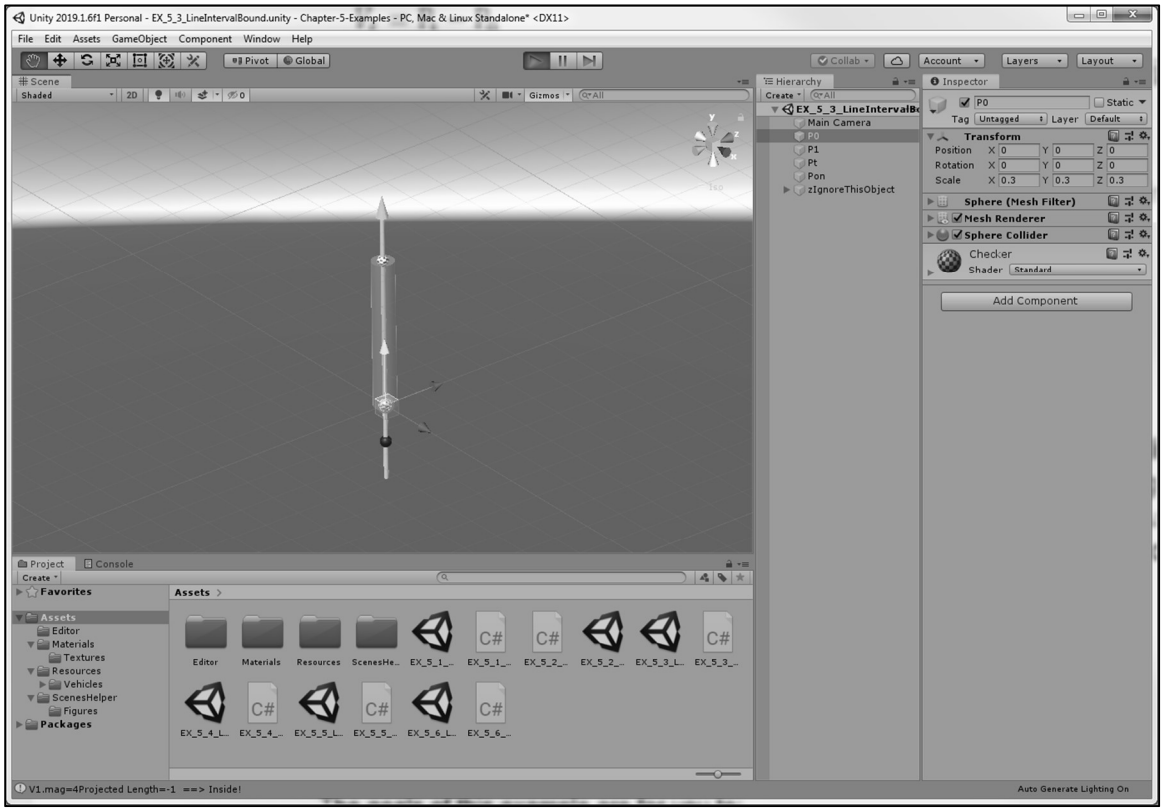

Figure 5-12. *Running the Line Interval Bound example*

The goals of this example are for you to:

- Experience defining and interacting with a general interval
- Examine the projection of a position onto a general interval
- Understand the implementation of an inside-outside test for the general interval

Examine the Scene

Take a look at the Example_5_3_LineIntervalBound scene and observe the predefined game objects in the Hierarchy Window. In addition to MainCamera, there are four objects in this scene: P0, P1, Pt, and Pon. Here, P0 and P1 are the bounds of the interval, Pt is the position to manipulate for the inside-outside test, and Pon represents the position when Pt is projected onto the interval.

Analyze MainCamera MyScript Component

The MyScript component on MainCamera shows four variables with names that correspond to the game objects in the scene. For all these variables, the transform.localPosition will be used for the manipulation of the corresponding positions.

Interact with the Example

Click on the Play Button to run the example. Observe that by default and design, this example is rather similar to the Interval Bounds in 1D example in Chapter 2. Select Pt and adjust its y-component value to move the position along the green line that defines the interval. Since Pt is on the green line, the projected position, Pon, is exactly the same as Pt. This is why you do not observe a separate projected position. Notice how the color of the interval changes if Pt is inside or outside of the interval. You can compare the interval color change to the debug messages printed in the Console Window and verify that proper inside-outside conditions are being computed. So far, this example has worked in exactly the same manner as the one from Chapter 2.

Now, adjust the x- and z-component values of Pt to move the test position away from the green line. Notice that as soon as Pt departs from the green line, you begin to observe the position Pon. You will also notice that Pon is connected to Pt by a thin black line that is perpendicular to the green line. Move the camera around to verify that the thin line connecting Pon to Pt is indeed perpendicular to the green line. You are observing the exact situation illustrated in Figure 5-11 (d).

Now, you can adjust P0 and P1 to manipulate the direction and length of the 1D interval. Observe that the perpendicular projection of Pon and the inside-outside test results are both consistently updated and correct for any interval you define.

Details of MyScript

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

```
// Positions: to deine the interval, the test, and projected
public GameObject P0 = null;    // Position P0
public GameObject P1 = null;    // Position P1
public GameObject Pt = null;    // Position Pt: test position
public GameObject Pon = null;   // Position Pon: position on the interval-line

#region For visualizing the vectors
#endregion

void Start() {
    Debug.Assert(P0 != null);    // Verify proper setting in the editor
    Debug.Assert(P1 != null);
    Debug.Assert(Pt != null);
    Debug.Assert(Pon != null);

    #region For visualizing the vectors
    #endregion
}
```

All the public variables for MyScript have been discussed when analyzing MainCamera's MyScript component, and as in all previous examples, the Debug.Assert() calls in the Start() function ensure proper setup regarding referencing the appropriate game objects via the Inspector Window. The Update() function is listed as follows.

```
void Update() {
    Vector3 v1 = P1.transform.localPosition - P0.transform.localPosition;

    if (v1.magnitude > float.Epsilon) {
        Vector3 vt = Pt.transform.localPosition - P0.transform.localPosition;
```

```
            Vector3 v1n = v1.normalized;
            float d = Vector3.Dot(vt, v1n);
            Pon.transform.localPosition = P0.transform.localPosition + d * v1.normalized;

            if ((d >= 0) && (d <= v1.magnitude))
                Debug.Log("V1.mag=" + v1.magnitude + "Projected Length=" + d + "  ==> Inside!");
            else
                Debug.Log("V1.mag=" + v1.magnitude + "Projected Length=" + d + "  ==> Outside!");
        }

        #region  For visualizing the vectors
        #endregion
}
```

