CHAPTER 6

■ ■ ■

Chapter 6: Vector Cross Products and 2D Planes

After completing this chapter, you will be able to:

- Differentiate between the Left-Handed and Right-Handed 3D Coordinate System,
- Discuss the vector cross product definition and the resulting vector direction and magnitude,
- Describe the geometric interpretation of the vector cross product,
- Relate the 2D plane equation to the vector plane equation and its parameters,
- Interpret the geometric implications of the vector plane equation,
- Relate the cross product result to 2D plane equations,
- Derive an axis frame when given two non-parallel vectors,
- Apply the vector concepts learned to solve point to plane distance, point to plane projection, line to plane intersection, and reflecting a vector across a plane.

Introduction

In Chapter 4, you learned about vectors, that the relationship between two positions can be defined by a direction and a distance. Vectors and their rules of operation enabled you to precisely describe and analyze object motions. In Chapter 5, you learned about vector dot products, that the relationship between two vectors can be characterized by their subtended angle and projected sizes. The vector dot product and its rules of operation allowed you to accurately represent and analyze arbitrary line segments, including distances between these line segments and other objects. In this chapter, you will learn about how the vector cross product can be used to relate two vectors to the space that defines these vectors and some applications of these concepts.

The result of the vector cross product is a new direction. Interestingly, and as you will learn, this new direction characterizes the space that defines the two vectors as a 2D plane, i.e., this new direction defines a plane that both vectors exist on. This new knowledge enables two general capabilities. First, a convenient representation of and the ability to analyze arbitrary 2D planes, including computing distances to, projections onto, and line intersections with any 2D plane. Although these are not direct applications, they are topics that become more comprehensible because of insights gained from the understanding of the vector cross product. The second general capability the vector cross product provides is a precise description of vector direction manipulations. This chapter will examine and pursue the first capability of working with arbitrary 2D planes. The study of vector direction manipulation is a subject within the more advanced topic of transformations and will not be covered in this book.

In video games, it is often necessary to process and analyze the relationships between planes and objects or the motion of objects. For example, in a city building game with a top-down view perspective, when a meteoroid is fast approaching the player's city, you may want to project the shadow of the meteoroid as it travels across the city as well as highlight its impact zone to warn players of the impending destruction. Additionally, immediately after the impact, you may want the meteoroid to bounce or slide across the ground. The shadow indicator can be accomplished by projecting the meteoroid onto the city plane, the reflection direction for the bounce is the velocity line reflecting off the ground plane, and the sliding direction would be the reflection direction projected onto the ground plane. As you can see from this brief example, the ability to represent and work with 2D planes are indeed fundamental to video game development.

The chapter begins by introducing conventions for representing a 3D coordinate system so that you can analyze three perpendicular vectors with consistency. The details of the cross products are then described. The application of the cross product results is then showcased in the solution to the inside-outside test of a general 2D region. At this point the chapter takes a slight change in perspective; instead of analyzing problems and solutions based on the results of the cross product, the chapter focuses on applying the insights gained from the vector cross product in the interpretation of the vector plane equation. The remaining of this chapter examines some of the important problems in video game development when working with 2D planes.

3D Coordinate System Convention

Since the analysis of the vector cross product involves understanding the direction of vectors in 3D space, you need to understand the conventions of representing a 3D coordinate system. In 2D space, when referencing the Cartesian Coordinate System, it is a generally agreed upon convention that the origin is on the lower left, the X-axis points towards the right, and the Y-axis points upwards. Note that this is a convention and not a mathematical rule or any kind of property. People simply agree to follow these sets of rules.

Unfortunately, there are two sets of generally accepted conventions for 3D space. Although you have been working with 3D vectors, until now, there has not been the need to focus on the specific directions of the major axes. As you will see, unlike the dot product, the vector cross product result is not a simple floating-point number, but a vector that is perpendicular to both of the operand vectors. In this case, it is critical and essential to understand, differentiate, and follow one of the 3D coordinate system conventions. Figure 6-1 illustrates the two different conventions in describing a 3D coordinate system, either according to the left or the right hand. These are referred to as the **Left-** or **Right-Handed Coordinate System**.

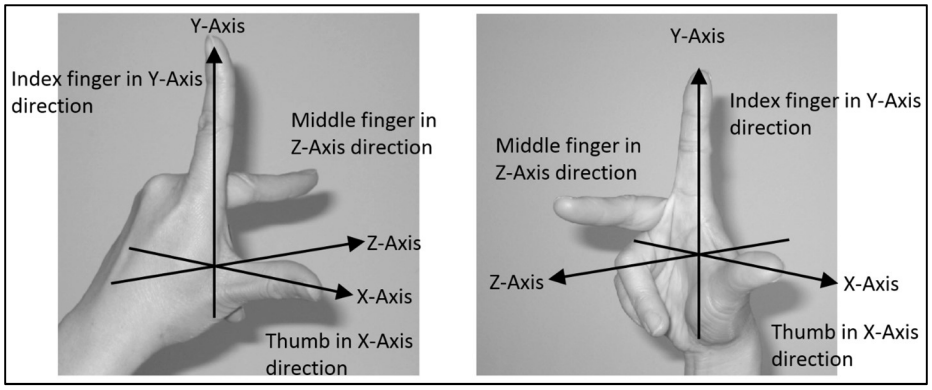

Figure 6-1. *The Left- and Right-Handed Coordinate System*

In both the Left- and Right-Handed Coordinate Systems, the first three fingers are used to represent and point in the directions of the X-, Y-, and Z-axis. The thumb represents and points in the direction of the X-axis, the index finger the Y-axis, and the middle finger the Z-axis. The left and right images of Figure 6-1 show that under this convention, while the X- and Y-axes still follow the right- and up-wards directions, the Z-axis directions are opposite.

Both the Left- and Right-Handed conventions are accepted in general by the video game and interactive graphics community. These are conventions for analyzing and discussing directions. It is critical to know the reference, the Left- or Right-Handed system, being used, and essential to be consistent in following the selected convention. Fortunately, once selected and followed consistently, there are no other consequences or special cases in any of the discussions concerning the fundamentals of vector math. It is simply important to know which convention is used and to be sure to follow that convention consistently throughout.

Unity Follows the Left-Handed Coordinate System

Figure 6-2 shows a screenshot of the Unity Editor Scene View where the top-right coordinate icon is zoomed in upon and shown on the right of the figure. You can verify with your left hand, that, with your thumb stretching out along the red X-axis, your index finger following the green Y-axis, and your middle finger in the direction of the blue Z-axis, that Unity follows the Left-Handed Coordinate System convention. Therefore, this is the convention that will be followed in this book. Once again, all the concepts being discussed are applicable to either 3D coordinate system convention as long as you follow the selected convention and maintained consistency.

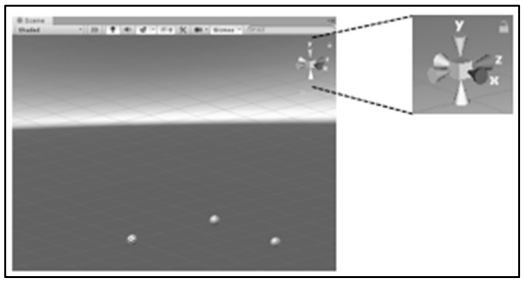

Figure 6-2. *The Unity Editor Scene View window coordinate icon*

Vector Cross Product: The Perpendicular Direction

Recall in the previous chapter where you verified that a 2D plane can always be derived to draw two non-parallel vectors. This 2D plane is the plane that represents the space, or area, that defines, or contains, these two vectors. Through this chapter, you will learn that 2D planes are characterized by a vector that is perpendicular to it and that this perpendicular vector is the result of the cross product between two non-parallel vectors.

Figure 6-3 shows that in general, there are two directions that are perpendicular to any two non-parallel vectors $\vec{V}_1$ and $\vec{V}_2$. Once again, as discussed previously, these two vectors are depicted at the same tail location for convenient visual analysis. It is important to reiterate that the vector definition is independent of positions and the following discussions are valid even when the two vectors do not share the same tail position.

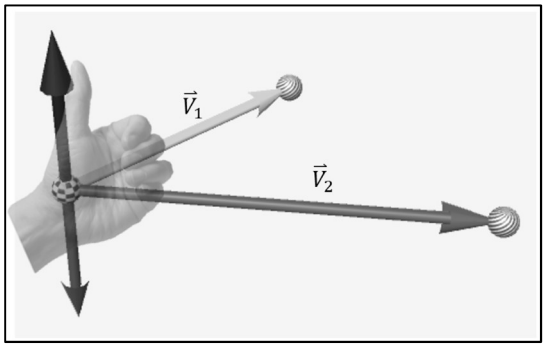

Figure 6-3. *Vectors that are perpendicular to the two non-parallel vectors, $\vec{V}_1$ and $\vec{V}_2$*

Figure 6-3 shows a left-hand thumb pointing in a direction where the index to little fingers are aligned with the direction of the first vector, $\vec{V}_1$, and then curl towards the second vector, $\vec{V}_2$. The left thumb direction is the one that is perpendicular to the plane that defines $\vec{V}_1$ and $\vec{V}_2$. Of course, the direction opposite to the left thumb is the second direction that is perpendicular to the plane that defines these two vectors.

■ ***Note*** *The left hand is used for direction resolution because this book follows Unity's choice of Left-Handed Coordinate System. A Right-Handed Coordinate System would follow the same finger curling process as Figure 6-3 with the right hand and identify a set of directions that seem opposite to that of Figure 6-3. Please do not be concerned. Remember that the left- and right-handed conventions also affect the directions of the major axes. Once again, in the end, both conventions, as long as followed consistently throughout, will produce identical results.*

The vector cross product computes the two new directions, along or opposite to the thumb-direction in Figure 6-3. These are the two directions that are perpendicular to both of the vectors, $\vec{V}_1$ and $\vec{V}_2$. This chapter will lead you on a journey to examine, understand, and relate these results to 2D planes in 3D space. After which, the problems and solutions associated with 2D planes that are relevant to video game development will be analyzed.

Definition of Vector Cross 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 **cross product**, or vector cross product, between the two vectors is defined as,

$$\vec{V}_1 \times \vec{V}_2 = (y_1z_2 - z_1y_2,\ z_1x_2 - x_1z_2,\ x_1y_2 - y_1x_2)$$

Notice that,

- **Symbol**: the symbol for the cross product operation, "×", is literally a "cross"
- **Operands**: the operation expects two vector operands
- **Result**: the result of the operation is a vector with x-, y-, and z-component values.

When compared to the other vector operations you have learned, the cross product also expects two vector operands. Additionally, similar to vector addition and subtraction, and in contrast to the vector dot product, the result of the vector cross product is a vector.

Unlike vector addition and subtraction, the vector cross product result, the x-, y-, and z-component values are not straightforward functions of its operands' corresponding components. Examine these values carefully and you will notice a pattern. For example, the x-component result, $y_1z_2 - z_1y_2$, is the subtraction of the multiplication of operand component values other than their x-components. This pattern is consistent for each of the y- and z-components. Though interesting and important in general, in the context of video game development, these observations do not lead to direct applications.

The left, center, and right tables in Figure 6-4 illustrate an approach that may help you remember the cross product formula. Each of the tables has an x-, y-, and z- heading with two rows consisting of the corresponding component values for the two operand vectors. The left table shows that the x-component cross product result is computed by ignoring the greyed-out x-component values, following the two arrows, and calculating and subtracting the products of the y- and z-components y_1z_2 and z_1y_2. The center table shows a similar computation for the y-component cross product results and the right table for the z-component cross product results. Note that the subtraction order for the y-component is reversed that of the x- and z-components.

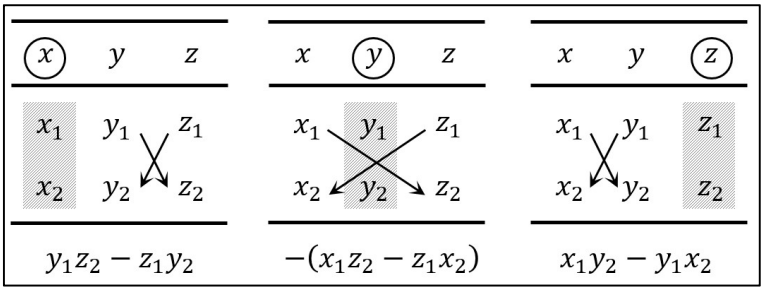

Figure 6-4. *Components of the cross product*

Geometric Interpretation of Vector Cross Products

Figure 6-5 shows the geometric interpretation of the vector cross product. Since Unity follows the Left-Handed Coordinate System, the result of $\vec{V}_1 \times \vec{V}_2$ is a vector in the direction of the thumb on your left hand when following the finger curling process described previously. It follows that for $\vec{V}_2 \times \vec{V}_1$, with the index to little fingers aligned with the first operand, in this case the $\vec{V}_2$ vector, and then curl towards the second operand, or the $\vec{V}_1$ vector, the resulting vector is in the opposite direction (turn your hand so you're giving a thumbs down instead of a thumbs up). The cross product results, $\vec{V}_1 \times \vec{V}_2$ and $\vec{V}_2 \times \vec{V}_1$, are perpendicular to their operand vectors, $\vec{V}_1$ and $\vec{V}_2$, and as a result, are perpendicular to the plane that defines $\vec{V}_1$ and $\vec{V}_2$.

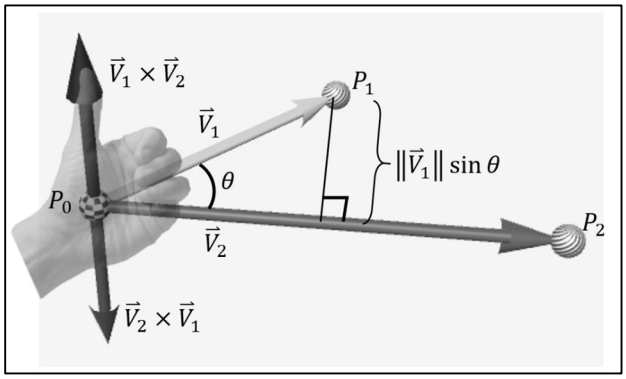

Figure 6-5. *The directions of vector cross product results*

The magnitude of the vector resulting from the cross product, or the magnitude of the perpendicular vector, with details left as an exercise, can be shown to be,

$$\|\vec{V}_1 \times \vec{V}_2\| = \sqrt{(y_1z_2 - z_1y_2)^2 + (z_1x_2 - x_1z_2)^2 + (x_1y_2 - y_1x_2)^2} = \|\vec{V}_1\|\|\vec{V}_2\| \sin\theta$$

Where θ is the subtended angle between $\vec{V}_1$ and $\vec{V}_2$. Notice that when both $\vec{V}_1$ and $\vec{V}_2$ are normalized, thus both with magnitude of 1.0, then,

$$\|\hat{V}_1 \times \hat{V}_2\| = \sin\theta$$

■ ***Note*** *Although the cross product result encodes the sine of the subtended angle, it is seldom, if ever, used specifically for analyzing subtended angles between vectors. Instead, the dot product is always used. This is because when comparing the two, the cross product operation involves more floating-point operations, and more importantly, the cross product result is a vector and thus a magnitude operation must be performed to convert the vector into a floating-point number for deriving the angle information. In contrast, the dot product is more efficient to compute and the result itself encodes the angle information and thus does not need further processing. For these reasons, the dot product is always used for analyzing angles subtended by vectors, e.g., testing for parallel or perpendicular.*

In Figure 6-5, notice that $P_0P_1P_2$ is a triangle. Assuming the edge, P_0P_2, is the base, then you know the area of the triangle is the half the length of the base, or $\|\vec{V}_2\|$, multiplied by the height. In this case, the height is the perpendicular distance between P_1 and the edge, P_0P_2, or, $\|\vec{V}_1\| \sin\theta$. In this way, the area of the triangle $P_0P_1P_2$ is,

$$Area\ of\ Triangle\ P_0P_1P_2 = \frac{1}{2}\|\vec{V}_1\|\|\vec{V}_2\|\sin\theta$$

And, the magnitude of the cross product result is twice the area of the triangle,

$$\|\vec{V}_1 \times \vec{V}_2\| = 2 \times Area\ of\ Triangle\ P_0P_1P_2 = \|\vec{V}_1\|\|\vec{V}_2\|\sin\theta$$

Though the magnitude of the resulting vector and the sine relationship of the subtended angle are important information to take note of when learning the vector cross product, the analysis presented in the rest of this book only take advantage of the fact that the resulting vector is perpendicular to the operands and the 2D plane that defines the operand vectors.

