CHAPTER 6

Defining Behaviors and Detecting Collisions

After completing this chapter, you will be able to:

- Implement autonomous, controlled, gradual turning and target-locked chasing behaviors
- Collide textured objects accurately
- Understand the efficiency concerns of pixel-accurate collision
- Program with pixel-accurate collision effectively and efficiently

Introduction

By this point, your game engine is capable of implementing games in convenient coordinate systems and presenting and animating objects that are visually appealing. However, there is a lack of abstraction support for the behaviors of objects. You can see the direct result of this shortcoming in the `initialize()` and `update()` functions of the `MyGame` objects in all the previous projects: the `initialize()` function is often crowded with mundane per-game object settings, while the `update()` function is often crowded with conditional statements for controlling objects, such as checking for key presses for moving the hero.

A well-designed system should hide the initialization and controls of individual objects with proper object-oriented abstractions, or classes. An abstract `GameObject` class should be introduced to abstract and hide the specifics of initialization and behaviors. There are two main advantages to this approach. First, the `initialize()` and `update()` functions of a game level can focus on managing individual game object and the interactions of these objects without being clustered with details specific to different types of objects. Second, as you have experienced with the `Renderable` and `Shader` object hierarchies, proper object-oriented abstraction creates a standardized interface and facilitates code sharing and reuse.

As you transition from working with the mere drawing of objects (in other words, `Renderable`) to programming with behavior of objects (in other words, `GameObject`), you will immediately notice that for the game to be entertaining or fun, the objects need to interact. Interesting behaviors of objects, such as facing or evading enemies, often require the knowledge of the relative positions of other relevant objects in the game. In general, resolving relative positions of all objects in a two-dimensional world is nontrivial. Fortunately, typical video games require the knowledge only of objects that are in close proximity to each other or of detecting objects that are about to collide or have collided. An efficient but somewhat crude approximation to detect collision is to compute, bound, and collide objects with bounding boxes. In the simplest cases, bounding boxes are rectangular boxes with edges that are aligned with the x/y-axes. Because of the axes alignments, it is computationally efficient to detect when two bounding boxes overlap or when

collision is about to occur. Most 2D game engines detect the actual collision between two textured objects by comparing the location of pixels from both objects and detecting the situation when at least one of the pixels overlaps. This computationally intensive process is known as per-pixel-accurate collision detection, pixel-accurate collision, or per-pixel collision.

This chapter begins by introducing the `GameObject` class to provide a platform for abstracting game object behaviors. The `GameObject` class is then generalized to introduce common behavior attributes including speed, movement direction, and target-locked chasing. The rest of the chapter focuses on deriving an efficient per-pixel accurate collision implementation that supports both textured and animated sprite objects.

Game Objects

As mentioned, an abstraction that encapsulates the intrinsic behavior of typical game objects should be introduced to minimize the clustering of code in the `initialize()` and `update()` functions of a game level and to facilitate reuse. This section introduces the simple `GameObject` class to illustrate how the cleaner and unclustered `initialize()` and `update()` functions clearly reflect the in-game logic and to demonstrate how the basic platform for abstracting object behaviors facilitates design and code reuse.

The Game Objects Project

This project defines the simple `GameObject` class as the first step in building an abstraction to represent actual objects with behaviors in a game. You can see an example of this project running in Figure 6-1. Notice the many minions charging from right to left and wrapping around when they reach the left boundary. This project leads you to create the infrastructure to support the many minions while keeping the logic in the `MyGame` level simple. The source code to this project is defined in the `Chapter6/6.1.GameObjects` folder.

Figure 6-1. *Running the Game Objects project*

The controls of the project are as follows:

- *WASD keys*: Move the hero up, left, down, and right

The goals of the project are as follows:

- To improve the engine drawing infrastructure to receive the `Camera` object instead of just the View-Projection matrix of a camera. Although this goal is minimally related to game object behaviors, this is a necessary step to include in order to support the growing game engine sophistication.
- To begin defining the `GameObject` class to encapsulate object behaviors in games.
- To demonstrate the creation of subclasses to the `GameObject` class to maintain the simplicity of the `MyGame` level `update()` function.