The first line of the Update() function computes,

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

The if condition ensures that $\vec{V}_1$ is not a zero vector, which cannot be normalized or projected onto. If $\vec{V}_1$ is not a zero vector, then the four statements within the if condition perform the following four computations,

$$\vec{V}_t = P_t - P_0$$

$$\hat{V}_1 = Normalize(\vec{V}_1)$$

$$d = \vec{V}_t \cdot \hat{V}_1$$

$$P_{on} = P_0 + d\hat{V}_1$$

The Debug.Log() function prints the inside-outside status of Pt according to $0 \leq d \leq \|\vec{V}_1\|$. Note that although the interval is represented by the line equation,

$$l(t) = P_0 + t\hat{V}_1$$

this representation is implicit. There is no explicit data structure definition for a specific variable referencing the line equation. This implicit evaluation without explicit representation is rather typical in the application of the line equation.

Take Away from This Example

This example links the interval discussions in Chapter 2 to the concepts of vectors. At this point, you know how to compute the inside-outside test of a position for a general interval that is not aligned with a major axis. Recall the discussion in Chapter 2, where in Figure 2-7, the point in a bounding area test was derived by applying the one-dimensional interval test twice, once each to two intervals that are defined along two perpendicular directions. The same idea of applying the 1D interval test twice can be used for a general bounding area and, following the same concept once more, you can use the 1D interval test three times for a general bounding box. Now you can perform the inside-outside test of a position for a bounding box with three perpendicular intervals that do not need to be aligned with the major axes!

Though exciting, the non-axis aligned bounding box has a severe limitation, the collision computation between these boxes is straightforward only when the three corresponding intervals that define the boxes are parallel. In general, given two bounding boxes, each with different interval directions, the collision detection between two such boxes are complex and non-trivial. For this reason, only axis-aligned bounding boxes are typically used in video game development.

Relevant mathematical concepts covered include:

- An interval along a direction is a line segment and can be represented by the vector line equation

- Vector projection can be applied to compute the projected distance of a point along a direction
- The projected position along a direction can be determined for any given position

EXERCISES

Verify the Axis-Aligned Interval Discussion with Vectors

Recall that the Y-axis interval is defined by its min and max values. These are actually P_0 with $(0, min, 0)$, and P_1 with $(0, max, 0)$. Now, by computing

$$\vec{V}_t = P_t - P_0$$
$$d = \vec{V}_t \cdot \hat{V}_1$$
$$P_{on} = P_0 + d\hat{V}_1$$

verify that given a general test position, P_t with (x_t, y_t, z_t), the projected position, P_{on}, is $(0, y_t - min, 0)$, showing that, in this case, the x- and z-component values of P_t are indeed irrelevant. You can setup the values of P0 and P1 in this example to visualize the described results.

Verify the P_{on} Position

Define $\vec{V}_{on}$ to be,

$$\vec{V}_{on} = d\hat{V}_1 - \vec{V}_t$$

And observe that,

$$P_{on} = P_t + \vec{V}_{on}$$

Modify MyScript to print out P_{on} values based on these equations and then compare them to the Pon values currently computed in the script to verify they are identical. Notice that, $\vec{V}_{on}$ is also, $P_{on} - P_t$.

Verify that Vector Projection is Perpendicular

Refer to the previous definition of $\vec{V}_{on}$,

$$\vec{V}_{on} = P_{on} - P_t$$

Since P_{on} is the projection of P_t onto $\hat{V}_1$, it follows that $\vec{V}_{on}$ is perpendicular to $\vec{V}_1$. Recall from the discussion of the dot product, that, when vectors have a subtended angle of $90°$, and because $\cos 90° = 0$, the dot product of two such vectors is zero. Modify MyScript to compute and print out the values of $\vec{V}_{on} \cdot \vec{V}_1$ and $\vec{V}_{on} \cdot \hat{V}_1$ and verify that both results are zero.

Line to Point Distance

Imagine an adventure game where hidden treasures are exposed when exploration agents are within their proximity. By now, you know how to define bounding volumes, e.g., bounding spheres, for both the treasure and the agent objects, as well as support the detection of collisions between these corresponding bounding volumes. Figure 5-13 illustrates that for a fast-moving agent, the simple bounding sphere collision test may result in missed treasures.

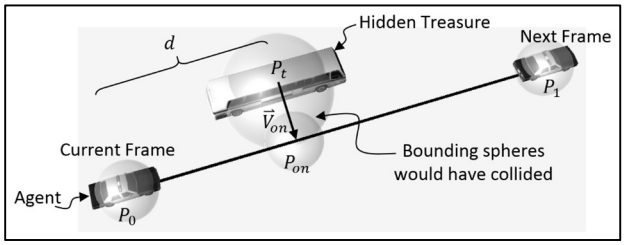

Figure 5-13. *A case where the bounding sphere misses with fast traveling objects*

In Figure 5-13, both the police car agent and the city bus treasure are bounded by their corresponding bounding spheres. In this case, the police car is traveling at a high speed along the velocity defined by the black line. Here, in one update the car traveled from position P_0 on the left to position P_1 on the right. Notice that the bounding spheres of the car and the bus would have collided around P_{on} if the police car was traveling at a much slower speed. However, at the described high speed, the bounding sphere collisions at both the current frame and the next frame would be false, thereby missing the police car (agent) and the city bus (treasure) collision.

A straightforward approach to address this problem is by modeling this situation as a line to point distance computation. In the case of Figure 5-13, the problem is to find the closest distance between the line segment defined by P_0 and P_1, and the point located at P_t. This distance would be used to compare against the radii of the bounding spheres of the agent and the treasure to determine if a collision should occur during the agent's motion. If this distance is less than the combined radii, then a collision should occur.

From basic geometry, you know that the closest distance between a line segment and a position should be measured along a direction that is perpendicular from the line to the position. Now, you also know that a vector projection projects the head of

a vector perpendicularly onto another given vector. In the case of Figure 5-13, these observations can be translated into, defining two vectors,

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

$$\vec{V}_t = P_t - P_0$$

Then you can project $\vec{V}_t$ onto $\vec{V}_1$ to compute P_{on}, the projection of P_t on the vector $\vec{V}_1$. In this case, you know the vector, $\vec{V}_{on}$,

$$\vec{V}_{on} = P_{on} - P_t$$

must be perpendicular to $\vec{V}_1$ and thus the distance between P_t and P_{on}, or $\|\vec{V}_{on}\|$, is the closest distance between the line segment defined by P_0, P_1 and the position P_t. This distance would be compared with the combined radii of the bounding spheres of the agent and treasure for collision determination.