Properties of Vector Cross Product

The vector cross product properties of commutative, associative, and distributive over a floating-point scaling factor s are summarized in Table 6-1.

Table 6-1. *Properties of vector cross product*

Properties	Vector Dot Product
Anti-Commutative	$\vec{V}_1 \times \vec{V}_2 = -\vec{V}_2 \times \vec{V}_1$
Not Associative	$(\vec{V}_1 \times \vec{V}_2) \times \vec{V}_3 \neq \vec{V}_1 \times (\vec{V}_2 \times \vec{V}_3)$
Distributive over scale factor, s	$s(\vec{V}_1 \times \vec{V}_2) = (s\vec{V}_1) \times \vec{V}_2 = \vec{V}_1 \times (s\vec{V}_2)$

Table 6-1 shows a set of rather unfamiliar properties. Fortunately, the applications of vector cross products in video game development are often limited to simple operations in the determination of directions. It is seldom for cross product operations to be embedded in complex vector equations. Finally, the definition of the vector cross product states that,

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

And that, any vector crossed with the zero vector will results in a zero vector,

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

The Vector Cross Products Example

This example demonstrates the results of performing the vector cross product between two given vectors. This example allows you to interactively manipulate and define two vectors and then examine the results of performing the cross product between these vectors. Figure 6-6 shows a screenshot of running the EX_6_1_VectorCrossProducts scene from the Chapter-6-Examples project.

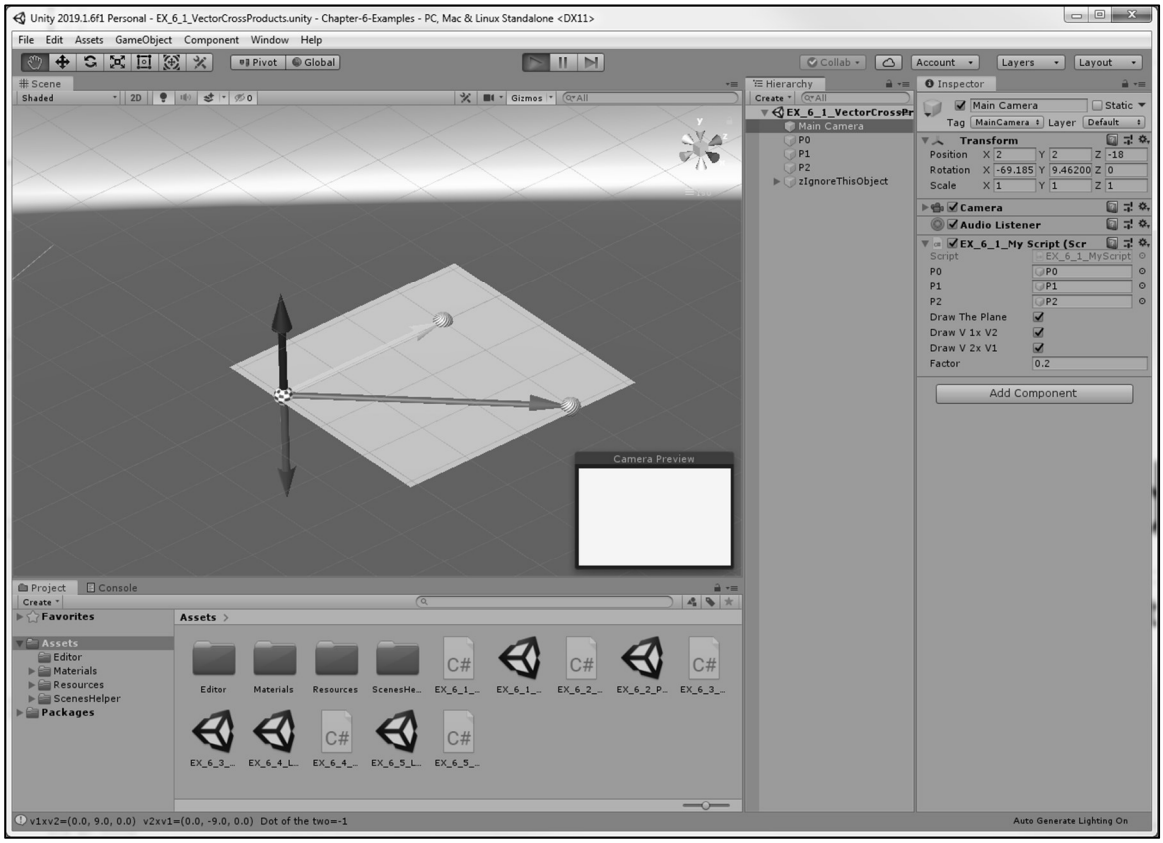

Figure 6-6. *Running the Vector Cross Products example*

The goals of this example are for you to:

- Examine the results of the cross product between two arbitrarily defined vectors
- Verify that the vector resulting from a cross product is perpendicular to both of the operands with a magnitude that is directly proportional to the sine of their subtended angle
- Examine the source code that computes and uses the results of the vector cross product

Examine the Scene

Take a look at the `Example_6_1_VectorCrossProducts` scene and observe the predefined game objects in the Hierarchy Window. In addition to `MainCamera`, there are three objects in this scene: a checkered sphere (`P0`) and two striped spheres (`P1` and `P2`). These three game objects will have their corresponding `transform.localPosition` properties referenced to define the two vectors for performing the cross product operations.

Analyze MainCamera MyScript Component

The MyScript component on MainCamera shows two sets of variables. One set is for defining the two vectors and the other set is for examining the visualization of the cross product between these two vectors and the plane that they define. The first set of variables are P0, P1, and P2, and are defined for accessing the game objects with their corresponding names. In this example, you will manipulate the positions of these 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 then examine the result of the cross product between these vectors.

The variables in the second set, DrawThePlane, DrawV1xV2, and DrawV2xV1 are toggles for hiding and showing the plane that defines $\vec{V}_1$ and $\vec{V}_2$, and the corresponding results of the cross products. While the last variable, Factor, is the scaling factor applied to the length of the vector from the cross product result, allowing for easier visualization.

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 $\vec{V}_1$ and $\vec{V}_2$, and are defined by the positions of P0, P1, and P2 game objects as previously explained. You will also observe two other vectors in this scene. Both of these vectors are located at the checkered sphere location (P0), a black vector that is the result of $\vec{V}_1 \times \vec{V}_2$, and a red vector, the result of $\vec{V}_2 \times \vec{V}_1$. You can confirm that both of these results follow the Left-Handed Coordinate System by extending the index to little fingers of your left hand along the $\vec{V}_1$ direction (the cyan vector) and then curling these fingers towards the $\vec{V}_2$ direction (the magenta vector). In a similar fashion to that of Figure 6-5, your thumb should be pointing along the direction of the black vector which is the result of $\vec{V}_1 \times \vec{V}_2$. You can repeat the left-hand finger curling process to verify that the red vector is indeed pointing in the direction of $\vec{V}_2 \times \vec{V}_1$.

In the Console Window, you can examine the text output where the subtended angle between $\vec{V}_1$ and $\vec{V}_2$ as well as various dot product results are printed for verification purposes. First, you can verify that the printed subtended angles between $\vec{V}_1$ and $\vec{V}_2$ reflect your observations in the Scene View. Next, examine the results of the dot product between the normalized black and red vectors. Since these two vectors are always parallel and pointing in the opposite directions, the angle between them is always $180°$ and thus the result of the dot product, or the cosine of this angle, is always -1,

$$(\hat{V}_1 \times \hat{V}_2) \cdot (\hat{V}_2 \times \hat{V}_1) = -(\hat{V}_1 \times \hat{V}_2) \cdot (\hat{V}_1 \times \hat{V}_2) = -1$$

Additionally, the results of the dot product between the cross product result, $(\hat{V}_1 \times \hat{V}_2)$, and the operands, $\hat{V}_1$and $\hat{V}_2$ are also printed out. You can verify that the cross product result is always perpendicular with its operands by observing that the dot product results between these vectors are always zero, or very close to being zero,

$$(\hat{V}_1 \times \hat{V}_2) \cdot \hat{V}_1 = (\hat{V}_1 \times \hat{V}_2) \cdot \hat{V}_2 = 0$$

Note that since the initial values of P0, P1, and P2 define the three positions to be on the X-Z plane, the initial $\vec{V}_1$ and $\vec{V}_2$ vectors are also in the X-Z plane. Therefore, the cross product results are vectors pointing in the positive and negative y-directions, perpendicular to both $\vec{V}_1$ and $\vec{V}_2$, and the plane that defines these two vectors, the X-Z plane.

In the following interactions, feel free to toggle and hide any of the components if you find them distracting. You can also adjust the Factor value to scale the lengths of the black and red vectors for easier visual examination.

Select P1 and adjust its z-component value to change the size of $\vec{V}_1$ without changing the subtended angle. Notice that although both are changing, the lengths of the black and red vectors are always the same. This is because both of the vectors vary in direct proportion to the length of $\vec{V}_1$. Now try moving P1 towards P2 such that the $\vec{V}_1$ vector approaches $\vec{V}_2$, or move P1 towards

P0 such that the $\vec{V}_1$ vector approaches the zero vector. Notice that in both cases, the cross product result, the black and the red vectors, both approach a length of zero. You can repeat and verify all these observations by adjusting P2, or by changing $\vec{V}_2$, in a similar fashion. These manipulations and observations verify that the magnitude of the cross product result is in direct proportion to the magnitude of the operand vectors,

$$\|\vec{V}_1 \times \vec{V}_2\| = \|\vec{V}_1\|\|\vec{V}_2\| \sin\theta$$

and that all cross products computed with the zero vector will result in the zero vector.

Now restart the game and adjust the x-component of P1 to change the subtended angle. Notice that when this angle is between $0°$ and $90°$, the lengths of the black and red vectors vary in direct proportion and then change to vary in the inverse proportion when the angle is beyond $90°$. Continue to adjust both the x- and z-component values to increase the subtended angle to beyond $180°$ and notice the direction swap between the black and red vectors. Recall that a subtended angle is always between $0°$ and $180°$, you can verify with your left-hand that after the direction swap the black vector is still pointing in the direction of $\hat{V}_1 \times \hat{V}_2$.

Notice that until this point, your manipulation has been restricted to the X-Z plane and that the cross product results, the black and red vectors, are always in the positive and negative y-directions. Now, select any of the positions and change the y-component values. As you have observed when investigating the dot product in the previous chapter, the green plane is updated and continues to cut through both $\vec{V}_1$ and $\vec{V}_2$. The interesting observation is that the cross product results, the black and red vectors, are always perpendicular to the green plane. This observation suggests that the green plane is defined by the cross product result. This concept will be explored in the next subsection.

Details of MyScript

Open MyScript and examine the source code in the IDE. The instance variables and the Start() function 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;
public bool DrawV1xV2 = true;
public bool DrawV2xV1 = true;
public float Factor = 0.4f;

#region For visualizing the vectors
#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(P2 != null);

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

All the public variables for MyScript have been discussed when analyzing the 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;
    Vector3 v2 = P2.transform.localPosition - P0.transform.localPosition;
    Vector3 v1xv2 = Vector3.Cross(v1, v2);
    Vector3 v2xv1 = Vector3.Cross(v2, v1);

    float d = Vector3.Dot(v1.normalized, v2.normalized);
    bool notParallel = (Mathf.Abs(d) < (1.0f - float.Epsilon));

    if (notParallel) {
        float theta = Mathf.Acos(d) * Mathf.Rad2Deg;
        float cd = Vector3.Dot(v1xv2.normalized, v2xv1.normalized);
        float dv1 = Vector3.Dot(v1xv2, v1);
        float dv2 = Vector3.Dot(v1xv2, v2);
        Debug.Log(" theta=" + theta + "  v1xv2=" + v1xv2 + "  v2xv1=" + v2xv1 +
                  "  v1xv2-dot-v2xv1=" + cd + " Dot with v1/v2=" + dv1 + " " + dv2);
    } else {
        Debug.Log("Two vectors are parallel, cross product is a zero vector");
    }

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

The first four lines of the `Update()` function compute,

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

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

$$v1\text{x}v2 = \vec{V}_1 \times \vec{V}_2$$

$$v2\text{x}v1 = \vec{V}_2 \times \vec{V}_1$$

Next, the cosine of the angle between $\vec{V}_1$ and $\vec{V}_2$ is computed as the dot product of the normalized vectors. This value is examined to ensure that the cross product results will not be zero vectors. The various dot product results are then computed and printed to the Console Window.

■ ***Note*** ***Collinear*** *and* ***collinear test.*** *In general, given three positions,* P_0*,* P_1*, and* P_2*, that define two vectors,* $\vec{V}_1 = P_1 - P_0$*, and* $\vec{V}_2 = P_2 - P_0$*. If* $\hat{V}_1 \cdot \hat{V}_2$ *is approximately* 1 *or* -1*, then, you can conclude that the three points are approximately along the same line. In this case,* P_0*,* P_1*, and* P_2 *are referred to as being collinear. The dot product check against approximately* 1 *or* -1 *is a convenient collinear test.*

Take Away from This Example

This example demonstrates that the result of the cross product is indeed a vector with a direction that can be derived by curling your left-hand fingers, and that the magnitude of the resulting vector is indeed directly proportional to the sizes of the operands and the sine of the subtended angle. You have also confirmed that the cross product of any vector with itself or with the zero vector results in

the zero vector. Additionally, you have verified that the cross product is anti-commutative as reversing the operand order results in a vector pointing in the perfectly opposite direction. However, the most interesting observation is that the cross product result is always perpendicular to the operand vectors and thus the 2D plane that defines the two operand vectors.