You can find the following external resource file in the `assets` folder: `minion_sprite.png`; you'll also find the `fonts` folder that contains the default system fonts. Note that, as shown in Figure 6-2, the `minion_sprite.png` image file has been updated from the previous project to include two extra sprite elements: the `DyePack` and the `Brain` minion.

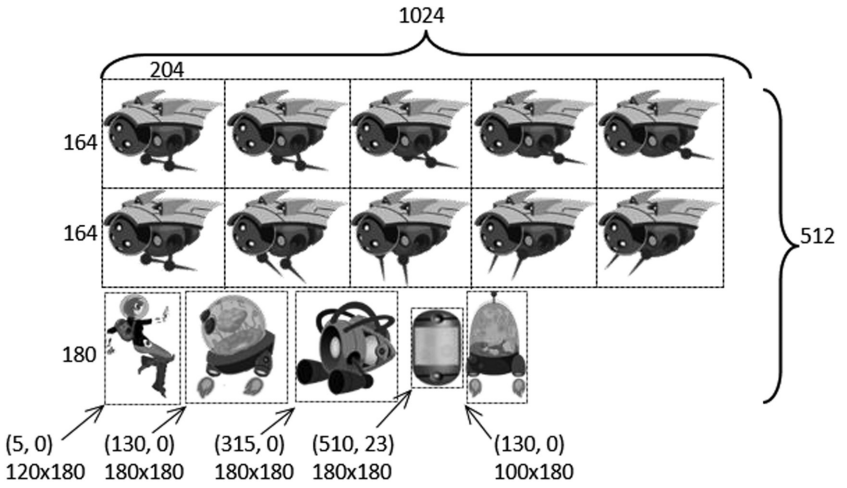

Figure 6-2. *The new sprite elements of the minion_sprite.png image*

Modify Renderable and Shader Objects to Draw with a Camera

As the game engine improves to a higher level of abstraction, it is important to upgrade its drawing capability from receiving and working with a simple View-Projection transformation to a `Camera` object. This change prepares the engine to support the more sophisticated shaders to be covered in later chapters. This change involves all the `Shader` and `Renderable` objects. Though many files are involved, the actual changes are mundane; they consist of a simple replacement of the matrix with a camera. For this reason, only changes to the base objects, `Renderable` and `SimpleShader`, are shown here. You can refer to the source code files of `TextureShader`, `SpriteShader`, `TextureRenderable`, `SpriteRenderable`, `SpriteAnimateRenderable`, and `FontRenderable` for details.

1. Modify the `Renderable` object's `draw()` function to replace the `vpMatrix` parameter with `aCamera` and forward this camera to the shader.

```
Renderable.prototype.draw = function (aCamera) {
    var gl = gEngine.Core.getGL();
    this.mShader.activateShader(this.mColor, aCamera);
    this.mShader.loadObjectTransform(this.mXform.getXform());
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
};
```

2. Modify the `SimpleShader` object's `activateShader()` function to work with a camera.

```
SimpleShader.prototype.activateShader = function (pixelColor, aCamera) {
    var gl = gEngine.Core.getGL();
    gl.useProgram(this.mCompiledShader);
    gl.uniformMatrix4fv(this.mViewProjTransform, false, aCamera.getVPMatrix());
    gl.enableVertexAttribArray(this.mShaderVertexPositionAttribute);
    gl.uniform4fv(this.mPixelColor, pixelColor);
};
```

Notice that these changes are trivial replacements of the parameter. Please refer to the source code files for all similar changes to all the objects derived from the `Renderable` and `SimpleShader` objects.

Define the GameObject

The goal is to define a logical abstraction to integrate the behavioral characteristics of a game object including the ability to control positions, drawing, and so on. As in the case for the `Scene` objects in the earlier chapter, the main result is to provide a well-defined interface governing the functions that subclasses implement. The more sophisticated behaviors will be introduced in the next section. This example demonstrates the potentials of only the `GameObject` class with minimal behaviors defined.