It is encouraging that this problem and its solution are familiar to those of the line segment and the general interval inside-outside test. Based on the previous discussions you know that,

$$d = \vec{V}_t \cdot \hat{V}_1$$

$$P_{on} = P_0 + d\hat{V}_1$$

You can observe that when the position P_t is within the bounds of the line segment end points, or when $0 \leq d \leq \|\vec{V}_1\|$, the closest distance between the line segment and the point is from P_t to P_{on}, or the magnitude of $\vec{V}_{on}$ or, $\|\vec{V}_{on}\|$.

Figure 5-14 (a) and (b) show that P_t can also be outside of the line interval. In these cases, the closest distance measurements are actually between P_t and the end points of the line segment. Figure 5-14 (a) illustrates that when $d < 0$, P_t is located at a position before the line segment and thus the closest distance is actually the distance between P_t and P_0, or simply the magnitude $\vec{V}_t$ or, $\|\vec{V}_t\|$. Figure 5-14 (b) illustrates that when $d > \|\vec{V}_1\|$, P_t is located at a position after the line segment and thus the closest distance is the distance between P_t and P_1, or the magnitude of $(\overrightarrow{P_t - P_1})$ or, $\|\overrightarrow{P_t - P_1}\|$. Figure 5-14 (c) is the same case as Figure 5-13, when $0 \leq d \leq \|\vec{V}_1\|$, and, the closest distance is the magnitude of $\vec{V}_{on}$ or, $\|\vec{V}_{on}\|$.

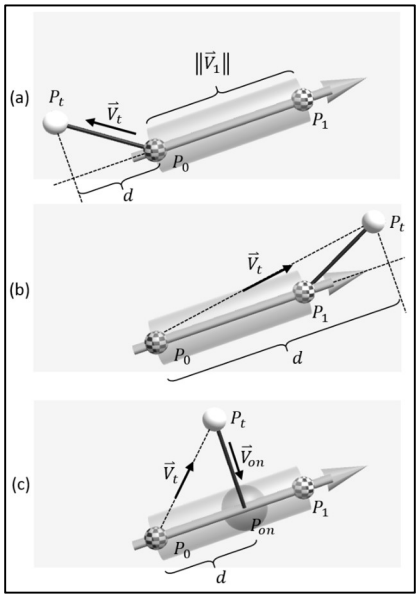

Figure 5-14. *The three conditions of line to point distance calculation*

The Line to Point Distance Example

This example demonstrates the results of the line to point distance computation. This example allows you to interactively define the line segment, manipulate the position of, and examine the results from the line to point distance computation. Figure 5-15 shows a screenshot of running the EX_5_4_LineToPointDistance scene from the Chapter-5-Examples project.

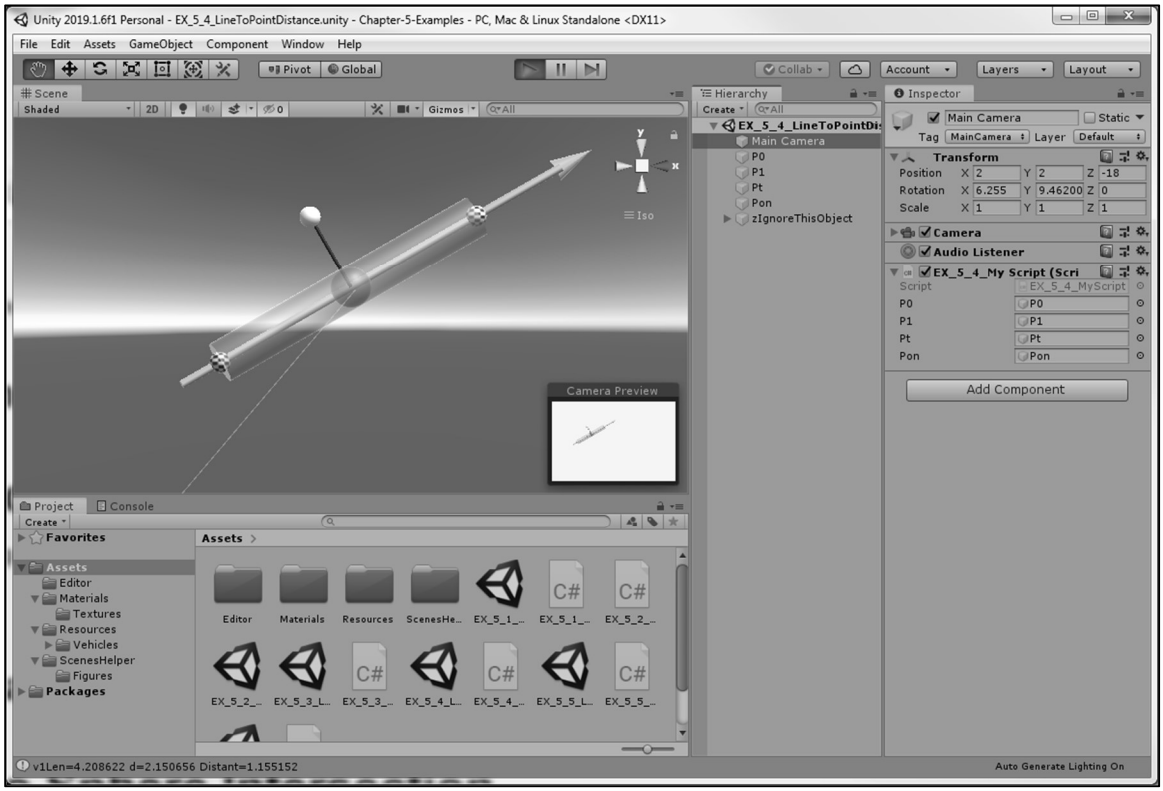

Figure 5-15. *Running the Line To Point Distance example*

The goals of this example are for you to:

- Experience working with a straightforward application of the vector dot product concepts
- Interact with and understand the results of line to point distance computation
- Examine the implementation of line to point distance computation

Examine the Scene

Take a look at the Example_5_4_LineToPointDistance scene and observe the predefined game objects in the Hierarchy Window. In addition to the MainCamera, there are the exact same four objects in this scene as in the previous example: P0, P1, Pt, and Pon. Here, P0 and P1 are the checkered spheres that identify the line segment. Pt is the white sphere and is the position (the point) used for the line to point distance computation. Finally, Pon, the red sphere, is the position where Pt is projected onto the line.