Relevant mathematical concepts covered include:

- The cross product result is a vector that is perpendicular to both of its operands and the 2D plane that defines the operands
- The magnitude of the vector resulting from a cross product is directly proportional to the magnitude of the operands and the sine of the subtended angle
- The cross product is not defined for vectors derived from three positions that are collinear. This is because three collinear positions can only define one direction and thus one vector, and the cross product of a vector with itself is the zero vector

EXERCISES

Derive the Magnitude of the Vector Resulting from a Cross Product

Given,

$$\vec{V}_1 = (x_1, y_1, z_1)$$
$$\vec{V}_2 = (x_2, y_2, z_2)$$

You know that the cross product is defined as,

$$\vec{V}_1 \times \vec{V}_2 = (y_1 z_2 - z_1 y_2,\ z_1 x_2 - x_1 z_2,\ x_1 y_2 - y_1 x_2)$$

Where the magnitude of the resulting vector is,

$$\|\vec{V}_1 \times \vec{V}_2\| = \|\vec{V}_1\| \|\vec{V}_2\| \sin\theta$$

Recall the Trigonometry identity and the dot product definition that,

$$\sin^2\theta + \cos^2\theta = 1$$
$$\hat{V}_1 \cdot \hat{V}_2 = \frac{\vec{V}_1 \cdot \vec{V}_2}{\|\vec{V}_1\| \|\vec{V}_2\|} = \cos\theta$$

So,

$$\|\vec{V}_1 \times \vec{V}_2\| = \|\vec{V}_1\| \|\vec{V}_2\| \sin\theta$$
$$= \|\vec{V}_1\| \|\vec{V}_2\| \sqrt{1 - \cos^2\theta}$$

$$= \|\vec{V}_1\|\|\vec{V}_2\| \sqrt{1 - \left(\frac{\vec{V}_1 \cdot \vec{V}_2}{\|\vec{V}_1\|\|\vec{V}_2\|}\right)^2}$$

Now, simplify the algebra expression and show that,

$$\|\vec{V}_1 \times \vec{V}_2\| = \sqrt{(y_1z_2 - z_1y_2)^2 + (z_1x_2 - x_1z_2)^2 + (x_1y_2 - y_1x_2)^2}$$

Verify the Cross Product Formula

When computing the cross products in MyScript,

```
Vector3 v1xv2 = Vector3.Cross(v1, v2);
Vector3 v2xv1 = Vector3.Cross(v2, v1);
```

Replace these two lines of code with the explicit cross product definition by creating `v1xv2` and `v2xv1` as new `Vector3` objects with appropriate component values and verify that the runtime results are identical.

The Vector Plane Equation

Throughout the last couple of chapters, you have been working with two vectors defined by three positions and observed that a 2D plane can always be defined when the two vectors are not parallel. Note that both of these observations are identical, two non-parallel vectors is the same as saying that the three positions that define the two vectors are non-collinear. Intuitively, this should not be surprising because from basic geometry you have learned that three points, as long as they are not all along the same line, define a triangle, and a triangle is the simplest shape in 2D space. For this reason, if a triangle can be formed, as you have observed, then it is always possible to form two non-parallel vectors, and a 2D plane can thus always be defined as well.

Now, you can derive the equation of this 2D plane based on the result of the cross product. Recall from basic geometry that the equation of a 2D plane in 3D space is,

$$Ax + By + Cz = E$$

where A, B, C, and E are floating-point constants, and x, y, and z are unknowns in 3D space. This equation states that if you gather all the positions (x, y, z) that satisfy the condition where the sum of multiplying x by A, y by B, and z by C, is equal to E, then, you will find that all these positions are points on the given 2D plane. Interestingly this equation can also be written in vector dot product form, where you can define the vector $\vec{V}$, and a position vector, p, where

$$\vec{V} = (A,\ B,\ C)$$

$$p = (x,\ y,\ z)$$

then, the 2D plane equation can be written as,

$$\vec{V} \cdot p = E$$

■ ***Note*** *Recall that a position, p, can be interpreted as a position vector, $\vec{V}$, from the origin position, P_0, where,*

$$\vec{V} = p - P_0 = p$$

Since in this case, P_0 is the origin, $(0,0,0)$. To avoid the confusion and nuance of introducing additional symbols, it is a common practice to reuse the symbol of the position (p) to represent the corresponding position vector. In the rest of this book, please do not be confused when you encounter language and a symbol such as, "following along the position vector p." Such statements are always referring to the vector from the origin towards the position, p.

If you divide both side of the equation by a non-zero floating-point number, in this case, $\|\vec{V}\|$, the equation becomes,

$$\hat{V} \cdot p = \frac{E}{\|\vec{V}\|}$$

now, let $D = \frac{E}{\|\vec{V}\|}$, then a 2D plane equation can be written as the **vector plane equation**, or,

$$\hat{V} \cdot p = D$$

This equation may look familiar because it is basically the vector projection equation as illustrated in Figure 5-7. Figure 6-7 shows the geometric interpretation of the vector plane equation.

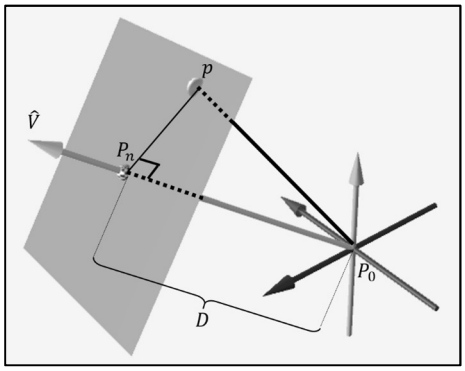

Figure 6-7. *Geometric interpretation of the vector plane equation*

In Figure 6-7, P_0 is the origin and the vector $\hat{V}$ is the direction from the origin that is perpendicular and passes through a plane at position P_n. The plane is at a distance D from the origin when measured along the direction $\hat{V}$. The vector plane equation states that for any position p on this plane, it is true that the projection of this position vector onto the direction $\hat{V}$ will be of length D. In this

way, the vector plane equation identifies all positions that satisfy the projected distance relationship with the $\hat{V}$ vector. As it turns out, these positions define the 2D plane. Notice that you must compute the $\hat{V}$ and D to derive the vector plane equation, $\hat{V} \cdot p = D$,

- **Normal vector**: $\hat{V}$ is the vector that is perpendicular to the plane, this vector is generally normalized such that the constant D in the equation indicates distance from the origin. As demonstrated in the derivation process, when this vector is not normalized, the magnitude of the vector can be divided through on both sides of the equation to compute the proper value for D.
- **Distance to the plane**: D, when the normal vector is normalized, this is the plane distance from the origin when measured along the $\hat{V}$ direction

It is important to recognize that the vector plane equation identifies a 2D plane that is of infinite size. Any position in the Cartesian Coordinate System that satisfies the projected distance relationship is part of the solution set of the 2D plane and there are infinitely many positions in the solution set. As will be explored later, a 2D region is a bounded area on a 2D plane. This is analogous to 1D region, or a 1D interval, being a bounded line segment within an infinitely long line that is identified by a line equation.

 Note *A normal vector is a vector that is perpendicular to a plane. This should not be confused with a normalized vector, which is any vector of size* 1*. You can compute a normal vector which may not be normalized. You can then decide to normalize the normal vector such that you can work with a normalized normal vector. In the rest of this book, the vector symbol,* $\vec{V}_n$*, will be used to represent the normal vector of a 2D plane. Once again, a normal vector may or may not be normalized. In this case,* $\vec{V}_n$*, is a normal vector that is not normalized, and, the vector,* $\hat{V}_n$*, is the normalized plane normal vector.*

The Position P_n on a Plane

Notice the position P_n in Figure 6-7, this is the point on the plane that is D distance away from the origin when measured along the $\hat{V}_n$ direction. For this reason,

$$P_n = P_0 + D\hat{V}_n = D\hat{V}_n$$

Since, in this case, P_0 is the origin, $(0, 0, 0)$. In the rest of this chapter, the P_n position is computed and displayed on the 2D planes in all examples to provide orientation for and facilitate visualization.

Given a Position on a Plane

If you are given a plane normal vector, $\hat{V}_n$, and a position, P_{on}, that is on the plane, then, you know that for any position, p, on the plane, $\overrightarrow{p - P_{on}}$ is a vector on the plane and that this vector must be perpendicular to $\hat{V}_n$. This means,

$$\hat{V}_n \cdot (p - P_{on}) = 0 \qquad \text{since } \hat{V}_n \text{ is perpendicular to } (p - P_{on})$$

This equation can be simplified as follows,

$$\hat{V}_n \cdot p - \hat{V}_n \cdot P_{on} = 0 \qquad \text{follow the distributive property over vector subtraction}$$

$\hat{V}_n \cdot p = \hat{V}_n \cdot P_{on}$	move the term across the equality
$\hat{V}_n \cdot p = D$	$\hat{V}_n \cdot P_{on} = D$ because P_{on} is on the plane

which is simply the vector plane equation.

Positions on 2D Planes

As a way of verifying the vector plane equation and to provide additional insights, Figure 6-8 shows that it is always possible to compute the point where a position vector intersects a plane.

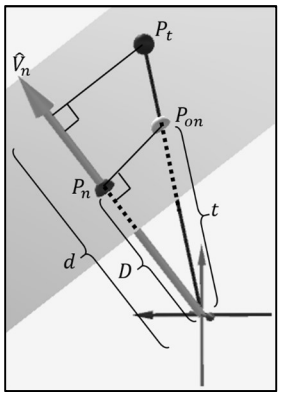

Figure 6-8. *Positions on a given plane.*

In Figure 6-8, the given plane is defined by the normalized normal vector, $\hat{V}_n$, and the distance, D, measured along the $\hat{V}_n$ direction from the origin, or,

$$\hat{V}_n \cdot p = D$$

For any arbitrary position, P_t, it is always possible to compute P_{on}, the point where the position vector P_t intersects the given plane. As illustrated in Figure 6-8, P_{on} is along the position vector P_t and is t distance away from the origin,

$$P_{on} = origin + t_{\ t} = tP_t$$

Since P_{on} is on the plane, then it must be true that,

$$\hat{V}_n \cdot P_{on} = D$$

or,

$\hat{V}_n \cdot tP_t = D$	Since you know $P_{on} = tP_t$
$t(\hat{V}_n \cdot P_t) = D$	Distributive property of dot product over scaling factor t
$t = \frac{D}{\hat{V}_n \cdot P_t}$	Divide both sides by the floating-point number, $\hat{V}_n \cdot P_t$

With the distance, t, defined, it is now possible to compute the value of P_{on}! In the next example, the plane equation will be examined, especially in relation to the cross product result.

The Vector Plane Equations Example

This example demonstrates the vector plane equation. The example allows you to interactively define a 2D plane, manipulate an arbitrary point and examine the intersection of this position vector with the 2D plane. Figure 6-9 shows a screenshot of running the EX_6_2_VectorPlaneEquations scene from the Chapter-6-Examples project.

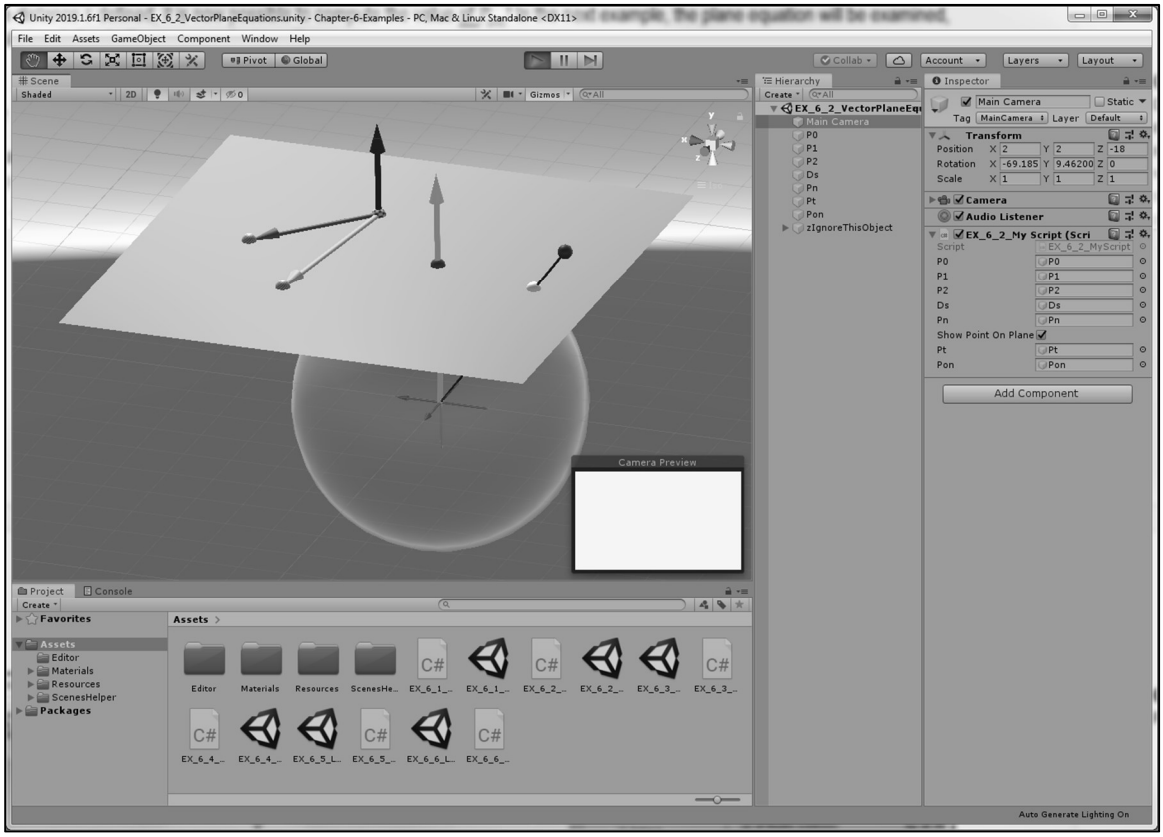

Figure 6-9. *Running the Vector Plane Equations example*

The goals of this example are for you to:

- Understand that the result of the cross product defines a plane normal vector
- Experience working with and gain an understanding of the parameters of the vector plane equation
- Verify the solution to the intersection between a position vector and a 2D plane
- Examine the implementation of working with the vector plane equation

Examine the Scene

Take a look at the Example_6_2_VectorPlaneEquations scene and observe the predefined game objects in the Hierarchy Window. In addition to MainCamera, there are three sets of variables as follows,

- P0, P1, and P2: game objects for defining two vectors to perform the cross product. The result from the cross product will be used as the plane normal vector
- Ds and Pn: Ds is a transparent sphere located at the origin for showing the plane distance, D, from the origin, and Pn is the position where the plane normal vector with tail at the origin intersects the plane. Note, this is the same as saying, Pn is the point on the plane with position vector in the plane normal direction.
- Pt and Pon: Pt is a position you can manipulate and Pon is the point that the position vector Pt intersects with the plane

Analyze MainCamera MyScript Component

The MyScript component on MainCamera contains variables with the same name as their referenced game objects in the scene, these variables are used for position manipulations. The only exception is Ds, which does not have its position manipulated, instead its radius is set according to the distance, D, in the vector plane equation. The variable that doesn't represent any game object, ShowPointOnPlane, is a toggle used to control the showing or hiding of Pt and Pon computation results.