1. Add a new folder `src/Engine/GameObjects` for storing `GameObject`-related files.
2. Create a new file in this folder, name it `GameObject.js`, and add the following code:

```
function GameObject(renderableObj) {
    this.mRenderComponent = renderableObj;
}

GameObject.prototype.getXform = function () {
    return this.mRenderComponent.getXform();
};

GameObject.prototype.update = function () {};

GameObject.prototype.getRenderable = function () {
    return this.mRenderComponent;
};

GameObject.prototype.draw = function (aCamera) {
    this.mRenderComponent.draw(aCamera);
};
```

With the assessors to the `Renderable` and `Transfrom` objects defined, all `GameObject` instances can be drawn and have defined locations and sizes. Note that the `update()` function is designed for subclasses to override with per-object specific behaviors, and thus it is left empty.

Manage Game Objects in Sets

Because most games consist of many interacting objects, it is useful to define a utility class to support working with a set of `GameObject` instances.

1. Create a new file in the `src/Engine/GameObjects` folder and name it `GameObjectSet.js`. Define the constructor to create an array for holding `GameObject` instances.

```
function GameObjectSet() {
    this.mSet = [];
}
```

2. Define functions for managing the set membership.

```
GameObjectSet.prototype.size = function() { return this.mSet.length; };

GameObjectSet.prototype.getObjectAt = function(index) {
    return this.mSet[index];
};

GameObjectSet.prototype.addToSet = function(obj) {
    this.mSet.push(obj);
};
```

3. Define functions to update and draw each of the `GameObject` instances in the set.

```
GameObjectSet.prototype.update = function() {
    var i;
    for (i = 0; i < this.mSet.length; i++) {
        this.mSet[i].update();
    }
};

GameObjectSet.prototype.draw = function(aCamera) {
    var i;
    for (i = 0; i < this.mSet.length; i++) {
        this.mSet[i].draw(aCamera);
    }
};
```

Test the GameObject and GameObjectSet

The goals are to ensure proper functioning of the new `GameObject` class, to demonstrate customization of behaviors by individual object types, and to observe a cleaner `MyGame` implementation clearly reflecting the in-game logic. To accomplish these goals, three object types are defined: `DyePack`, `Hero`, and `Minion`. Before you begin to examine the detail implementation of these objects, following good source code organization practice, create the new folder `src/MyGame/Objects` for storing the new object types.

The DyePack GameObject

The `DyePack` object derives from the `GameObject` class to demonstrate the simplest example of a `GameObject` where the object can be drawn only without any specific behaviors.

- Create a new file in the `src/MyGame/Objects/` folder and name it `DyePack.js`. Define this as a subclass of `GameObject` and implement the constructor as follows:

```
function DyePack(spriteTexture) {
    this.kRefWidth = 80;
    this.kRefHeight = 130;

    this.mDyePack = new SpriteRenderable(spriteTexture);
    this.mDyePack.setColor([1, 1, 1, 0.1]);
    this.mDyePack.getXform().setPosition(50, 33);
```

```
    this.mDyePack.getXform().setSize(this.kRefWidth / 50, this.kRefHeight / 50);
    this.mDyePack.setElementPixelPositions(510, 595, 23, 153);
    GameObject.call(this, this.mDyePack);
}
gEngine.Core.inheritPrototype(DyePack, GameObject);
```

Notice that even without specific behaviors, the `DyePack` is implementing code that used to be found in the `initialize()` function of the `MyGame` level. In this way, the `DyePack` object hides specific geometric information and simplifies the `MyGame` level.

The Hero GameObject

The `Hero` object supports direct user keyboard control. This object demonstrates hiding of game object control logic from the `MyGame` level's `update()` function.