Analyze MainCamera MyScript Component

The MyScript component on MainCamera shows four variables with names that correspond to the game objects in the scene. For all these variables, the transform.localPosition will be used for the manipulation of the corresponding positions.

Interact with the Example

Click on the Play Button to run the example. Observe that P0 and P1 define the green vector direction and a line segment. There is a thin black line connecting Pt, the white sphere, to the projected position, Pon, the red sphere, on the line segment. Select Pt and adjust its y-component value. Try to move Pt away from the line, e.g., by increasing the y-component value, and observe the red sphere increase in size. If you move Pt closer instead, you will observe the red sphere shrink. The size of the red sphere, Pon, is directly proportional to the distance between Pt and the line segment. The results of this computation can also be observed in the Console Window.

Now, change the x-component value of Pt to observe the corresponding movement of the projection position, Pon. Notice that when Pt is within the bounds of the line segment, the thin black line connecting Pt to Pon is always perpendicular to the line segment, indicating the projection of Pt onto the line segment. When Pt is moved to outside of the line segment, the thin black line becomes connected to the closest end point of the line segment, either P0 or P1. This signifies that the closest distance in these situations is actually the measurement to one of the end points of the line segment.

You can now select and manipulate P0 and P1 to verify that the distance computation is indeed correct for any line segment, including a line segment defined by the zero vector, which occurs when P0 and P1 are located at the same position.

Details of MyScript

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

```
// Positions: to deine the interval, the test, and projected
public GameObject P0 = null;  // Position P0
public GameObject P1 = null;  // Position P1
public GameObject Pt = null;   // Position for distance computation
public GameObject Pon = null;  // closest point on line

#region For visualizing the line
#endregion

// Start is called before the first frame update
void Start() {
    Debug.Assert(P0 != null);    // Verify proper setting in the editor
    Debug.Assert(P1 != null);
    Debug.Assert(Pt != null);

    #region For visualizing the lines
    #endregion
}
```

All the public variables for MyScript have been discussed when analyzing MainCamera's MyScript component, and as in all previous examples, the Debug.Assert() calls in the Start() function ensure proper setup regarding referencing the appropriate game objects via the Inspector Window. The Update() function is listed as follows.

```
void Update() {
    float distance = 0;  // closest distance
    Vector3 v1 = P1.transform.localPosition - P0.transform.localPosition;
```

```
        float v1Len = v1.magnitude;

        if (v1Len > float.Epsilon) {
            Vector3 vt = Pt.transform.localPosition - P0.transform.localPosition;
            Vector3 v1n = (1f / v1Len) * v1; // <<-- what is going on here?
            float d = Vector3.Dot(vt, v1n);
            if (d < 0) {
                Pon.transform.localPosition = P0.transform.localPosition;
                distance = vt.magnitude;
            } else if (d > v1Len) {
                Pon.transform.localPosition = P1.transform.localPosition;
                distance = (Pt.transform.localPosition - P1.transform.localPosition).magnitude;
            } else {
                Pon.transform.localPosition = P0.transform.localPosition + d * v1n;
                Vector3 von = Pon.transform.localPosition - Pt.transform.localPosition;
                distance = von.magnitude;
            }
            float s = distance * kScaleFactor;
            Pon.transform.localScale = new Vector3(s, s, s);
            Debug.Log("v1Len=" + v1Len + " d=" + d + " Distance=" + distance);
        }

        #region  For visualizing the lines
        #endregion
    }
```

The first two lines of the $\mathrm{Update}()$ function compute,

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

$$v1Len = \left\|\vec{V}_1\right\|$$

The if condition checks for and avoids performing the normalization operation on a zero vector. When the condition is favorable, the following are computed,

$$\vec{V}_t = P_t - P_0$$

$$\hat{V}_1 = \frac{1}{v1Len}\vec{V}_1$$ Note that this is simply normalizing the $\vec{V}_1$ vector

$$d = \vec{V}_t \cdot \hat{V}_1$$

With P_{on} being the closet point on the line segment and the position being $distance$ away from P_t, notice how the computation is governed by the values of the projected length, d,

- When $d < 0$, the condition is as illustrated in Figure 5-14 (a), and,

  $$P_{on} = P_0$$

  $$distance = \left\|\vec{V}_t\right\|$$

- When $d > v1Len$, or, $d > \left\|\vec{V}_1\right\|$, the condition is as illustrated in Figure 5-14 (b), and,

  $$P_{on} = P_1$$

  $$distance = \left\|\overrightarrow{P_t - P_1}\right\|$$

- The final condition, when $0 \leq d \leq \left\|\vec{V}_1\right\|$ the condition is as illustrated in Figure 5-14 (c), and,

$$P_{on} = P_0 + d\hat{V}_1$$
$$\vec{V}_{on} = P_{on} - P_t$$
$$distance = \|\vec{V}_{on}\|$$

The last three lines of code scales the red sphere that represents P_{on} in proportion to the value of $distance$ and outputs the computation results to the Console Window.

Take Away from This Example

This example demonstrates a solution to a fundamental problem in video games and interactive computer graphics. In video games, closest distance and intersection computations are some of the most straightforward solutions to the problem of missed collisions from fast moving objects. In graphical interactions, many basic operations depend on the results of line to point distance computation. For example, in a drawing editor, clicking on the mouse button to select a line object is typically implemented as determining if the clicked position is sufficiently close to the line object, as clicking perfectly on a one-pixel wide line can be challenging and frustrating!

The solution presented in this example to these types of problems is based on the concepts of vector projection and builds directly on the knowledge gained from the line equation and the general interval inside-outside test discussions. These concepts are some of the most important topics in interactive graphic applications and are widely applied in video game development.

Relevant mathematical concepts covered include:

- The distance between a line segment and a point, P_t, can be solved by finding the position, P_{on}, along the line segment that is closest to P_t, and computing the distance between P_{on} and P_t
- When P_t is outside of the line segment, P_{on} is located at one of the line segment end points
- When P_t is inside the line segment, P_{on} is the projection of P_t onto the line segment

EXERCISE

Experience Solve the Missing Collision Problem

Modify MyScript to continuously send a fast-moving agent from P0 to P1, e.g., traveling at a speed of 20 units per second. You can refer to the EX_4_3_VelocityAndAiming scene of Chapter-4-Examples for a sample approach of how to implement this functionality. In your Update() function, compute the collision between the agent and the Pt sphere. Notice, even when the P0 to P1 line segment passes right through the Pt sphere, you can fail to detect the collision between the agent and the Pt sphere. This is because the agent is simply

moving too fast for the spheres to overlap. Verify you can resolve this problem with the line to point distance computation.