Interact with the Example

Click on the Play Button to run the example. Notice that initially the ShowPointOnPlane toggle is switched off. You will first focus on examining and understanding the cross product result and its relationship with the plane normal before examining the intersection between a position vector and a plane.

In the initial scene you can observe, similar to the previous example, the familiar P0, P1, and P2 positions defining the $\vec{V}_1$ (in cyan) and $\vec{V}_2$(in magenta) vectors. You can also observe the black vector being computed as the result of $\vec{V}_1 \times \vec{V}_2$. As with the previous example, the $\vec{V}_1$ and $\vec{V}_2$ vectors are defined on a 2D plane. In this scene, the 2D plane tangents, or touches at a single point, a transparent sphere centered at the origin. Here you will also find a white vector with its tail position at the origin, extending and cutting through the 2D plane perpendicularly at the red position, Pn. The white vector is the cross product result, and is thus the plane normal vector, $\hat{V}_n$. The transparent sphere mentioned earlier has a radius, D, which is defined by projecting position P0 in the plane normal direction, or,

$$D = \hat{V}_n \cdot P_0$$

In this way, the 2D plane has a vector plane equation,

$$\hat{V}_n \cdot p = D$$

The red sphere on the plane, P_n, is the position vector that is D distance along the $\hat{V}_n$ direction from the origin, or,

$$P_n = D\hat{V}_n$$

It is worth repeating that this vector plane equation is defined completely by the positions P0, P1, and P2. The plane normal, $\hat{V}_n$, is the cross product of the two vectors defined by those positions, and the plane distance from the origin is the projection of the position vector P_0, in the $\hat{V}_n$ direction. Since the position P0 is referenced in defining both of the parameters of the vector plane equation, adjusting this position causes a profound change in the resulting 2D plane. To verify this, select P0 and adjust its y-component value. Notice the drastic changes to the plane as a result, and, how the transparent sphere size changes accordingly

such that the plane always tangents the sphere. Feel free to adjust any of the P0, P1, and P2 positions to verify that the derived vector plane equation is always correct.

Now that you have verified how the cross product result relates to the plane normal vector and that the plane equation is always correct, you can enable the ShowPointOnPlane toggle. The blue sphere, Pt, is a position that you can manipulate and observe where it would intersect the plane if it followed its direction path to or from the origin, or its position vector. The thin black line, extending from the origin to this blue sphere represents the position vector, P_t. The white sphere, Pon, is the intersection of the position vector P_t with the 2D plane, or where the blue sphere would intersect the plane if it followed the black line back to the origin. Feel free to adjust both the 2D plane and the position vector by manipulating the P0, P1, and P2 positions, and Pt to verify that the intersection result is always correct. Note that when P_t is perpendicular to $\hat{V}_n$, the position vector will be parallel to the plane and there can be no intersection.

Details of MyScript

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

```
// Defines two vectors: V1 = P1 - P0, V2 = P2 - P0
public GameObject P0 = null;    // The three positions
public GameObject P1 = null;    //
public GameObject P2 = null;    //

// Plane equation:    P dot vn = D
public GameObject Ds;           // To show the D-value
public GameObject Pn;           // Where Vn crosses the plane

public bool ShowPointOnPlane = true;  // Show or Hide Pt
public GameObject Pt;           // Point to adjust
public GameObject Pon;          // Point in the Plane, in the Pt direction

#region For visualizing the vectors
#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(P2 != null);
    Debug.Assert(Ds != null);
    Debug.Assert(Pn != 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 the 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() {
    // Computes V1 and V2
    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))
        return;

    // Plane equation parameters
    Vector3 vn = Vector3.Cross(v1, v2);
    vn.Normalize();  // keep this vector normalized
    float D = Vector3.Dot(vn, P0.transform.localPosition);

    // Showing the plane equation is consistent
    Pn.transform.localPosition = D * vn;
    Ds.transform.localScale = new Vector3(D * 2f, D * 2f, D * 2f); // sphere expects diameter

    // Set up for displaying Pt and Pon
    Pt.SetActive(ShowPointOnPlane);
    Pon.SetActive(ShowPointOnPlane);
    float t = 0;
    bool almostParallel = false;
    if (ShowPointOnPlane) {
        float d = Vector3.Dot(vn, Pt.transform.localPosition);  // distance
        almostParallel = (Mathf.Abs(d) < float.Epsilon);
        Pon.SetActive(!almostParallel);
        if (!almostParallel) {
            t = D / d;
            Pon.transform.localPosition = t * Pt.transform.localPosition;
        }
    }

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

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

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

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

and verifies that both are non-zero vectors before continuing. The next three lines compute the vector plane equation parameters,

$$\vec{V}_n = \vec{V}_1 \times \vec{V}_2$$

$$\hat{V}_n = \vec{V}_n.\text{Normalized()}$$

$$D = \hat{V}_n \cdot P_0$$

The two lines that follow, set the P_n position and the diameter of the transparent sphere, `Ds`, such that you can examine these parameters of the vector plane equation,

$$P_n = D\hat{V}_n$$

The `if` condition that follows ensures that `Pt` and `Pon` are computed and displayed only under the command of the user. The two lines in the `if` statement compute,

$$d = \hat{V}_n \cdot P_t$$

and verify that d is not close to zero. This check verifies that the plane normal, $\hat{V}_n$, is not almost perpendicular to the position vector, P_t, or that the position vector is not almost parallel to the plane. Recall that in such a case, there can be no intersection and thus

Pon cannot be computed. When verified that the P_t position vector is not parallel to the plane, the position of Pon is computed within the last if statement,

$$t = \frac{D}{\hat{V}_n \cdot P_t} = \frac{D}{d}$$

$$P_{on} = tP_t$$

Take Away from This Example

This example demonstrates how three non-collinear positions can define two non-parallel vectors which can define a 2D plane. You have examined and analyzed the parameters of the vector plane equation to develop an understanding for their geometric interpretations. The plane equation,

$$\hat{V}_n \cdot p = D$$

describes the plane that is at a fixed distance, D, measured from the origin along the plane normal vector, $\hat{V}_n$. Geometrically, this equation can be interpreted as all positions on this plane have a projected distance, D, when measured from the origin along $\hat{V}_n$. The equation and this interpretation were verified when you manipulated an arbitrary position vector, P_t, and observed the computed intersection position, P_{on}, between the position vector and the plane equation.

By now you have observed quite a few examples of vector value checking, but its importance cannot be overstated. Please do note that the almostParallel condition is effectively ensuring that when computing t,

$$t = \frac{D}{\hat{V}_n \cdot P_t}$$

that the denominator is not a zero value. Once again, it is the responsibility of a video game developer to ensure all mathematic operations performed are well defined and edge cases are checked and handled. Ill-defined conditions for mathematical operations often present themselves as intuitive geometric situations. In this case, when the denominator is close to zero, geometrically, it represents when the position vector, P_t, is almost parallel to the plane and thus an intersection does not exist.

Relevant mathematical concepts covered include:

- Three non-collinear positions define two non-parallel vectors which define a 2D plane
- A 2D plane can be described as being perpendicular to a normal direction and at a fixed distance away from the origin when measured along the normal direction
- An alternative description of a 2D plane is that it is the collection of all positions with position vectors that have the same projected distance along the plane normal

EXERCISES

Verify the Vector Plane Equation

The vector plane equation says that all positions on the plane have the same projected distance. Replace P0 *with* P1 *and then* P2 *in* MyScript *when computing the distance,* D*, and verify that the results are identical.*

The Plane at the Negative Distance

Examine the vector plane equation,

$$\hat{V}_n \cdot p = D$$

and take note that the distance, D, is a projected result and is thus a signed floating-point number. This observation says that there is always a complementary plane that is D away in the negative $\hat{V}_n$ direction. Now, modify MyScript to compute,

$$P_d = -D\hat{V}_n$$

You can visualize this point and begin to imagine the associated plane by defining and using a new sphere game object to represent the position of P_d. This exercise brings home the point that you must be careful with the signs, a simple careless mistake can result in an entirely plausible solution on a completely wrong geometry.

Axis Frames and 2D Regions

Recall that the vector plane equation identifies a 2D plane of infinite size. A 2D region can be defined on this 2D plane for determining if a given position is within the bounds of the region. This functionality is the generalization of the study of interval bounds from Chapter 2. For example, Figure 2-7 illustrated a 2D region on the X-Z plane. Here, the description is a 2D region on any arbitrary plane.

Defining 2D regions on 2D planes is interesting and has some important applications in video game development. However, what is much more important is the implication that given three positions that define two non-parallel vectors, you can actually define a **general axis frame**. Recall that the default axis frame of the Cartesian Coordinate System is the three perpendicular X-, Y-, and Z-axis directions centered at the origin. A general axis frame is three perpendicular directions which need not be aligned with the major axes and can be centered at any position. Figure 6-10 shows such an axis frame centered at the position P_0.

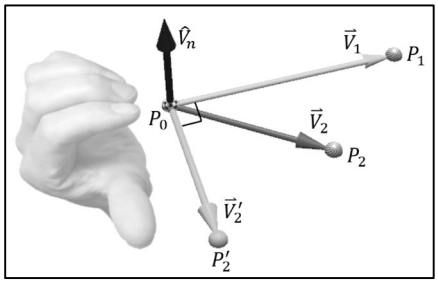

Figure 6-10. *Defining an axis-frame*

In Figure 6-10, the three positions, P_0, P_1, and P_2, define two vectors,

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

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

When these two vectors are not parallel, a new vector, $\vec{V}_n$, that is perpendicular to both $\vec{V}_1$and $\vec{V}_2$ can be computed,

$$\vec{V}_n = \vec{V}_1 \times \vec{V}_2$$

An important observation is that the cross product of $\vec{V}_n$ with $\vec{V}_1$, as indicated by the curling left hand in Figure 6-10, defines, $\vec{V}_2'$,

$$\vec{V}_2' = \vec{V}_n \times \vec{V}_1$$

a vector perpendicular to both $\vec{V}_n$ with $\vec{V}_1$. Notice that $\vec{V}_1$, $\vec{V}_n$, and $\vec{V}_2'$ are three vectors that are mutually perpendicular and is an axis frame that can be located at any position. In more advanced studies of mathematics for video game development, this axis frame can serve as the basis for a new coordinate system, for example, serving to define what can be seen by a camera. Here, the focus will be on defining a 2D region and a general bounding box as an exercise.

Bounds on a 2D Plane

Recall from Figure 5-9 that a general 1D interval, or a line segment, is a direction with two positions along that direction defining the beginning and the ending point of that line segment. Also recall from Figure 2-7 that a 2D interval, or a 2D rectangular region, is two 1D intervals along two perpendicular directions. Figure 6-11 shows two perpendicular general 1D intervals. The first interval is along $\vec{V}_1$, with P_0 and P_1, and the second interval is along $\vec{V}_2'$, with P_0 and P_2' as their beginning and ending positions. The two intervals have respective lengths of L_1 and L_2.

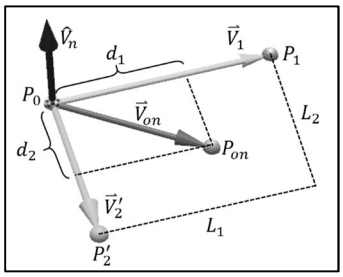

Figure 6-11. *Inside condition of a general 2D region*

You can follow the exact same logic as in Chapter 2 when generalizing results from a 1D interval to a 2D bounding area and apply the logic to a general axis frame. In this case, instead of 1D intervals along the X- and Z-axis, you are working with general 1D intervals along the $\vec{V}_1$ and $\vec{V}_2'$ directions. The inside-outside status of the 2D region can be determined by applying the general 1D test, as illustrated in Figure 5-11 (d), on each of the two perpendicular general 1D intervals. For example, look at the given position P_{on} in Figure 6-11, this position defines the vector $\vec{V}_{on}$,

$$\vec{V}_{on} = P_{on} - P_0$$

The vector, $\vec{V}_{on}$, can be used to determine if the position P_{on} is within the 2D region. In this case, the position P_{on} is within the bounds of the region if the projected size of $\vec{V}_{on}$ along both $\vec{V}_1$ and $\vec{V}_2'$ are positive and smaller than the corresponding interval lengths, or,

$d_1 = \vec{V}_{on} \cdot \hat{V}_1$	Projected size of $\vec{V}_{on}$ along $\hat{V}_1$
$d_2 = \vec{V}_{on} \cdot \hat{V}_2'$	Projected size of $\vec{V}_{on}$ along $\hat{V}_2'$
$0 \leq d_1 \leq L_1$ and $0 \leq d_2 \leq L_2$	Positive and smaller than the interval lengths

Generalization of the Vector Line Equation

Recall the vector line equation that describes all positions located on the line segment that begins from position P_0 and extends in the direction of $\hat{V}_1$, is,

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

In this example, you have observed that the corresponding **vector plane equation**, where all positions that are located in the 2D rectangular region that begins at position P_0 and extends in the perpendiuclar directions of $\hat{V}_1$, and, $\hat{V}_2'$, as,

$$p(d_1, d_2) = P_0 + d_1\hat{V}_1 + d_2\hat{V}_2'$$

Similar to the vector line equation where the range of the value, t, determines the inside-outside status, in 2D region the ranges of the values, d_1, and d_2, determine the inside-outside status of a position. Note the straightforward generalization to the third dimension for a bounding box,

$$b(d_1, d_2, d_3) = P_0 + d_1\hat{V}_1 + d_2\hat{V}_2' + d_3\hat{V}_{\mathrm{n}}$$

The Axis Frames and 2D Regions Example

This example builds on the previous example by supporting two additional features. It demonstrates the derivation of axis frames and the determination of the position inside-outside status for a given 2D region. The example allows you to interactively define an axis frame by manipulating three positions while it continuously computes the inside-outside status of the intersection of a position vector with the 2D plane. Figure 6-12 shows a screenshot of running the `EX_6_3_AxisFramesAnd2DRegions` scene from the `Chapter-6-Examples` project.

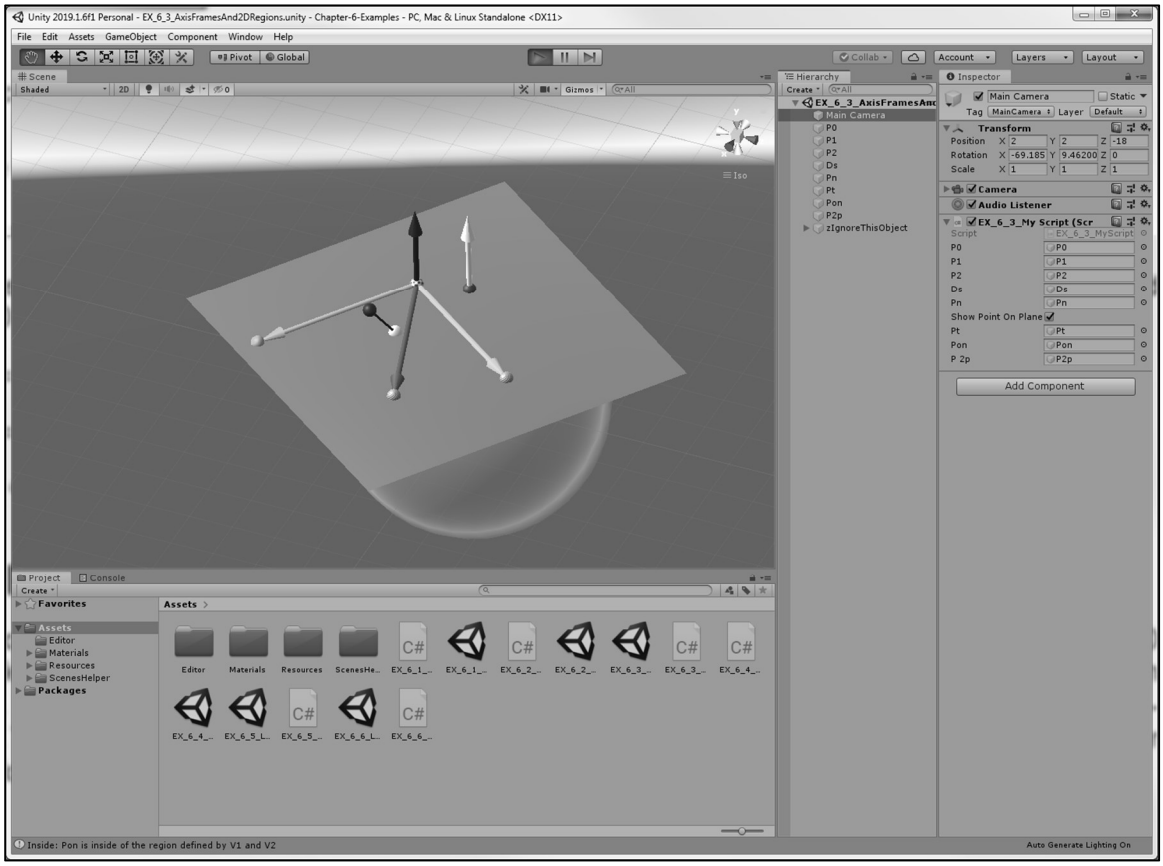

Figure 6-12. *Running the Axis Frames and 2D Regions example*

The goals of this example are for you to:

- Observe the creation of axis frames based on three non-collinear positions
- Appreciate the fact that a 2D region on a plane is indeed defined by two perpendicular 1D regions
- Examine the implementation of the axis frame definition and the inside-outside test for the 2D region

Examine the Scene

Take a look at `Example_6_3_AxisFramesAnd2DRegions` scene, observe the predefined game objects in the Hierarchy Window and note that the only differences between this scene and that of `Example_6_2_VectorPlaneEquations` is a single additional game object, `P2p`. The `transform.localPosition` of this game object will represent the position of P_2' in Figure 6-10, the head position of the $\vec{V}_2'$ vector that is perpendicular to both $\vec{V}_1$ and $\vec{V}_n$. All other game objects serve the same purpose as they did in the previous example.

Analyze MainCamera MyScript Component

The MyScript component on MainCamera also shows that P2p is the only additional variable when compared to the previous example. This new variable is meant to reference the game object with the same name for position manipulation in the script.

Interact with the Example

Click on the Play Button to run the example. Notice the almost identical results of this example to that of the previous example. As a quick reminder, pay attention to the checkered sphere, P0, and the two striped spheres, P1, and P2. These three positions define the two vectors, $\vec{V}_1$ (in cyan), and $\vec{V}_2$ (in magenta), according to Figure 6-10. The black vector at P_0 is, $\vec{V}_n = \vec{V}_1 \times \vec{V}_2$. The blue sphere, P_t, defines the position vector that intersects the plane at P_{on}, the red sphere. The only addition to this scene is the green sphere, P2p, identifying the head position of the $\vec{V}_2'$ vector, where,

$$\vec{V}_2' = \|\vec{V}_2\|(\vec{V}_n \times \vec{V}_1).normalized \qquad \text{size of } \vec{V}_2 \text{ and direction of cross product}$$

Now, select P2 and manipulate its position. Notice how the green vector, $\vec{V}_2'$, has the exact same length as $\vec{V}_2$ and is always perpendicular to $\vec{V}_1$ and $\vec{V}_n$, and that the three vectors, $\vec{V}_1$, $\vec{V}_2'$, and $\vec{V}_n$, do indeed define a valid axis frame with three perpendicular directions centered at P0, independent of where P0 is located, and as long as P0, P1, and P2 are not collinear.

Now restart the scene and select Pt and manipulate its position to move Pon, the red sphere, into the region bounded by $\vec{V}_1$, and $\vec{V}_2'$ by increasing its x-component value. Notice as soon as Pon crosses into the region, its color changes from red to white. As long as Pon is located within the 2D region it will remain white. Feel free to adjust P0, P1, or P2 to change the bounds of the region to verify the inside-outside test is consistent and always correct.

Details of MyScript

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