1. Create a new file in the `src/MyGame/Objects/` folder and name it `Hero.js`. Define this as a subclass of `GameObject` and implement the constructor to initialize the sprite UV values, size, and position as follows:

```
function Hero(spriteTexture) {
    this.kDelta = 0.3;
    this.mDye = new SpriteRenderable(spriteTexture);
    this.mDye.setColor([1, 1, 1, 0]);
    this.mDye.getXform().setPosition(35, 50);
    this.mDye.getXform().setSize(9, 12);
    this.mDye.setElementPixelPositions(0, 120, 0, 180);
    GameObject.call(this, this.mDye);
}
gEngine.Core.inheritPrototype(Hero, GameObject);
```

2. Add a function to support the update of this object by user keyboard control:

```
Hero.prototype.update = function () {
    // control by WASD
    var xform = this.getXform();
    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.W))
        xform.incYPosBy(this.kDelta);
    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.S))
        xform.incYPosBy(-this.kDelta);
    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.A))
        xform.incXPosBy(-this.kDelta);
    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.D))
        xform.incXPosBy(this.kDelta);
};
```

The `Hero` object moves at a `kDelta` rate based on WASD input from the keyboard.

The Minion GameObject

The Minion object demonstrates that simple autonomous behavior can also be hidden.

1. Create a new file in the src/MyGame/Objects/ folder and name it Minion.js. Define this as a subclass of GameObject and implement the constructor to initialize the sprite UV values, sprite animation parameters, size, and position as follows:

```
function Minion(spriteTexture, atY) {
    this.kDelta = 0.2;
    this.mMinion = new SpriteAnimateRenderable(spriteTexture);
    this.mMinion.setColor([1, 1, 1, 0]);
    this.mMinion.getXform().setPosition(Math.random() * 100, atY);
    this.mMinion.getXform().setSize(12, 9.6);
    this.mMinion.setSpriteSequence(512, 0,  // first element pixel
                                   204, 164,     // widthxheight in pixels
                                   5,            // number of elements
                                   0);           // horizontal padding in between
    this.mMinion.setAnimationType(SpriteAnimateRenderable.eAnimationType.eAnimateSwing);
    this.mMinion.setAnimationSpeed(15);     // show each element for mAnimSpeed updates
    GameObject.call(this, this.mMinion);
}
gEngine.Core.inheritPrototype(Minion, GameObject);
```

2. Add a function to update the sprite animation, support the simple right-to-left movements, and provide the wrapping functionality.

```
Minion.prototype.update = function () {
    // remember to update this.mMinion's animation
    this.mMinion.updateAnimation();

    // move toward the left and wraps
    var xform = this.getXform();
    xform.incXPosBy(-this.kDelta);

    // if fly off to the left, re-appear at the right
    if (xform.getXPos() < 0) {
        xform.setXPos(100);
        xform.setYPos(65 * Math.random());
    }
};
```

MyGame Scene

As in all cases, the MyGame level is implemented in the MyGame.js file. In this case, with the three specific GameObject subclasses defined, follow these steps:

1. The constructor and the loadScene(), unloadScene(), and draw() functions are similar as in previous projects, so the details are not shown here. Please refer to the MyGame.js file for the actual implementations.

2. Edit the `initialize()` function and add the following code:

```
MyGame.prototype.initialize = function () {
    // ... identical code to previous project ...

    // Step B: The dye pack: simply another GameObject
    this.mDyePack = new DyePack(this.kMinionSprite);

    // Step C: A set of Minions
    this.mMinionset = new GameObjectSet();
    var i = 0, randomY, aMinion;
    // create 5 minions at random Y values
    for (i = 0; i <  5; i++) {
        randomY = Math.random() * 65;
        aMinion = new Minion(this.kMinionSprite, randomY);
        this.mMinionset.addToSet(aMinion);
    }

    // Step D: Create the hero object
    this.mHero = new Hero(this.kMinionSprite);

    // Step E: Create and initialize message output
    this.mMsg = new FontRenderable("Status Message");
    this.mMsg.setColor([0, 0, 0, 1]);
    this.mMsg.getXform().setPosition(1, 2);
    this.mMsg.setTextHeight(3);
};
```

The details of step A, the creation of the camera and initialization of the background color, are not shown because they are identical to previous projects. Steps B, C, and D show the instantiation of the three object types, with step C showing the creation and insertion of the right-to-left moving `Minion` objects into the `mMinionset`. Notice that the initialization function is free from the clustering of setting each object's textures, geometries, and so on.