Line to Line Distance

Imagine in another adventure game, you want to know if the path of the explorer will come too close to a monster pathway. This is a simple case of determining the distance between two line segments. This problem has a simple and elegant solution that allows you to practice the vector algebra learned. Figure 5-16 illustrates the general case of two line segments, where the problem is how to compute the perpendicular, or the shortest, distance between the lines.

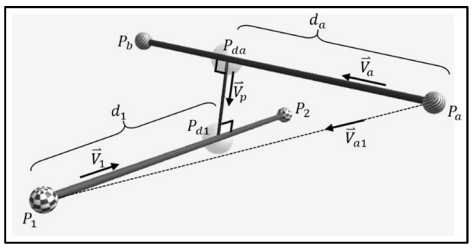

Figure 5-16. *Distance between two line segments*

The problem of finding the closest, or the perpendicular, distance between two given lines is similar to the line to point distance problem. The solution boils down to locating a point on each line where, when connected are perpendicular to both of the two given lines. This description is depicted in Figure 5-16, where the two lines are defined by positions P_1 and P_2, and P_a and P_b respectively. In this figure, the position P_{d1} is d_1 distance away from P_1, and P_{da} is d_a distance away from P_a where the line segment from P_{d1} to P_{da} is perpendicular to both of the other two lines. In this way, the shortest distance between the lines is the length of the vector, $P_{d1} - P_{da}$. In order to find P_{d1} and P_{da}, the task is to find the distances d_1 and d_a. You can begin deriving the solution by defining,

$$\vec{V}_1 = P_2 - P_1$$

$$\vec{V}_a = P_b - P_a$$

$$\vec{V}_p = P_{d1} - P_{da}$$

The descriptions of P_{d1} and P_{da} can be formulated as two separate line segments,

$$P_{d1} = P_1 + d_1\hat{V}_1$$

$$P_{da} = P_a + d_a\hat{V}_a$$

Since $\vec{V}_p$ is perpendicular to both $\vec{V}_1$ and $\vec{V}_a$, it must be true that both of the following are true,

$$\hat{V}_1 \cdot \vec{V}_p = 0$$

$$\hat{V}_a \cdot \vec{V}_p = 0$$

Now, if you substitute P_{d1} and P_{da} into $\vec{V}_p$, these two equations become,

$$\hat{V}_1 \cdot \vec{V}_p = \hat{V}_1 \cdot (P_{d1} - P_{da}) = \hat{V}_1 \cdot (P_1 + d_1\hat{V}_1 - P_a - d_a\hat{V}_a) = 0$$

$$\hat{V}_a \cdot \vec{V}_p = \hat{V}_a \cdot (P_{d1} - P_{da}) = \hat{V}_a \cdot (P_1 + d_1\hat{V}_1 - P_a - d_a\hat{V}_a) = 0$$

Note that these are two simultaneous equations with two unknowns, d_1 and d_a. Now, examine the first of the two equations, by following the distributive property of dot product over vector operations, collecting the terms with $\hat{V}_1$, and recognizing $\hat{V}_1 \cdot \hat{V}_1$ is 1.0,

$$\begin{aligned}
\hat{V}_1 \cdot \vec{V}_p &= \hat{V}_1 \cdot (P_{d1} - P_{da}) \\
&= \hat{V}_1 \cdot (P_1 + d_1\hat{V}_1 - P_a - d_a\hat{V}_a) && \text{Substitute the definitions of } P_{d1} \text{ and } P_{da} \\
&= \hat{V}_1 \cdot P_1 + \hat{V}_1 \cdot d_1\hat{V}_1 - \hat{V}_1 \cdot P_a - \hat{V}_1 \cdot d_a\hat{V}_a && \text{Distributive property of dot product} \\
&= \hat{V}_1 \cdot (P_1 - P_a) + \hat{V}_1 \cdot d_1\hat{V}_1 - \hat{V}_1 \cdot d_a\hat{V}_a && \text{Collect the } P_1 \text{ and } P_a \text{ terms} \\
&= \hat{V}_1 \cdot (P_1 - P_a) + d_1(\hat{V}_1 \cdot \hat{V}_1) - d_a(\hat{V}_1 \cdot \hat{V}_a) && \text{Distributive property over factors } d_1 \text{ and } d_a \\
&= \hat{V}_1 \cdot (P_1 - P_a) + d_1 - d_a(\hat{V}_1 \cdot \hat{V}_a) && \hat{V}_1 \text{ dot } \hat{V}_1 \text{ is equal to } 1
\end{aligned}$$

Now, let

$$d = \hat{V}_1 \cdot \hat{V}_a$$

$$\vec{V}_{a1} = P_1 - P_a$$

Then,

$$\hat{V}_1 \cdot \vec{V}_p = \hat{V}_1 \cdot \hat{V}_{a1} + d_1 - d_a d = 0$$

Following similar simplification steps, left as an exercise, you can show that,

$$\hat{V}_a \cdot \vec{V}_p = -\hat{V}_a \cdot \hat{V}_{a1} - d_a + d_1 d = 0$$

In this way, the simultaneous equations become,

$$\hat{V}_1 \cdot \hat{V}_{a1} + d_1 - d_a d = 0$$

$$-\hat{V}_a \cdot \hat{V}_{a1} - d_a + d_1 d = 0$$

Recall that dot product results are floating-point numbers, therefore, $\hat{V}_1 \cdot \hat{V}_{a1}$ and $\hat{V}_a \cdot \hat{V}_{a1}$ return simple floating-point numbers. These equations are thus simple algebraic equations that are independent from vector operations, and, once again their simplification and solution derivation are left as an exercise. You can show that the solution to the simultaneous equations is,

$$d_1 = \frac{-(\hat{V}_1 \cdot \vec{V}_{a1}) + d(\hat{V}_a \cdot \vec{V}_{a1})}{1 - d^2}$$

$$d_a = \frac{(\hat{V}_a \cdot \vec{V}_{a1}) - d(\hat{V}_1 \cdot \vec{V}_{a1})}{1 - d^2}$$

In this case, to allow easier interpretation of text output, instead of distances you can compute the portion of line segment covered or,

$$d_1' = \frac{d_1}{\|\vec{V}_1\|}$$

$$d'_a = \frac{d_a}{\|\vec{V}_a\|}$$

and,

$$P_{d1} = P_1 + d'_1\vec{V}_1$$

$$P_{da} = P_a + d'_a\vec{V}_a$$

where, you know P_{d1} and P_{da} are within the bounds of their respective line segments only when d'_1 and d'_a are both within the range of 0 to 1. Now, the closest distance between the two lines is the distance between P_{d1} and P_{da}, or, $\|\overrightarrow{P_{d1} - P_{da}}\|$. Note that this is also the length of the vector $\vec{V}_p$, or $\|\vec{V}_p\|$.