```
#region Identical to EX_6_2
#endregion
public GameObject P2p;  // The perpendicular version of P2

#region For visualizing the vectors
#endregion

// Start is called before the first frame update
void Start() {
    #region Identical to EX_6_2
    #endregion

    Debug.Assert(P2p != null);

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

As explained, P2p is the only additional variable from an otherwise identical example to the previous subsection. The Update() function is listed as follows.

```
void Update() {
    #region Identical to EX_6_2
    #endregion

    float l1 = v1.magnitude;
```

```
        float l2 = v2.magnitude;
        Vector3 v2p = l2 * Vector3.Cross(vn, v1).normalized;
        P2p.transform.localPosition = P0.transform.localPosition + v2p;

        bool inside = false;
        if (!almostParallel) {
            Vector3 von = Pon.transform.localPosition - P0.transform.localPosition;
            float d1 = Vector3.Dot(von, v1.normalized);
            float d2 = Vector3.Dot(von, v2p.normalized);

            inside = ((d1 >= 0) && (d1 <= l1)) && ((d2 >= 0) && (d2 <= l2));
            if (inside)
                Debug.Log("Inside: Pon is inside of the region defined by V1 and V2");
            else
                Debug.Log("Outside: Pon is outside of the region defined by V1 and V2");
        }
        #region  For visualizing the vectors
        #endregion
    }
```

The first part of the `Update()` function in the collapsed region contains code that is identical to previous example. Recall that the collapsed code computes, $\vec{V}_1, \vec{V}_2, \vec{V}_n,$ and P_{on}. The first four lines of new code derive the vector, $\vec{V}_2'$, of the axis frame and its head position, P_2' ,

$$L_1 = \|\vec{V}_1\|$$

$$L_2 = \|\vec{V}_2\|$$

$$\vec{V}_2' = L_2(\vec{V}_n \times \vec{V}_1).normalized$$

$$P_2' = P_0 + \vec{V}_2'$$

When the `Pt` position vector is not parallel with the plane, `Pon` is defined, and the inside-outside status is computed by the code in the `if` statement,

$$\vec{V}_{on} = P_{on} - P_0$$

$d_1 = \vec{V}_{on} \cdot \hat{V}_1$ Projected size of $\vec{V}_{on}$ along $\hat{V}_1$

$d_2 = \vec{V}_{on} \cdot \hat{V}_2'$ Projected size of $\vec{V}_{on}$ along $\hat{V}_2'$

And finally, the `inside` condition is computed as,

$$inside = (0 \le d_1 \le L_1)\ and\ (0 \le d_2 \le L_2)$$

Take Away from This Example

This example demonstrates that an axis frame can be defined based on three non-collinear positions. Although no actual applications are demonstrated, the ability to derive axis frames is of key importance in supporting many advanced operations in video game development including the support for coordinate transformations.

The generalization of intervals and bounds is now complete. In Chapter 2 you learned about intervals and bounds that are aligned with the major axes. In Chapter 5, you learned to work with general 1D intervals where the interval does not need to be aligned with any major axis. There, you have also learned that if you were given two general 1D intervals that are perpendicular then a general 2D region can be defined for inside-outside tests. The challenge was that you did not know how to derive the two

perpendicular, general, 1D intervals. Now, with the knowledge of axis frame derivation, when given three non-collinear positions, you can compute the two perpendicular general 1D intervals and proceed to define a general 2D region.

Following the 2D to 3D generalization logic from Chapter 2, together with the fact that the derived axis frame provides the third perpendicular vector, you can now define and compute the inside-outside status of any position for bounding boxes at any orientation. However, remember that the collisions of two bounding boxes based on different axis frames are tedious and non-trivial.

Relevant mathematical concepts covered include:

- Three non-collinear positions not only define two non-parallel vectors, they also define an axis frame
- A general 2D rectangular bound can be defined by two general 1D intervals along perpendicular directions
- A position can be projected onto any general 1D interval to determine its inside-outside status

EXERCISES

Implement a General Bounding Box

Modify MyScript *to include a public floating-point variable,* vnSize*. Initialize it to a reasonable value, e.g.,* 3.0*. Use this variable as the size of the third general 1D interval along the* $\vec{V}_n$ *direction. Notice a general bounding box is now defined with the two intervals identified in Figure 6-11. Now, implement the bounding box inside-outside test for* Pt*. You can print out the status and verify the correctness of your implementation.*

Verify the Importance of Cross Product Ordering

Notice that in Figure 6-10, $\vec{V}_2'$ *is defined to be* $\vec{V}_n \times \vec{V}_1$ *and not* $\vec{V}_1 \times \vec{V}_n$*. This is because a Left-Handed Coordinate System axis frame is followed and thus is required. You can verify with your left hand thumb, index, and middle finger, that the proper third vector to the existing* $\vec{V}_n$ *and* $\vec{V}_1$ *must be computed by* $\vec{V}_n \times \vec{V}_1$*. For example, if you align your index finger with* $\vec{V}_n$*, then the middle finger is along the* $\vec{V}_1$ *direction, and your thumb will point in the* $\vec{V}_n \times \vec{V}_1$ *direction. Alternatively, if your index finger is aligned with* $\vec{V}_1$*, then, your*

thumb is in the $\vec{V}_n$ direction, and once again, the middle finger will be in the $\vec{V}_n \times \vec{V}_1$ direction. Now, try reversing the cross product order when computing $\vec{V}_2'$ (the `v2p` variable) and run the game again. Can you explain what you observe?

Projections onto 2D Planes

In video games and many interactive graphical applications, it is a common practice to drop shadows of objects in space to convey hints of relative spatial location. For example, dropping the shadow of an in-flight meteoroid on the grounds of the approaching city or casting the shadow of an amulet tossed by the explorer on the walls of secret chamber to help better track its movement. In these cases, the shadows will convey a clear sense of the actual location of the in-flight objects and will allow the player to strategize their next move and react. Figure 6-13 shows that the shadow casting functionality can be modeled as a point to plane projection problem.

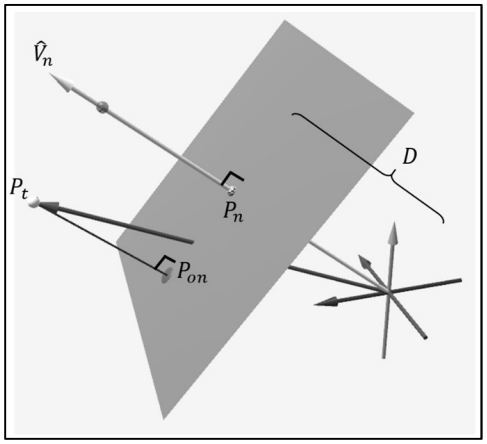

Figure 6-13. *Projection of a point onto a plane, or, casting shadow onto the plane*

Figure 6-13 shows a plane defined by the plane normal vector, $\hat{V}_n$, located at a distance, D, away from the origin. You know that the vector plane equation for this plane is,

$$\hat{V}_n \cdot p = D$$

where,

$$P_n = D\hat{V}_n$$

In Figure 6-13, P_t is the position of the object in-flight, and P_{on} is the projection of P_t on the given plane. Note that this projection is along the line connecting P_t to P_{on}, where the projection direction is parallel to the plane normal, $\hat{V}_n$. Figure 6-14 includes the following additional explanation for the derivation of point to plane solution.

$d = P_t \cdot \hat{V}_n$ the projected size of position vector P_t onto $\hat{V}_n$

$P_l = d \cdot \hat{V}_n$ the projected position of P_t on $\hat{V}_n$

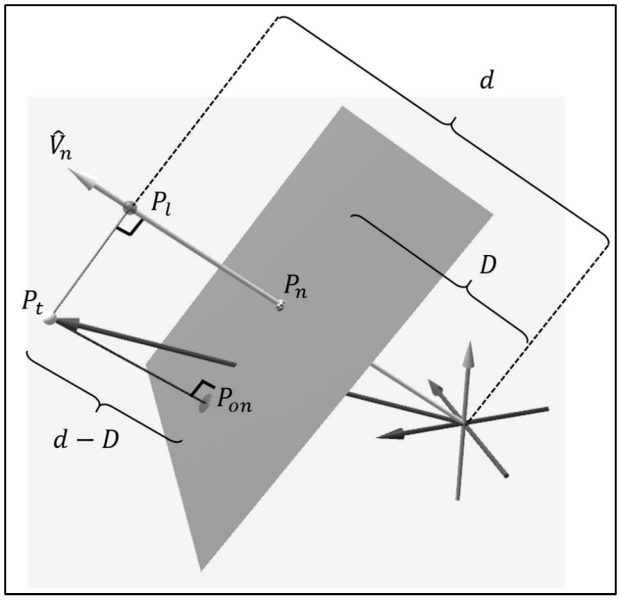

Figure 6-14. *Solving for point to plane projection*

The solution of point to plane projection can be explained by referring to Figure 6-14 and observing the followings:

- First, a decision is made that a projection will only occur if position P_t is in front of the plane. This condition is true when the projected length of the P_t position vector in the $\hat{V}_n$ direction is greater than the plane distance, D, or, if $d > D$
- Second, because the projection is along the $\hat{V}_n$ direction, the distance between P_l and P_n is the same as the distance between P_t and P_{on}, and this distance is simply $d - D$
- Finally, P_{on}, is $d - D$ distance away from P_t in the negative $\hat{V}_n$ direction, or,

$$P_{on} = P_t - (d - D)\hat{V}_n$$

■ ***Note*** *The derived solution for the point projection is valid for* Pt *located on either side of the plane. In this case, projection is restricted to one of the sides of the plane to showcase the "in front of" test. Modifying the solution to support proper projections for all locations of* Pt *is left as an exercise for you to complete.*

The Point to Plane Projections Example

This example demonstrates the results of point to plane projection computation. The example allows you to interactively define a 2D plane, manipulate the point to be projected, and examine the results of projecting the point onto the plane. Figure 6-15 shows a screenshot of running the EX_6_4_PointToPlaneProjections scene from the Chapter-6-Examples project.