3. Edit the `update()` function to update the game state.

```
MyGame.prototype.update = function () {
    this.mHero.update();
    this.mMinionset.update();
    this.mDyePack.update();
};
```

With the well-defined behaviors for each object type abstracted, the clean `update()` function clearly shows that the game consists of three noninteracting objects.

You can now run the project and notice that the slightly more complex movements of six minions are accomplished with much cleaner `initialize()` and `update()` functions. The `initialize()` function simply consists of only logic and controls of placing created objects in the game world and does not include any specific settings for different object types. With the `Minion` object defining its motion behaviors in its own `update()` function, the logic in the `MyGame` level's `update()` function can focus on object-level controls without being clustered by the details. Note that the structure of this function clearly shows the three objects are updated independently and do not interact with each other.

Chasing of a GameObject

A closer examination of the previous project reveals that though there are quite a few minions moving on the screen, their motions are simple and boring. Even though there are variations in speed and direction, the motions are without purpose or awareness of other game objects in the scene. To support more sophisticated or interesting movements, a `GameObject` needs to be aware of the locations of other objects and determine motion based on that information.

Chasing behavior is one such example. The goal of a chasing object is usually to catch the game object that it is targeting. This requires programmatic manipulation of the chaser's front direction and speed so it can hone in on its target. However, it is generally important to avoid implementing a chaser that has perfect aim and always hits its target—because if the projectile always hits its target and characters are unable to avoid being hit, the challenge will essentially be removed from the game. Nonetheless, this does not mean you should not implement a perfect chaser if your game design requires it. You will implement a chaser in the next project.

Vectors and the associated operations are the foundation for implementing object movements and behaviors. Before programming with vectors, a quick review is provided. As in the case of matrices and transform operators, the following discussion is not meant to be a comprehensive coverage of vectors. Instead, the focus is on the application of a small collection of concepts that are relevant to the implementation of the game engine. This is not a study of the theories behind the mathematics. If you are interested in the specifics of vectors and how they relate to computer graphics, please refer to the discussion in Chapter 1 where you can learn more about these topics in depth by delving into relevant books on linear algebra and computer graphics.

Vectors Review

Vectors are used across many fields of study, including mathematics, physics, computer science, and engineering. They are particularly important in games; nearly every game uses vectors in one way or another. Because they are used so extensively, this section is devoted to understanding and utilizing vectors in games.

One of the most common uses for vectors is to represent an object's displacement and direction, or *velocity*. This can be done easily because a vector is defined by its size and direction. Using only this small amount of information, you can represent attributes such as an object's velocity or acceleration. If you have the position of an object, its direction, and its velocity, then you have sufficient information to move it around the game world without user input.

Before going any further, it is important to review the concepts of a vector, starting with how you can define one. A vector can be specified using two points. For example, given the arbitrary positions $P_a = (x_a, y_a)$ and $P_b = (x_b, y_b)$, you can define the vector from P_a to P_b or ($\bar{V}_{ab}$) as $P_b - P_a$. You can see this represented in the following equations and Figure 6-3:

- $P_a = (x_a, y_a)$
- $P_b = (x_b, y_b)$
- $\bar{V}_{ab} = P_b - P_a = (x_b - x_a, y_b - y_a)$

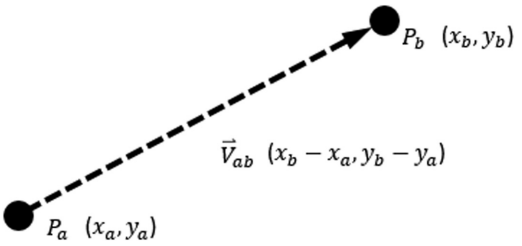

Figure 6-3. *A vector defined by two points*

Now that you have a vector $\vec{V}_{ab}$, you can easily ascertain its length (or size) and its direction. A vector's length is equal to the distance between the two points that created it. In this example, the length of $\vec{V}_{ab}$ is equal to the distance between P_a and P_b, while the direction of $\vec{V}_{ab}$ goes from P_a toward P_b.