The Line to Line Distance Example

This example demonstrates the results of line to line distance computation. This example allows you to interactively define the two line segments and examine the results from the line to line distance computation. Figure 5-17 shows a screenshot of running the EX_5_5_LineToLineDistance scene from the Chapter-5-Examples project.

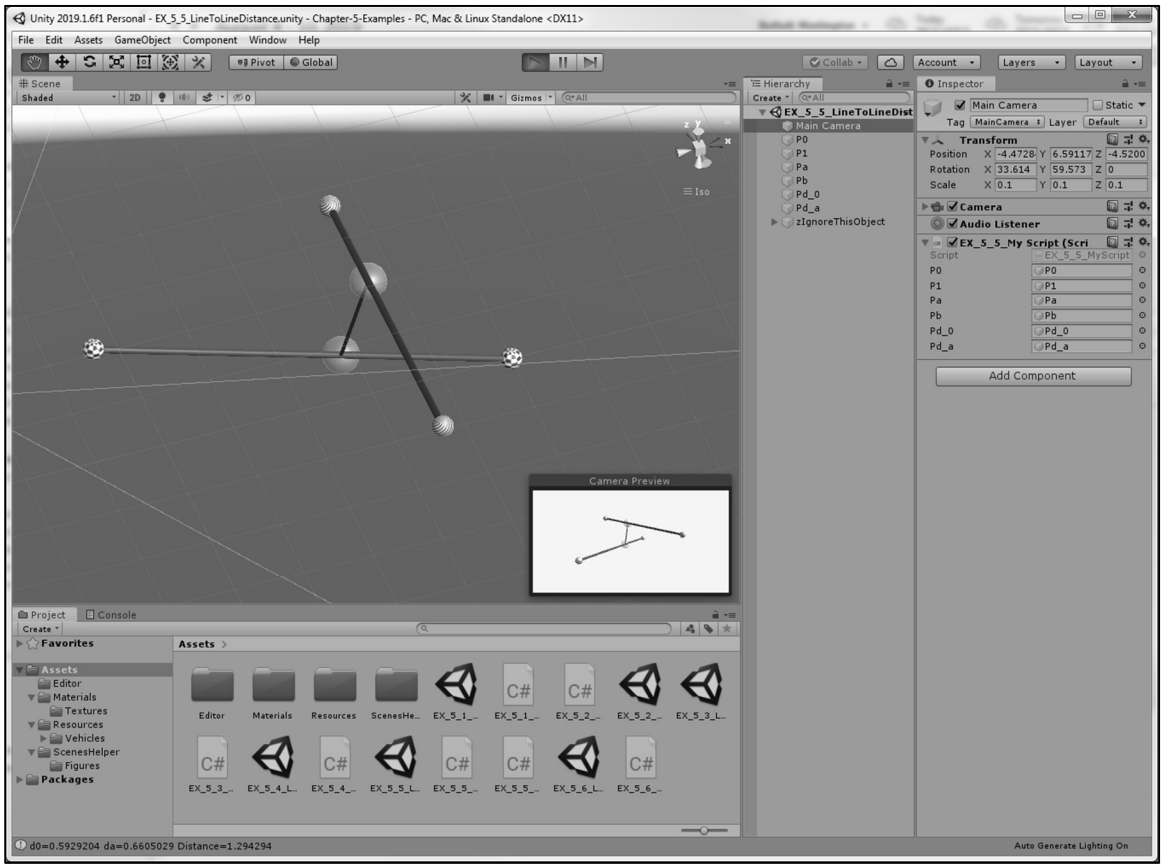

Figure 5-17. *Running the Line to Line Distance example*

The goals of this example are for you to:

- Experience deriving and simplifying non-trivial vector expressions
- Verify solutions to vector equations with a straightforward implementation
- Examine the implementation of line to line distance computation

Examine the Scene

Take a look at the Example_5_5_LineToLineDistance scene and observe the predefined game objects in the Hierarchy Window. In addition to the MainCamera, there are three sets of objects defined for the visualization of the two line segments: the two checkered spheres P1, P2, the two stripped spheres Pa, Pb, and the two solid color spheres Pd_1 and Pd_a. The transform.localPosition of P1, P2 and Pa, Pb define the bounding positions of the two line segments. The transform.localPosition of Pd_1 is a position along the line defined by P1 to P2, and Pd_a a position along the Pa to Pb line where the distance from Pd_1 to Pd_a is the closest distance between the two lines.

Analyze MainCamera MyScript Component

The MyScript component on MainCamera shows six variables with names that correspond to the game objects in the scene. These variables are setup to reference the game objects with the corresponding names in the scene.

Interact with the Example

Click on the Play Button to run the example. Once running, you will observe two line segments. The first is red and is defined by a pair of checkered spheres, P1 and P2. The second line segment is blue and is defined by a pair of stripped spheres, Pa and Pb. Along each line segment is a semi-transparent sphere, Pd_1 on the red line segment and Pd_a on the blue line segment. Notice that the two spheres are connected by a thin black line that is perpendicular to both the red and the blue line segments. You are observing the solution to the line to line distance computation.

Now, rotate the Scene View camera to verify that the thin black line is indeed perpendicular to both the red and blue line segments. Feel free to manipulate any of the line segment end points to verify the computation results. Note that when the locations of Pd_1 or Pd_a are outside of their respective line segments, the semi-transparent spheres will turn opaque. You can also observe the text output in the Console Window. There, the values for d1 and da are in the range between 0 to 1, assisting your verification of the corresponding position's inside-outside status on their respective line segment.

Lastly, set both of the line segments to be along the same direction, e.g., set P1 and P2 to the values $(0, 0, 0)$ and $(5, 0, 0)$; and Pa and Pb to $(0, 2, 0)$, and $(5, 2, 0)$. Once done, notice that the results of both Pd_1 and Pd_a are no longer visualized. You can verify in the Console Window that the line segments are in the exact same direction. This is a special case not handled in the derived solution. One of the exercises at the end of this example will tell you what this special case is and allow you to practice handling this special case.