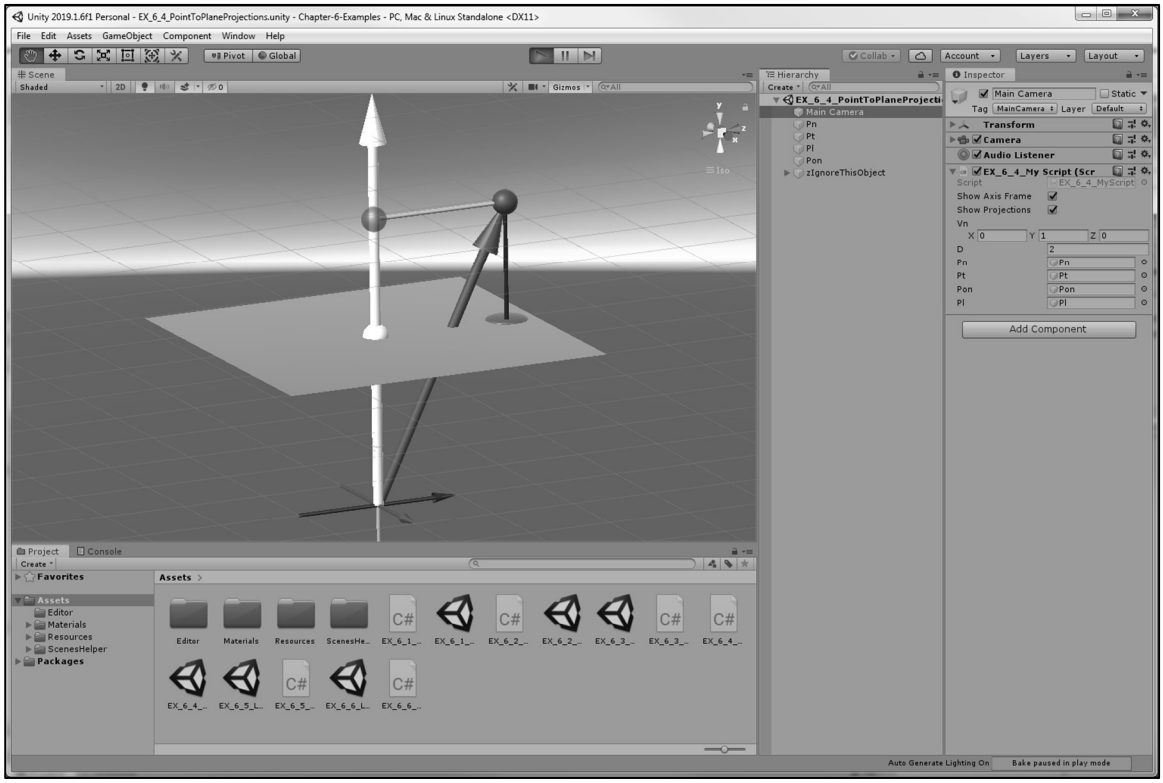

Figure 6-15. *Running the Point to Plane Projections example*

The goals of this example are for you to:

- Gain experience with the "in front of a plane" test
- Verify the solution of point to plane projection
- Examine the implementation of the in front of a plane test and point to plane projection
- Observe the elegance and simplicity of typical implementation of vector solutions

Examine the Scene

Take a look at the Example_6_4_PointToPlaneProjections scene and observe the predefined game objects in the Hierarchy Window. In addition to MainCamera, there are four objects in this scene: Pn, Pt, Pl, and Pon. Following the illustration in Figure 6-14, Pn is the

position vector along the plane normal that intersects the 2D plane, Pt is the position to be projected, Pl is the projection of Pt on the plane normal vector, and, Pon is the projection of Pt on the plane.

Analyze MainCamera MyScript Component

The MyScript component on MainCamera shows three sets of variables as follows.

- Display toggles: ShowAxisFrame and ShowProjections will show or hide the axis frame and the projections accordingly. These toggle switches are meant to assist your visualization, allowing you to hide the illustration vectors to avoid screen cluttering.
- Vector plane equation parameters: Vn and D are the plane normal vector and the distance of the plane from the origin along the normal vector direction and will be used to create and modify the plane.
- Variables for the positions: Pn, Pt, Pl, and Pon are variables with names that correspond to the game objects in the scene. For all these game objects, the transform.localPosition will be used for the manipulation of their corresponding positions.

Interact with the Example

Click on the Play Button to run the example. The white sphere is Pn, the white vector is $\hat{V}_n$, the red sphere is Pt, and the red vector is the position vector P_t. The semi-transparent black sphere on the white vector or the projected position on the plane normal vector, is Pl, while the semi-transparent blob on the 2D plane or the projected position on the plane, is Pon. Notice the thin green line connecting Pt to Pl, since Pl is the projection of Pt onto the plane normal vector, this line is always perpendicular to the plane normal and parallel to the plane. The thin black line connecting Pt to Pon represents the projection of Pt onto the plane and thus is always perpendicular to the plane and parallel to $\hat{V}_n$. In the following interactions, feel free to toggle off either or both of the display toggles to declutter the Scene View.

With the scene running, first verify the "in front of plane" test by selecting Pt and decreasing its y-component value. Notice that as soon as Pt is below the 2D plane, the projected positions disappear, verifying that the projection computation is only performed when the point, Pt, is in front of the plane. You can also verify this test by manipulating the D or Vn variables to move the plane or rotate the plane normal vector. Notice once again, as soon as Pt drops below the plane, the projected positions will both disappear.

Feel free to manipulate Pt or the plane parameters D or Vn in any way you like. Pay attention to the in front of plane test result, and the consistent perpendicular relationships between the green line and the white $\hat{V}_n$ vector, and the black line and the 2D plane.

Details of MyScript

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

```
public bool ShowAxisFrame = true;
public bool ShowProjections = true;

// Plane Equation: P dot Vn = D
public Vector3 Vn = Vector3.up;
public float D = 2f;

public GameObject Pn = null;
public GameObject Pt = null;  // The point to be projected onto the plane
public GameObject Pl = null;  // Projection of Pt on Vn
public GameObject Pon = null; // Projeciton of Pt on the plane
```

```
#region For visualizing the vectors
#endregion

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

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

All the public variables for MyScript have been discussed when analyzing the 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() {
    Vn.Normalize();
    Pn.transform.localPosition = D * this.Vn;
    bool inFront = (Vector3.Dot(Pt.transform.localPosition , Vn) > D);  // Pt infront of the plane

    Pon.SetActive(inFront);
    Pl.SetActive(inFront);
    float d = 0f;
    if (inFront) {
        d = Vector3.Dot(Pt.transform.localPosition, Vn);
        Pl.transform.localPosition = d * Vn;
        Pon.transform.localPosition = Pt.transform.localPosition - (d - D) * Vn;
    }
    #region For visualizing the vectors
    #endregion
}
```

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

$\hat{V}_n = \|\vec{V}_n\|$	normalize the user specified $\vec{V}_n$
$P_n = D\hat{V}_n$	position at D distance along $\hat{V}_n$
$infront = (P_t \cdot \hat{V}_n) > D$	P_t is further away than D along $\hat{V}_n$

The if condition checks for when P_t is indeed in front of the plane. When the condition is favorable,

$d = P_t \cdot \hat{V}_n$	the projected size of position vector P_t onto $\hat{V}_n$
$P_l = d \cdot \hat{V}_n$	the projected position of P_t on $\hat{V}_n$
$P_{on} = P_t - (d - D)\hat{V}_n$	$d - D$ distance from P_t in the $-\hat{V}_n$ direction

Notice the exact one-to-one implementation code when compare with the solution derivation. Once again, the implementation of vector solutions is typically simple, elegant, and closely matches the mathematic derivation.

Take Away from This Example

This example demonstrates an efficient and graceful way to drop shadows which is a commonly encountered situation in video games. The example also demonstrates that the vector solution to projecting along a 2D plane normal is straightforward, stable, and involves a small number of lines of code. Additionally, the example shows how dot product results can be used to determine the in front of or behind relationship between an object position and a given 2D plane.

Relevant mathematical concepts covered include:

- An object is in front of a given plane when the dot product of the object's position vector with the plane normal is greater than the plane distance
- The projection of a position to a given plane is a subtraction of the position vector by a perpendicular distance to the plane, along the plane normal

EXERCISES

Projection Support for Both Sides of the Plane

Notice that the derivation and the vector solution for projection is valid independent of whether Pt is in front of or behind the plane. The analysis of MyScript actually demonstrated extra computation to purposefully hide the projection results when Pt is not in front of the plane. Modify MyScript to disable the in front of check and verify that the projection solution is indeed valid for all positions of Pt.

Criteria for Shadow Casting

The result of the "in front of test" is binary--an object is either in front of the plane or not. In this example, an object can either cast shadow, or, the object cannot cast shadow. Notice that the result from the dot product performed, $(P_t \cdot \hat{V}_n)$, encodes more information than just in front of or not. The result also tells you the projected distance, or, if P_t is normalized, the cosine of the subtended angle. This information can be used to refine the criterial of when shadow casting should occur. For example, casting a shadow should only happen when the subtended angle is within a certain range. Now, modify

MyScript to compute the subtended angle and allow shadows to be casted only when the subtended angle is less than a degree that is under the user's control.

Characteristics of the Shadow Casted

The shadow casted on the 2D plane contains attributes of its object that can also be refined according to the additional information from the projection computation. For example, the projected size on the plane normal, $(P_t \cdot \hat{V}_n)$, carries the height information of the object. This value can be used to scale the size and the transparency of the shadow object. Modify MyScript to compute and use the length of the projected size to scale the size of the Pon game object.

Let User Manipulate Pn

The very simple relationship between Pn, D, and Vn,

$$P_n = D\hat{V}_n$$

states that a user can also define the plane by manipulating Pn instead of D and Vn. In such a case,

$$D = \|P_n\|$$
$$\hat{V}_n = \frac{P_n}{D}$$

Notice that with this approach, instead of the four floating-point numbers, D, and the x-, y-, and z-components of Vn, the user only has the three floating-point components of Pn to manipulate the 2D plane. While, this is easier for the user, it also means that the user cannot define planes with D of zero. With this caveat in mind, please modify MyScript to allow the user the option of defining the 2D plane with either approach.

Projection with 2D Bound Inside-Outside Test

Notice that as you move Pt *in the X- and Z-axis directions, the size of the plane adapts and continuously shows the projected position on the plane. In an actual application, a 2D bound would be defined on this plane and an inside-outside test could be performed and projected positions outside of the 2D bound would simply be ignored. Refer to the previous example where, instead of allowing the users to adjust* Vn *and* D *to define the plane, three positions,* P0*,* P1*, and* P2 *are used to define both the plane and an axis frame and then a 2D bound. Adapt the solution and support bound testing for the projected position.*

■ ***Note*** *The last exercise challenges you to replace the* Vn *and* D *parameters with three positions to define the 2D plane and an axis frame. In practice, such extra efforts are not necessary. This is because an axis frame is actually conveniently defined by the initial orientation of the 2D plane and the plane normal vector,* Vn*. This information is available in the rotation matrix of the plane's transform component. However, more advanced knowledge in vector transformations and matrix algebra are required to decode this information. Unfortunately, these are topics beyond the scope of this book. For now, if you want to define an axis frame on a 2D plane, the plane must be defined by three positions that are not collinear. In the rest of the examples in this chapter, 2D plane sizes are always adapting to include the projected or intersected positions as these planes are created using the plane equation which relates better to the math at hand.*

Line to Plane Intersection

You may recall that at the end of Chapter 2's discussion of bounds, when comparing what you have learned with the Unity Bounds class, one of the methods whose details was not discussed was,

- IntersectRay: Does ray intersect this bounding box?

You are now in a position to closely examine this function. By now, you know that a ray is simply a line segment. The IntersectRay() function computes and returns the closest intersection position between a line segment and the six sides of the bounding box. Note that each side of a bounding box is simply a 2D region as you have previously examined in the Axis Frames and 2D Regions example. The IntersectRay() function answers the question of how to intersect a line segment with a 2D plane. This solution is illustrated in Figure 6-16.

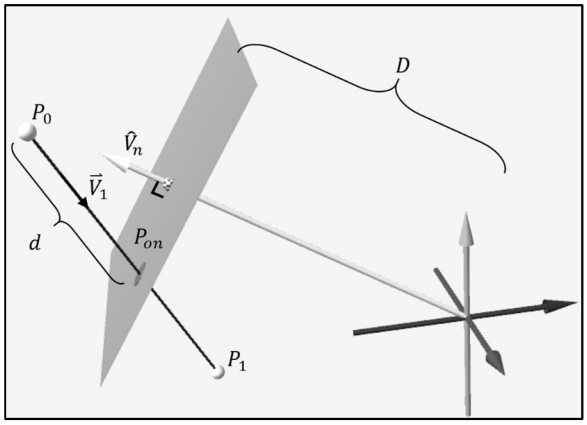

Figure 6-16. *Solving the line-plane interesection*

Figure 6-16 depicts two positions, P_0 and P_1, that define a vector $\vec{V}_1$,

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

and the positions, p, along the line segment, with parameter s, that can be written as,

$$p = P_0 + s\vec{V}_1$$

Notice that in this formulation, since the $\vec{V}_1$ vector is not normalized, s values between 0 and 1, or when $0 \leq s \leq 1$, identify positions that are inside the line segment. In Figure 6-16, the position P_{on} is at a distance, $s = d$, along the $\vec{V}_1$ vector, or,

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

Remember that the vector plane equation states that, given a plane defined by normal vector, $\hat{V}_n$, and a distance, D, from the origin, all positions, p, on the plane are identified as,

$$p \cdot \hat{V}_n = D$$

In Figure 6-16, the position P_{on} lies on the 2D plane, so,

$$P_{on} \cdot \hat{V}_n = D$$

$$(P_0 + d\vec{V}_1) \cdot \hat{V}_n = D \qquad \text{substitute } P_{on} = P_0 + d\vec{V}_1$$

Note that the only unknown in this equation is d, the distance to travel along the line segment. By simplifying this equation, left as an exercise, you can show that,

$$d = \frac{D - (P_0 \cdot \hat{V}_n)}{(\vec{V}_1 \cdot \hat{V}_n)}$$

With the d value computed; you can now find the exact P_{on} position. Note that this solution is not defined when the denominator, or $(\vec{V}_1 \cdot \hat{V}_n)$, is close to zero. Once again, this can be explained by your knowledge of the dot product. A dot product result of zero means that the cosine of the subtended angle is zero, which says the subtended angle is $90°$ or that the two vectors are perpendicular. These observations indicate that when $(\vec{V}_1 \cdot \hat{V}_n)$ is close to zero, vectors $\vec{V}_1$ and $\hat{V}_n$ are almost perpendicular, the line segment is almost parallel to the plane, and therefore there can be no intersection between the two.

■ **Note** *Ray casting is the process of intersecting a line segment, or a ray, with geometries. For example, if you were told to "cast a ray into a scene", then you would simply intersect geometries in the scene with a given line segment. In this case, you are learning about ray casting with a 2D plane.*

The Line Plane Intersections Example

This example demonstrates the results of the line plane intersection solution. The example allows you to interactively define a 2D plane and a line segment and then examine the results of the line plane intersection computation. Figure 6-17 shows a screenshot of running the `EX_6_5_LinePlaneIntersections` scene from the `Chapter-6-Examples` project.