■ **Note** The size of a vector is often referred to as its length, or *magnitude*.

In the `gl-matrix` library, the `vec2` object implements the functionality of a two-dimensional (2D) vector. Conveniently, you can also use the `vec2` object to represent 2D points, or positions in space. In the preceding example, P_a, P_b, and $\vec{V}_{ab}$ can all be implemented as instances of the `vec2` object; however, $\vec{V}_{ab}$ is the only mathematically defined vector. P_a and P_b represent positions, or points used to create a vector.

Recall that a vector can also be normalized. A *normalized* vector (also known as a *unit vector*) always has a size of 1. You can see a normalized vector by the following function and Figure 6-4. Notice that the mathematic symbol for a regular vector is $\vec{V}_a$ and for a normalized vector is $\hat{V}_a$:

- $vec2.normalized(\vec{V}_a)$: Normalizes vector $\vec{V}_a$ and stores the results to the `vec2` object

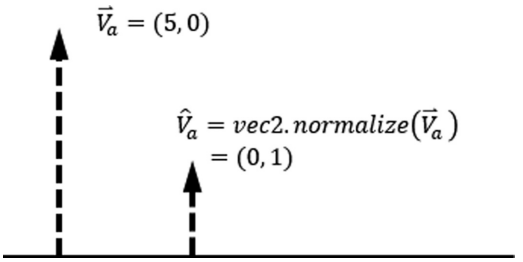

Figure 6-4. *A vector being normalized*

Vectors to a position can also be rotated. If, for example, the vector $\vec{V} = (x_v, y_v)$ represents the direction from the origin to the position (x_v, y_v) and you want to rotate it by θ, then you can use the following equations shown to derive x_r and y_r. Figure 6-5 shows the rotation of a vector to a position from the origin being applied.

- $x_r = x_v \cos\theta - y_v \sin\theta$
- $y_r = x_v \sin\theta + y_v \cos\theta$

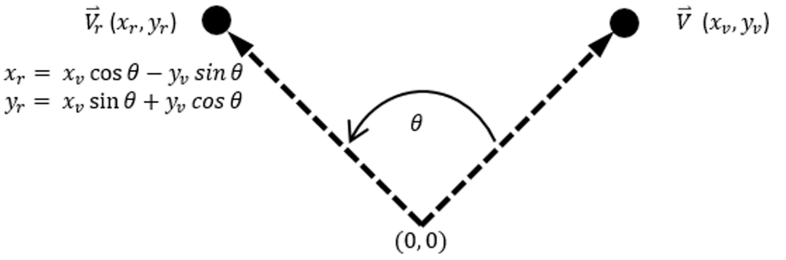

Figure 6-5. *A vector from the origin to a position being rotated by the angle theta*

■ **Note** JavaScript trigonometric functions, including the `Math.sin()` and `Math.cos()` functions, assume input to be in radians and not degrees. Recall that 1 degree is equal to $\frac{\pi}{180}$ radians.

Furthermore, it is important to remember that vectors are defined by their direction and size; in other words, two vectors can be equal to each other independent of the locations of the vectors. Figure 6-6 shows two vectors ($\vec{V}_a$ and $\vec{V}_{bc}$) that are located at different positions but have the same direction and magnitude. Because their direction and magnitude are the same, these vectors are equal to each other. In contrast, the vector $\vec{V}_d$ is not the same because its direction and magnitude are different from the others.