Details of MyScript

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

```
public GameObject P1, P2;  // define the line V1
public GameObject Pa, Pb;  // define the line Va
```

```
public GameObject Pd_1;      // point on V1 closest to Va
public GameObject Pd_a;      // point on va closest to V1

#region For visualizing the line
#endregion

void Start() {
     Debug.Assert(P1 != null);    // Verify proper setting in the editor
     Debug.Assert(P2 != null);
     Debug.Assert(Pd_1 != null);
     Debug.Assert(Pa != null);
     Debug.Assert(Pb != null);
     Debug.Assert(Pd_a != null);

     #region For visualizing the line
     #endregion
}
```

All the public variables for MyScript have been discussed when analyzing MainCamera's MyScript component, and as in all previous examples, the Debug.Assert() calls in the Start() function ensure proper setup regarding referencing the appropriate game objects via the Inspector Window. The Update() function is listed as follows.

```
void Update() {
     Vector3 v1 = (P2.transform.localPosition - P1.transform.localPosition);
     Vector3 va = (Pb.transform.localPosition - Pa.transform.localPosition);

     if ((v1.magnitude < float.Epsilon) || (va.magnitude < float.Epsilon))
          return;  // will only work with well defined line segments

     Vector3 va1 = P1.transform.localPosition - Pa.transform.localPosition;
     Vector3 v1n = v1.normalized;
     Vector3 van = va.normalized;
     float d = Vector3.Dot(v1n, van);

     bool almostParallel = (1f - Mathf.Abs(d) < float.Epsilon);

     float d1 = 0f, da = 0f;

     if (!almostParallel) {  // two lines are not parallel
          float dot1A1 = Vector3.Dot(v1n, va1);
          float dotAA1 = Vector3.Dot(van, va1);

          d1 = (-dot1A1 + d * dotAA1) / (1 - (d * d));
          da = (dotAA1 - d * dot1A1) / (1 - (d * d));

          d1 /= v1.magnitude;
          da /= va.magnitude;

          Pd_1.transform.localPosition = P1.transform.localPosition + d1 * v1;
          Pd_a.transform.localPosition = Pa.transform.localPosition + da * va;
          float dist = (Pd_1.transform.localPosition - Pd_a.transform.localPosition).magnitude;
          Debug.Log("d1=" + d1 + " da=" + da + " Distance=" + dist);
     } else {
          Debug.Log("Line segments are parallel, special case not handled");
     }

     #region  For visualizing the line
```

```
        #endregion
}
```

The first two lines of the $\mathrm{Update}()$ function compute,

$$\vec{V}_1 = P_2 - P_1$$

$$\vec{V}_a = P_b - P_a$$

The code then ensures that both are not zero vectors, and continues to compute,

$$\vec{V}_{a1} = P_1 - P_a$$

$\hat{V}_1$ and $\hat{V}_a$

$$d = \hat{V}_1 \cdot \hat{V}_a$$

Recall that the dot product of two normalized vectors is the cosine of the subtended angle, and that the cosine of $0°$ or $180°$ is equal to 1 and -1 respectively. For this reason, the $\mathrm{almostParallel}$ variable is true when $\hat{V}_1$ and $\hat{V}_a$ are almost parallel. In the implementation, the computation only proceeds when the two directions are not almost parallel. This check is necessary because the solutions for both $d_1$and d_a involve a division by $1 - d^2$, and when the two directions are almost parallel, $d \approx 1.0$, which means $d_1$and d_a will be divided by 0, thus causing neither $d_1$nor d_a to be defined. When the two lines are not parallel, the code computes,

$$dot1A1 = \hat{V}_1 \cdot \vec{V}_{a1}$$

$$dotAA1 = \hat{V}_a \cdot \vec{V}_{a1}$$

and

$$d_1 = \frac{-dot1 \quad + d*dotAA}{1-d^2}$$

$$d_a = \frac{dotA \quad - d*dot1A}{1-d^2}$$

where notice that both d_1 and d_a are scaled to values between 0 and 1 for positions that are inside the respective line segments, and closest positions are computed accordingly,

$$P_{d1} = P_1 + d_1\vec{V}_1$$

$$P_{da} = P_a + d_a\vec{V}_a$$

And lastly, the closest distance between the two lines is simply the distance between the closest positions,

$$dist = \left\|\overrightarrow{P_{d1} - P_{da}}\right\|$$

Take Away from This Example

Though the presented solution of the line to line distance is interesting, it is incomplete. First of all, the solution does not address the situation when the line segments are parallel. Secondly, the solution does not address the situations when the closest points are outside of the given line segments, i.e., when either P_{d1} or P_{da} or both are outside of their corresponding line segments. As in the case of line to point distance, when the closest position is outside of the line segment, the closest distance should be measured to the corresponding end position of the line segment. Although not a complete solution, this example does demonstrate and allow you to practice simplifying vector equations based on the learned vector algebra and serves as a way to illustrate an implementation of a typical solution to vector equations.

Through working with this example, you have observed that the actual vector equations and their solution process may be complex and involved. However, thankfully, as you have also witnessed, the derived solutions are typically elegant and can be implemented in a straightforward fashion with a relatively small number of steps. To ensure proper implementation, it is essential to maintain precise drawings and notes with symbols that correspond to variable names. Lastly, and very importantly, attention must be maintained when working with normalized vs non-normalized vectors.

Relevant mathematical concepts covered include:

- Vector algebra, or the rules governing vector operations, are invaluable in simplifying non-trivial vector equations

Relevant observations on implementation include:

- It is vital to understand and check for situations when mathematic expressions are undefined, e.g., normalization of zero vectors, or, divisions by 0.
- It is often possible to relate mathematic expressions to real-world geometric orientations. For example, you know that the dot product, $\hat{V}_1 \cdot \hat{V}_a$, computes the cosine of the angle subtended by two vectors, therefore a value of 1 or -1 means the vectors are parallel. It is the responsibility of the software developer to understand these implications and ensure all appropriate conditions are considered and supported.

EXERCISES

Verify the Solutions for d_1 and d_a

The derived simultaneous equations for line to line distance are,

$$\hat{V}_1 \cdot (P_1 + d_1\hat{V}_1 - P_a - d_a\hat{V}_a) = 0$$
$$\hat{V}_a \cdot (P_1 + d_1\hat{V}_1 - P_a - d_a\hat{V}_a) = 0$$

You know,

$$\vec{V}_1 = P_2 - P_1$$
$$\vec{V}_a = P_b - P_a$$
$$\vec{V}_{a1} = P_1 - P_a$$
$$d = \hat{V}_1 \cdot \hat{V}_a$$

Now, show that,

$$d_1 = \frac{-(\hat{V}_1 \cdot \vec{V}_{a1}) + d(\hat{V}_a \cdot \vec{V}_{a1})}{1 - d^2}$$
$$d_a = \frac{(\hat{V}_a \cdot \vec{V}_{a1}) - d(\hat{V}_1 \cdot \vec{V}_{a1})}{1 - d^2}$$

In your solution derivation process, make sure to pay special attention to normalized and un-normalized vectors.