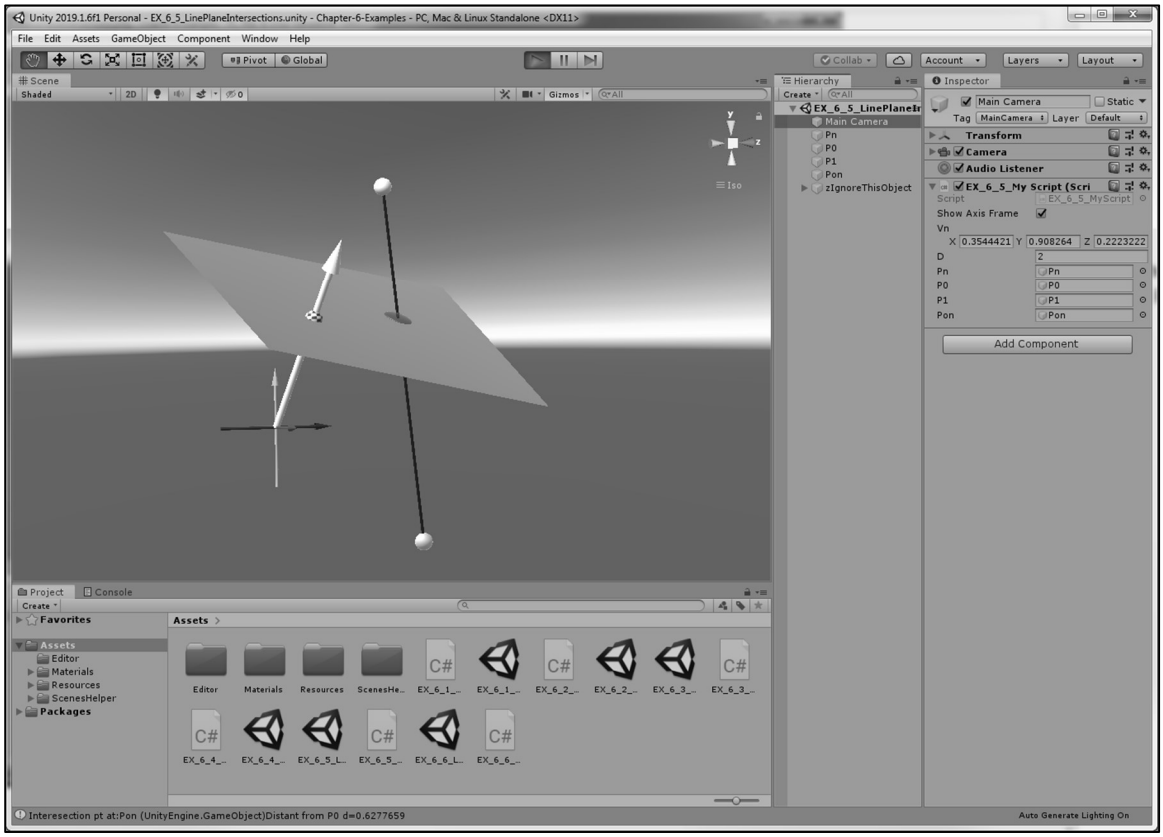

Figure 6-17. *Running the Line Plane Intersections example*

The goals of this example are for you to:

- Verify the line plane intersection solution
- Gain experience with the perpendicular vectors test
- Reaffirm that it is important to check for all conditions when a solution is not defined, in this case, when the line segment is parallel to the plane
- Examine the implementation of the line plane intersection solution

Examine the Scene

Take a look at the `Example_6_5_LinePlaneIntersections` scene and observe the predefined game objects in the Hierarchy Window. In addition to `MainCamera`, there are four objects in this scene: `Pn`, `P0`, `P1`, and `Pon`. Where `Pn`, the checkered sphere, is the position on the plane that is at the defined distance, `D`, along the plane normal. This position is displayed to assist in visualizing the 2D plane. The positions `P0` and `P1` define the black line segment, and `Pon` is the intersection position between this line segment and the defined plane.

Analyze MainCamera MyScript Component

The MyScript component on MainCamera shows three sets of variables as follows.

- Display toggles: ShowAxisFrame will show or hide the axis frame to assist your visualization, allowing you to hide the axis frame to avoid screen cluttering.
- Vector plane equation parameters: Vn and D are the plane normal vector and the distance of the plane from the origin along the normal vector direction and will be used to create and modify the plane.
- Variables for the positions: Pn, P0, P1, and Pon are variables with names that correspond to the game objects in the scene. For all these game objects, the transform.localPosition will be used for the manipulation of their corresponding positions.

Interact with the Example

Click on the Play Button to run the example. You can observe a 2D plane with a white normal vector extending from the origin and passing through the plane at Pn. You can also observe a thin black line between the positions P0 and P1 that define the line segment. At the intersection of the plane and the line segment is position Pon. You should be familiar with the 2D plane and its parameters, Vn and D.

Select the end points of the line segment, P0 or P1, and adjust its x- and z-component values. Observe that Pon changes in response to your manipulation, always locating itself at the line plane intersection. This verifies the solution you have derived for Pon. You can verify the intersection computation results by referring to the text output in the Console Window. Remember, the values for the d parameterization (see Figure 6-16 for a reminder of what this variable is) is based on a non-normalized vector, therefore d values between 0 and 1 indicate that Pon is inside the line segment.

Now, select P0 and increase its y-component value. When P0's position is above the plane, the Pon position is still along the line, but is outside of the line segment, occurring before position P0. This fact is reflected by the red line segment between Pon and P0. Notice that as you continue to increase the P0 y-component value, as the line segment comes close to being parallel to the plane, the intersection position is located at positions further and further away from Pn. Eventually, when P0 and P1 y-component values are exactly the same, the line segment and the plane are exactly parallel and therefore there is no intersection between the two. You can verify this condition by referring to the printout in the Console Window. If you continue to increase the P0 y-component value, you will notice the red-line segment switching between P0 to Pon to between P1 and Pon. In the case when P0 is above P1, the intersection position is along the line segment and after position P1. When this occurs, the value of d will be greater than 1 which you can verify has happened via the Console Window.

Feel free to manipulate all of the parameters, Vn, D, P0, and P1, and verify that the line plane intersection solution does indeed compute a proper Pon result except when the line is almost parallel to the plane, or when the length of the line is very small (when P0 an P1 are located at almost 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.

```
public bool ShowAxisFrame = true;

// Plane Equation: P dot Vn = D
public Vector3 Vn = Vector3.up;
public float D = 2f;
public GameObject Pn = null;  // The point where plane normal passes

public GameObject P0 = null, P1 = null;  // The line segment
```

```
public GameObject Pon = null;  // The intersection position

#region For visualizing the vectors
#endregion

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

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

All the public variables for MyScript have been discussed when analyzing the 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() {
    Vn.Normalize();
    Pn.transform.localPosition = D * Vn;

    // Compute the line segment direction
    Vector3 v1 = P1.transform.localPosition - P0.transform.localPosition;
    if (v1.magnitude < float.Epsilon) {
        Debug.Log("Ill defined line (magnitude of zero). Not processed");
        return;
    }

    float denom = Vector3.Dot(Vn, v1);
    bool lineNotParallelPlane = (Mathf.Abs(denom) > float.Epsilon);  // Vn is not perpendicular with V1
    float d = 0;

    Pon.SetActive(lineNotParallelPlane);
    if (lineNotParallelPlane) {
        d = (D - (Vector3.Dot(Vn, P0.transform.localPosition))) / denom;
        Pon.transform.localPosition = P0.transform.localPosition + d * v1;
        Debug.Log("Interesection pt at:" + Pon + "Distant from P0 d=" + d);
    } else {
        Debug.Log("Line is almost parallel to the plane, no interesection!");
    }
}
```

The first two lines of the Update() function normalize the user specify plane normal vector and computes Pn's position to help the user better visualize the 2D plane. The code that follows computes,

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

and checks to ensure that this line segment is well defined and has a length of greater than almost zero. When the line is well-defined, the denominator for the solution to d, $\vec{V}_1 \cdot \hat{V}_n$, is computed and the condition for the line being parallel to the plane is checked. Note the use of the absolute value function when checking for the perpendicular condition. This is because the subtended angles of $89.99°$ and $90.01°$ are both almost perpendicular and the cosine, or the dot product, results are both close to zero but with different signs. Finally, d is computed and printed out to the Console Window when the line is not almost parallel to the plane.

Take Away from This Example

This example demonstrates the solution to the line to plane intersection, an important problem that is straightforward to solve based on vector concepts you have learned. The concepts applied including working with the vector plane equation, the sign of the vector dot product, vector projections, and fundamental vector algebra. The line to plane intersection is a core functionality that can be found in typical game engine utility libraries. In the case of Unity, this functionality is presented via the `IntersectRay( )` function of the `Bounds` class.

Relevant mathematical concepts covered include:

- Two vectors are almost perpendicular when the result of their dot product is close to zero
- When a line is almost perpendicular to the normal of a plane, it is almost parallel to the plane
- The intersection point of a line and a plane can be derived based on vector algebra

Relevant observations on implementation include:

- Testing for perpendicular vectors, or when dot product result is close to zero, must be performed via the absolute value function, as very small positive and negative numbers are both close to zero

EXERCISES

Verify the Line Plane Intersection Equation

Recall that in Figure 6-16, the position P_{on} is at a distant, $s = d$, along the $\vec{V}_1$ vector, or,

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

You have observed that since this position is also on the 2D plane,

$$(P_0 + d\vec{V}_1) \cdot \hat{V}_n = D$$

Now, apply the distributive property of the vector dot product over the vector addition operation and, remembering that the result of a dot product is a floating-point number, show that,

$$d = \frac{D-(P_0 \cdot \hat{V}_n)}{(\vec{V}_1 \cdot \hat{V}_n)}$$

A More General Shadow Casting Solution

One approach to interpret Figure 6-16 is to ignore P_1 and interpret P_{on} as the projection of P_0 on the 2D plane along the $\vec{V}_1$ direction. Given this interpretation, you can now cast shadows of objects onto a 2D plane along any direction specified by the user. Modify MyScript to replace P1 by a 3D projection direction and implement the functionality of casting a shadow of P_0 on the plane along the player specified projection direction ($\vec{V}_1$).

Ray casting or intersecting the General Bounding Box

Refer to your solution from the "Implement a General Bounding Box" exercise from the Axis Frames and 2D Regions section. With the results from line plane intersection, you can now implement the IntersectRay( ) function. Modify your solution to this previous exercise by allowing your user to define a line segment and then compute the intersection of the line segment with all six sides of the bounding box. The intersection position between the ray, or line segment, and the bounding box is simply the closest of all the valid intersection positions.

Mirrored Reflection across a Plane

The intersection computation from the previous subsection allows you to collide an incoming object with flat planes or walls. In many video games, a typical response to the results of collisions is to reflect the colliding object. For example, when an amulet is tossed by an explorer, it should bounce and reflect off walls or the floor when it collides with them to convey some sense of realism. This reflection is depicted in Figure 6-18 and can be described as reflecting the velocity of an incoming object in the mirrored reflection direction.

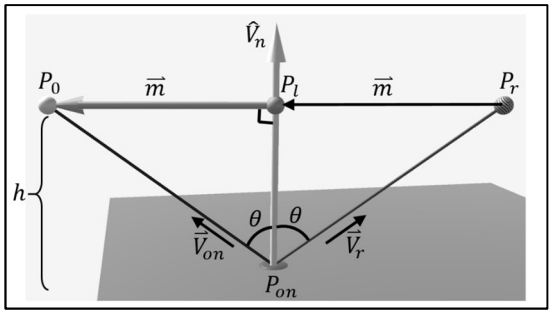

Figure 6-18. *Mirrored reflection across a plane*

In Figure 6-18, P_0, on the left, is the incoming object approaching the plane with normal vector $\hat{V}_n$, and is about to collide with the plane at position P_{on}. P_r is the mirrored reflection of P_0 across the plane normal $\hat{V}_n$, and is the unknown that must be computed.

Since this is a mirrored reflection, the right angle triangle formed by the incoming object, $P_0P_{on}P_l$ is identical to the one formed by the reflected position, $P_rP_{on}P_l$, where, P_l is the position that both P_0 and P_r would project onto in the $\hat{V}_n$ direction. Additionally, the vector, $\vec{m}$, from P_l to P_0 is identical to the vector from P_r to P_l. Given these observations, as illustrated in Figure 6-18, you can define the vector $\vec{V}_{on}$ from P_{on} to P_0,

$$\vec{V}_{on} = P_0 - P_{on} \qquad \text{vector from } P_{on} \text{ to } P_0$$

Project vector $\vec{V}_{on}$ onto the plane normal direction, $\hat{V}_n$, to compute the length of $\vec{V}_{on}$ when measured along the $\hat{V}_n$ direction,

$$h = \vec{V}_{on} \cdot \hat{V}_n \qquad \text{length of } \vec{V}_{on} \text{ along the } \hat{V}_n \text{ direction}$$

Compute, P_l, the projected position of P_0 on the plane normal, $\hat{V}_n$. This position is traveling from P_{on} along the $\hat{V}_n$ direction by the projected distance, h,

$$P_l = P_{on} + h\hat{V}_n \qquad \text{projected position is travel from } P_{on} \text{ along } \hat{V}_n \text{ by } h \text{ distance}$$

With the P_l position, you can compute, $\vec{m}$, the vector from P_l to P_0,

$$\vec{m} = P_0 - P_l \qquad \text{vector from } P_l \text{ to } P_0$$

And finally, the mirrored reflection position of P_0 across the normal vector $\hat{V}_n$ is simply traveling along the negative $\vec{m}$ vector from P_l,

$$P_r = P_l - \vec{m} \qquad P_r \text{ is traveling by the negative } \vec{m} \text{ vector}$$

In these steps, you have derived the reflected position, P_r, of the incoming position P_0 with plane normal $\hat{V}_n$ and collision position P_{on}.

The Reflection Direction

The derived solution for P_r can be organized to assist the interpretation of mirrored reflection geometrically.

$$\begin{aligned}
P_r &= P_l - (P_0 - P_l) && \text{substitute } \vec{m} = P_0 - P_l \\
&= 2P_l - P_0 && \text{collecting the two } P_l \text{ terms} \\
&= 2\left(P_{on} + h\hat{V}_n\right) - P_0 && \text{substitute } P_l = P_{on} + h\hat{V}_n \\
&= 2P_{on} + 2h\hat{V}_n - P_0 && \text{distributive property over floating-point number 2} \\
&= P_{on} + 2h\hat{V}_n - (P_0 - P_{on}) && \text{group } P_{on} \text{ with } P_0 \\
&= P_{on} + 2h\hat{V}_n - \vec{V}_{on} && \text{substitute } \vec{V}_{on} = P_0 - P_{on} \\
&= P_{on} + 2(\vec{V}_{on} \cdot \hat{V}_n)\hat{V}_n - \vec{V}_{on} && \text{substitute } h = \vec{V}_{on} \cdot \hat{V}_n
\end{aligned}$$

Note that this last equation may seem complex, however, it is actually in a simple form. If you define the vector, $\vec{V}_r$ to be,

$$\vec{V}_r = 2(\vec{V}_{on} \cdot \hat{V}_n)\hat{V}_n - \vec{V}_{on}$$

Then,

$P_r = P_{on} + \vec{V}_r$ P_r is arrived by traveling from P_{on} along the $\vec{V}_r$ vector

Refer to Figure 6-18, this is the exact compliment to the incoming position, P_0,

$P_0 = P_{on} + \vec{V}_{on}$ P_0 is arrived by traveling from P_{on} along the $\vec{V}_{on}$ vector

In this way, given an incoming direction of $\vec{V}_{on}$ and the normal vector $\hat{V}_n$, the reflected direction, $\vec{V}_r$, is,

$$\vec{V}_r = 2(\vec{V}_{on} \cdot \hat{V}_n)\hat{V}_n - \vec{V}_{on}$$

This is the **reflection direction equation**. Note that this equation says, the reflected direction, $\vec{V}_r$, is a function of only two parameters--the incoming direction,$\vec{V}_{on}$, and the normal direction, $\hat{V}_n$, that defines the reflection.