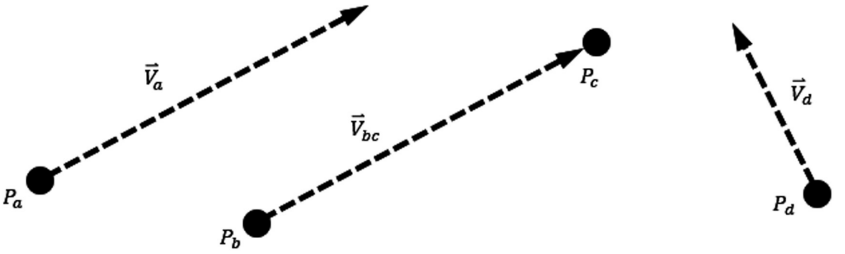

Figure 6-6. *Three vectors represented in 2D space with two vectors equal to each other*

The Dot Product

The dot product of two normalized vectors provides you with the means to find the angle between those vectors. For example, given the following:

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

then the following is true:

- $\vec{V}_1 \cdot \vec{V}_2 = \vec{V}_2 \cdot \vec{V}_1 = x_1 x_2 + y_1 y_2$.

Additionally, if both vectors $\vec{V}_1$ and $\vec{V}_2$ are normalized, then:

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

You can see this concept reinforced in Figure 6-7. Moreover, it is also important to recognize that if $\vec{V}_1 \cdot \vec{V}_2 = 0$, then the two vectors are perpendicular.

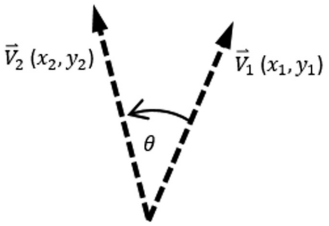

Figure 6-7. *The angle between two vectors, which can be found through the dot product*

■ **Note** If you need to review or refresh the concept of a dot product, please refer to `http://www.//mathsisfun.com/algebra/vectors-dot-product.html`.

The Cross Product

The cross product of two vectors produces a vector that is *orthogonal*, or perpendicular, to both of the original vectors. In 2D games, where the 2D dimensions lie flat on the screen, the cross product results in a vector that points either inward (toward the screen) or outward (away from the screen). This may seem odd because it is not intuitive that crossing two vectors in 2D or the XY plane results in a vector that lies in the third dimension, or along the z-axis. However, this vector is essential for the sole purpose of determining whether the game object needs to rotate in the clockwise or counterclockwise direction. Take a closer look at the following:

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

Given the previous, the following is true:

- $\vec{V}_3 = \vec{V}_1 \times \vec{V}_2$ is a vector perpendicular to both $\vec{V}_1$ and $\vec{V}_2$.

Also, you know that the cross product of two vectors on the XY plane results in a vector in the Z direction. When $\vec{V}_1 \times \vec{V}_2 > 0$, you know that $\vec{V}_1$ is in the clockwise direction from $\vec{V}_2$; similarly, when $\vec{V}_1 \times \vec{V}_2 < 0$, you know that $\vec{V}_1$ is in the counterclockwise direction. Figure 6-8 should help clarify this concept.

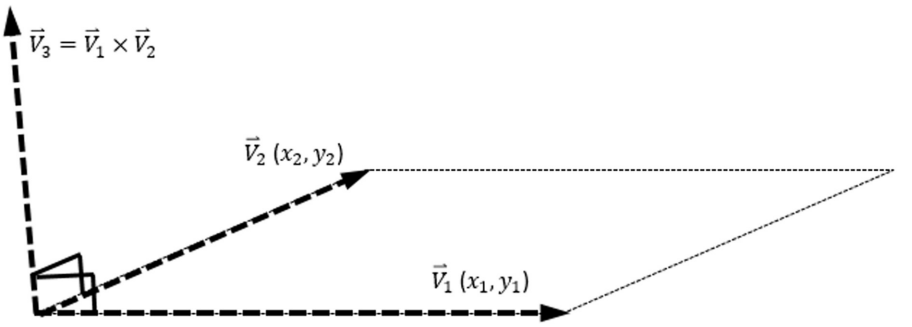

Figure 6-8. *The cross product on two vectors*

■ **Note** If you need to review or refresh the concept of a cross product, please refer to `http://www.mathsisfun.com/algebra/vectors-cross-product.html`.