Handling Parallel Lines

Recall that the solutions for d_1 and d_a are derived based on the observation and simplification of the simultaneous equations,

$$\hat{V}_1 \cdot \vec{V}_p = 0$$
$$\hat{V}_a \cdot \vec{V}_p = 0$$

Now, if the two line segments are parallel, then, $\hat{V}_1 = \hat{V}_a$ and thus there is only one equation with two unknowns. For this reason, the derived solution is valid only when $\hat{V}_1 \neq \hat{V}_a$, or when the two lines are not parallel.

In general, the shortest distance between two parallel lines can be determined by computing the shortest distance between one of the lines to the end point on the other line. Now, modify MyScript to support distance computation between parallel lines.

Notice your solution assumes both line segments are infinitely long where the closest positions on each line can be outside of their respective line segments. Once again, this is not a complete solution to closest distance between the two finite length line segments. Imagine the explorer and the monster pathways, when the closest positions are outside of the line segments, the distance computed would be based on positions that the explorer or the monsters will not move to. The general solution is similar to that of the line to point distance, when the closest position is outside, it should be clamp to the corresponding end point.

Summary

This chapter continues with the exploration of vectors by introducing the vector dot product, a tool for analyzing relationships between two vectors. Since a vector is defined by a size and a direction, the tool for analyzing the relationships between two vectors reports on the relative directions and sizes of these vectors.

The definition of the vector dot product is straightforward, the sum of the products of the corresponding components of the two vectors, and the result is a simple signed floating-point number. There are four ways to compute the dot product between two vectors and each offers a unique geometric insight into the resulting floating-point number.

The first way of computing a dot product is by operating on two non-normalized vectors. The resulting floating-point number is the product of the sizes of the two vectors and the cosine of their subtended angle. While the least useful, this floating-point number does provide slight insight into the subtended angle between the two vectors. If the number is positive, then the subtended angle is less than $90°$, otherwise, the angle is between $90°$ and $180°$.

The second way of computing a dot product is by operating on two normalized vectors. In this case, the resulting floating-point number is simply the cosine of the subtended angle. This result is invaluable when you need to determine how much two directions differ. In fact, checking the dot product results of two normalized vectors against approximately 0 or 1 for when the two vectors are almost perpendicular or parallel, are two of the most frequently encounter test cases in video game development.

The third and fourth way of computing a dot product is to ensure only one of the operands is normalized. In this scenario, you are computing the projected length of the non-normalized vector along the direction of the normalized vector. These forms of computing the dot product have the broadest application. This is because projected sizes, as you have experience with line to point and line to line distance computation, are the basis for computing distances, and, as you will learn in the next chapter, for computing intersections.

You have learned about vectors, gained knowledge on how to analyze the relationships between vectors, and applied these concepts in solving some interesting and non-trivial geometric problems. In the next chapter, you will learn about the vector cross product, a tool to relate two vectors to the space that contains those vectors. But before you continue, here are the summaries of the vector dot product definition, rules, and straightforward applications.

Vector dot product definition and implications

Dot Product Definition	Remark
$\vec{V}_1 \cdot \vec{V}_2 = x_1x_2 + y_1y_2 + z_1z_2$	Definition of the dot product, also referred to as the inner product
$\vec{V}_1 \cdot \vec{V}_2 = \|\vec{V}_1\|\|\vec{V}_2\| \cos\theta$	Geometric interpretation of the dot product definition, θ is the angle subtended by the two vectors
$\vec{V}_1 \cdot \vec{V}_1 = \|\vec{V}_1\|^2$	Dot product of a vector with itself is the squared of its magnitude
$\vec{V}_1 \cdot ZeroVector = ZeroVector$	Dot product with the zero vector is the zero vector

Interpreting the dot product results

Dot Product	Geometric Interpretations
Direction: $\hat{V}_1 \cdot \hat{V}_2 = \cos\theta$	When both operands are normalized, the result of dot product is the cosine of the subtended angle

Projected size: $\hat{V}_1 \cdot \vec{V}_2 = \|\vec{V}_2\| \cos\theta$	Projected size of $\vec{V}_2$ (the un-normalized vector) along the $\hat{V}_1$ (the normalized vector) direction
Projected size: $\vec{V}_1 \cdot \hat{V}_2 = \|\vec{V}_1\| \cos\theta$	Projected size of $\vec{V}_1$ along the $\hat{V}_2$ direction

Insights into the subtended angle

Dot Product Results	The Angle θ	Conclusions
$\hat{V}_1 \cdot \hat{V}_2 = \cos\theta = 1$	$\theta = 0°$	The vectors are in the exact same direction, $\hat{V}_1 == \hat{V}_2$
$\hat{V}_1 \cdot \hat{V}_2 = \cos\theta = 0$	$\theta = 90°$	The vector directions are perpendicular to each other
$\hat{V}_1 \cdot \hat{V}_2 = \cos\theta > 0$	$\theta < 90°$	The vectors are pointing along similar directions
$\hat{V}_1 \cdot \hat{V}_2 = \cos\theta < 0$	$\theta > 90°$	The vectors are pointing along similar, but opposite directions
$\hat{V}_1 \cdot \hat{V}_2 = \cos\theta = -1$	$\theta = 180°$	The vectors are pointing in the exact opposite direction $\hat{V}_1 == -\hat{V}_2$

The Line Equations

The line segment bounded by the given two positions, P_0 and P_1, can be expressed as either of the following,

$$lv(s) = P_0 + s\vec{V}_1$$

$$l(t) = P_0 + t\hat{V}_1$$

where,

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

and the values of the parameters s, and t, provide the following insights into a position on the line segment.

Values of s	Values of t	Position identified
$s < 0$	$t < 0$	Measured along the $\vec{V}_1$ direction, a position before the beginning position, P_0
$0 \le s \le 1$	$0 \le t \le \|\vec{V}_1\|$	A position within the line segment
$s > 1$	$t > \|\vec{V}_1\|$	Measured along the $\vec{V}_1$ direction, a position after the end position, P_1