Lastly, it is important to note that in this derivation, the incoming direction, $\vec{V}_{on}$, is defined as a vector pointing away from the intersection position (see the arrow above $\vec{V}_{on}$ in Figure 6-18 for clarification). This convention of defining all vectors pointing *away* from the position of interest is a common practice in many video games and computer graphics related vector solutions.

The Line Reflections Example

This example demonstrates the results of line reflection across a 2D plane. This example allows you to interactively define the line segment and the 2D plane, as well as examine the results of reflecting the line segment across the normal direction of the 2D plane. Figure 6-19 shows a screenshot of running the `EX_6_6_LineReflections` scene from the `Chapter-6-Examples` project.

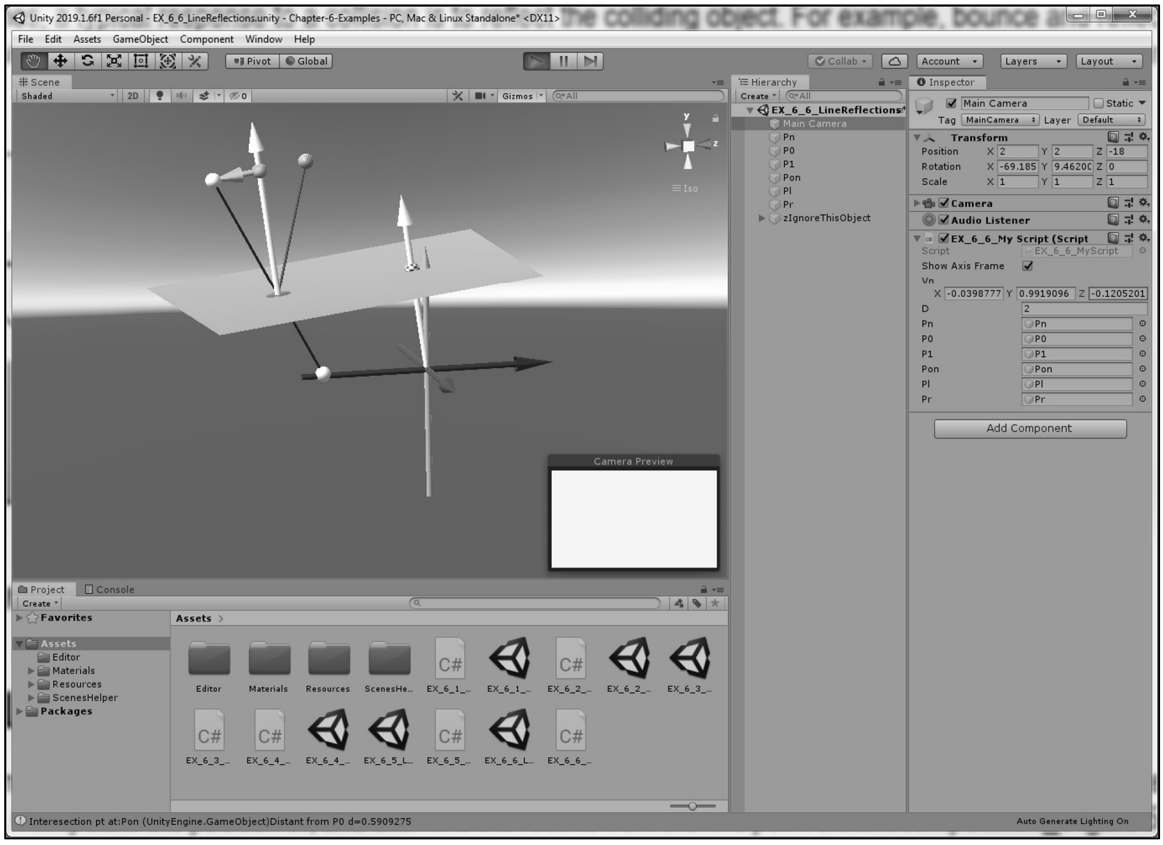

Figure 6-19. *Running the Line Reflections example*

The goals of this example are for you to:

- Verify the reflection direction equation
- Examine the reflection of a position across the normal of a plane
- Examine the implementation of the reflection computation

Examine the Scene

Take a look at the `EX_6_6_LineReflections` scene and observe the predefined game objects in the Hierarchy Window. Take note that this example builds directly on the results from the `EX_6_5_LinePlaneIntersects` scene. Similar to the previous example, the parameters, `Vn` and `D`, define the 2D plane where `Pn` is the position on the plane to assist visualization. The parameters `P0` and `P1` define the line segment, and `Pon` is the intersection between the line and the 2D plane.

The two new game objects in this scene are the projection of `P0` on the plane normal vector, `Pl`, and `Pr` the mirrored reflection of `P0` across the plane normal.

Analyze MainCamera MyScript Component

The MyScript component on MainCamera shows that there are two additional public variables with names that correspond to the Pl and Pr game objects. As in previous cases, the transform.localPosition of these variables will be used for the manipulation of the corresponding positions.

Interact with the Example

Click on the Play Button to run the example. When compared with the Scene View of EX_6_5_LinePlaneIntersects, you will observe the similar 2D plane defined by Vn and D, the thin black line segment defined by P0 and P1, and their intersection at Pon. Note that the plane normal vector is copied and displayed at Pon to assist in the visualization of reflection. Also note that the green sphere, Pl, is the projection of P0 onto the plane normal, and the green vector is the $\vec{m}$ vector as depicted in Figure 6-18,

$$\vec{m} = P_0 - P_l$$

The striped sphere, Pr, connected with a thin red line to Pon, is the mirrored reflection of P0 across the plane normal vector.

Tumble the Scene View camera to examine the running scene from different viewing position to verify that the red line segment and the black line segment above the plane are indeed mirrored reflections. Notice Pl is the projection of P0 onto the normal vector, and thus, the green $\vec{m}$ vector is always perpendicular to the plane normal vector. You can manipulate the plane, by adjusting Vn and D, and the line segment, by adjusting P0, and P1, to verify that the reflection solution is correct for all cases. Recall from the previous example to be careful when the line segment is almost parallel to the plane as the plane size will increase drastically to accommodate the intersection position that will now be located at a very far distance.

You can set P0 and P1 such that the line segment is in the same direction as the plane normal. Observe that in this case, the reflection direction would be parallel to the normal vector direction, and that the projected position, Pl, and the reflected position Pr, will be located at the same point. In other words, the reflection vector would be exactly the same as in the incoming vector!

Details of MyScript

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

```
#region identical to EX_6_5
#endregion
public GameObject Pl = null;  // Projection of P0 on Vn
public GameObject Pr = null;  // reflected position of P0

#region For visualizing the vectors
#endregion

// Start is called before the first frame update
void Start() {
    #region identical to EX_6_5
    #endregion
    Debug.Assert(Pl != null);
    Debug.Assert(Pr != null);

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

As explained, Pl and Pr are the only additional variables from an otherwise identical example to the previous subsection and as in all previous examples, the Debug.Assert() calls in the Start() function ensure proper setup regarding referencing these game objects via the Inspector Window. The Update() function is listed as follows.

```
void Update() {
    #region identical to EX_6_5
    #endregion

    float h = 0;
    Vector3 von, m;
    Pr.SetActive(lineNotParallelPlane);
    if (lineNotParallelPlane) {
        von = P0.transform.localPosition - Pon.transform.localPosition;
        h = Vector3.Dot(von, Vn);
        Pl.transform.localPosition = Pon.transform.localPosition + h * Vn;
        m = P0.transform.localPosition - Pl.transform.localPosition;
        Pr.transform.localPosition = Pl.transform.localPosition - m; ;
        Debug.Log("Incoming object position P0:" + P0.transform.localPosition +
                                " Reflected Position Pr:" + Pr.transform.localPosition);
    } else {
        Debug.Log("Line is almost parallel to the plane, no reflection!");
    }

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

Recall that previous example computes the intersection position, Pon, when the line segment is not almost parallel to the 2D plane. Similar to line plane intersection, a line can only reflect off a plane that it is not parallel with. The if condition checks for the parallel condition and outputs a warning message to the Console Window. Otherwise, the five lines inside the if condition follow the P_r position derivation exactly and compute,

$\vec{V}_{on} = P_0 - P_{on}$	vector from P_{on} to P_0
$h = \vec{V}_{on} \cdot \hat{V}_n$	length of $\vec{V}_{on}$ along the $\hat{V}_n$ direction
$P_l = P_{on} + h\hat{V}_n$	projected position is travel from P_{on} along $\hat{V}_n$ by h distance
$\vec{m} = P_0 - P_l$	vector from P_l to P_0
$P_r = P_l - \vec{m}$	P_r is traveling by the negative $\vec{m}$ vector

Take Away from This Example

This example once again illustrates a straightforward but important application of vector algebra. Note that the reflection direction equation,

$$\vec{V}_r = 2(\vec{V}_{on} \cdot \hat{V}_n)\hat{V}_n - \vec{V}_{on}$$

is independent of plane to origin distance, D, or, the actual incoming object position, P_0, or intersection position P_{on}. As depicted in Figure 6-20, this makes intuitive sense.

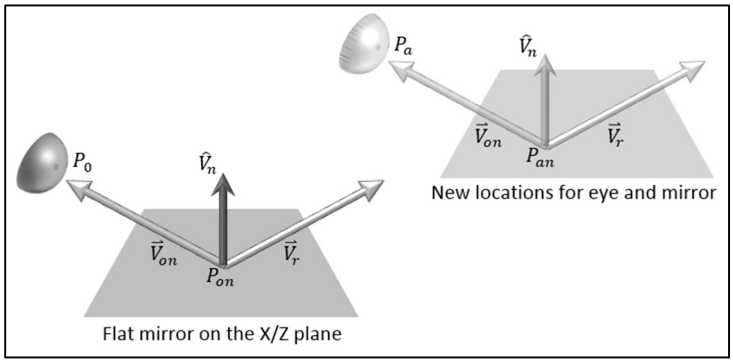

Figure 6-20. *The mirrored reflection direction*

On the left of Figure 6-20, it depicts your eye at an initial position, P_0, looking at a point, P_{on}, on a flat mirror on your desk. The right of Figure 6-20 shows that you have moved your eye and the mirror such that your eye is now located at, P_a, and you are looking at a new position, P_{an} on the mirror. You know that in both of the mirror locations, for the same incoming viewing direction, $\vec{V}_{on}$, as long as the mirror normal, $\hat{V}_n$, is not changed, the reflection direction will always be the same, $\vec{V}_r$. Notice that, the reflection direction, $\vec{V}_r$, is only dependent on the incoming direction, $\vec{V}_{on}$, and the mirror normal vector, $\hat{V}_n$. Neither the location of the mirror, which corresponds to the D-value of the plane equation, nor the location of your eye, P_0 and P_a, nor the location of where you are looking at, P_{on} or P_{an}, affect the reflection direction, $\vec{V}_r$. Only your viewing angle and the rotation orientation of the mirror will affect the reflection direction, just as the reflection direction equation states.

Relevant mathematical concepts covered include:

- The mirrored reflection direction is a function of a normal vector and the incoming direction
- The mirrored reflection of a position can be found by applying the reflection direction to the impact position

Relevant observations on implementation include:

- In the mirrored reflection implementation, the normal vector must be normalized. Additionally, the vector representing the reflection direction is the same length as the vector representing the incoming direction

EXERCISES

Verify the Reflection Direction

Edit MyScript *to replace the implemented solution by first computing the reflection direction, $\vec{V}_r$,*

$$\vec{V}_r = 2(\vec{V}_{on} \cdot \hat{V}_n)\hat{V}_n - \vec{V}_{on}$$

And then compute,

$$P_r = P_{on} + \vec{V}_r$$

Verify your results are identical to the existing implementation. How would you modify your solution if $\vec{V}_{on}$ is a normalized vector?

Compare with the Vector3.Reflect() Function

Please refer to https://docs.unity3d.com/ScriptReference/Vector3.Reflect.html, the Unity Vector3 class also supports the reflection function. Edit MyScript to replace the implementation with the Vector3.Reflect() function and verify the results are identical.

Working with the 'in Front of' Test

Modify MyScript to reflect the line only when P0 is in front of the 2D plane and P1 is behind the 2D plane.

Support 2D Bound Test

Modify MyScript to remove Vn and D and include three user control positions for defining the plane and a 2D bound where reflection only occurs for intersections that are within the bound.

Summary

This chapter summarizes the discussions on vectors and vector algebra by introducing the vector cross product. You have seen that while the results of the vector dot product relate two vectors via a simple floating-point number, the results of the vector cross product provide information on the space that contains the operand vectors in the form of a new vector in a new direction. This new vector is perpendicular to both operand vectors and has a magnitude that is the product of the sizes of the two vectors and the sine of their subtended angle. You have also learned that the cross product of a vector with itself or with a zero vector is the zero vector. In typical video game related problems, it is rare to encounter solutions that depend on the result of the cross product of a vector with itself.

You have also learned that an axis frame, or three perpendicular vectors, can be derived from the result of the cross product. This is accomplished by performing one more cross product between the initial cross product result vector and one of the original operand vectors. This newly derived axis frame can serve as a convenient reference for more advanced applications.

Although such applications were not examined; you did experience working with derived axis frames in 2D space to compute position inside-outside tests for 2D bounds. However, remember that it is important to follow the chosen coordinate space convention, left- or right-handed, when computing an axis frame.

You have built on the results of the cross product to gain insights into 2D planes and to relate the algebraic plane equation, $Ax + By + Cz = D$, to the vector plane equation, $P \cdot \hat{V}_n = D$. You have also examined the geometric implications of the vector plane equation where the vector, $\hat{V}_n$, is the plane normal and is perpendicular to the 2D plane, and D is the distance between the origin of the Cartesian Coordinate System and the 2D plane measured along the plane normal, $\hat{V}_n$, direction.

These insights into 2D planes allowed the derivation of three important solutions with wide applications in video games and computer graphics applications: projection of a position, intersection with a line segment, and reflection direction. You have interacted with and examined the implementation of these solutions as well as verified that these solutions are general and can work with any input conditions. Lastly, you have observed that the typical implementation of vector solutions match closely with the vector algebraic solution, are elegant, and typically involve a small number of lines of code.