The Front and Chase Project

This project implements more interesting and sophisticated behaviors based on the vector concepts reviewed. Instead of constant and aimless motions, you will experience defining and varying the front direction of an object and guiding an object to chase after another object in the scene. You can see an example of this project running in Figure 6-9. The source code to this project is defined in the `Chapter6/6.2.FrontAndChase` folder.

Figure 6-9. *Running the Front and Chase project*

The controls of the project are as follows:

- *WASD keys*: Moves the `Hero` object
- *Left/Right Arrow keys*: Change the front direction of the `Brain` object when it is under user control
- *Up/Down Arrow keys*: Increase/decrease the speed of the `Brain` object
- *H key*: Switches the `Brain` object to be under user arrow keys control
- *J key*: Switches the `Brain` object to always point at and move toward the current `Hero` object position
- *K key*: Switches the `Brain` object to turn and move gradually toward the current `Hero` object position

The goals of the project are as follows:

- To experience working with speed and direction
- To practice traveling along a predefined direction

- To implement algorithms with vector dot and cross products
- To examine and implement chasing behavior

You can find the same external resource files as in the previous project in the `assets` folder.

Add Vector Rotation to the gl-matrix Library

The `gl-matrix` library does not support rotating a position in 2D space. This can be rectified by adding the following code to the `gl-matrix.js` file in the `lib` folder:

```
vec2.rotate = function(out, a, c){
    var r=[];
    // perform rotation
    r[0] = a[0]*Math.cos(c) - a[1]*Math.sin(c);
    r[1] = a[0]*Math.sin(c) + a[1]*Math.cos(c);
    out[0] = r[0];
    out[1] = r[1];
    return r;
};
```

■ **Note** This modification to the `gl-matrix` library must be present in all the projects from this point forward.

Modify GameObject to Support Interesting Behaviors

The `GameObject` class abstracts and implements the desired new object behaviors:

1. Edit the `GameObject.js` file and modify the `GameObject` constructor to define visibility, front direction, and speed.

```
function GameObject(renderableObj) {
    this.mRenderComponent = renderableObj;
    this.mVisible = true;
    this.mCurrentFrontDir = vec2.fromValues(0, 1);  // current front direction
    this.mSpeed = 0;
}
```

2. Add assessor and setter functions for the instance variables.

```
GameObject.prototype.getXform = function () { return this.mRenderComponent.getXform(); };
GameObject.prototype.setVisibility = function (f) { this.mVisible = f; };
GameObject.prototype.isVisible = function () { return this.mVisible; };

GameObject.prototype.setSpeed = function (s) { this.mSpeed = s; };
GameObject.prototype.getSpeed = function () { return this.mSpeed; };
GameObject.prototype.incSpeedBy = function (delta) { this.mSpeed += delta; };
```

```
GameObject.prototype.setCurrentFrontDir = function (f) { vec2.normalize(this.
    mCurrentFrontDir, f); };
GameObject.prototype.getCurrentFrontDir = function () { return this.mCurrentFrontDir; };

GameObject.prototype.getRenderable = function () { return this.mRenderComponent; };
```

3. Implement a function to rotate the front direction toward a position, p.

```
GameObject.prototype.rotateObjPointTo = function (p, rate) {
    // Step A: determine if reach the destination position p
    var dir = [];
    vec2.sub(dir, p, this.getXform().getPosition());
    var len = vec2.length(dir);
    if (len < Number.MIN_VALUE)
        return; // we are there.
    vec2.scale(dir, dir, 1 / len);

    // Step B: compute the angle to rotate
    var fdir = this.getCurrentFrontDir();
    var cosTheta = vec2.dot(dir, fdir);
    if (cosTheta > 0.999999)  // almost exactly the same direction
        return;

    // Step C: clamp the cosTheda to -1 to 1
    // in a perfect world, this would never happen! BUT ...
    if (cosTheta > 1)
        cosTheta = 1;
    else
        if (cosTheta < -1)
            cosTheta = -1;

    // Step D: compute whether to rotate clockwise, or counterclockwise
    var dir3d = vec3.fromValues(dir[0], dir[1], 0);
    var f3d = vec3.fromValues(fdir[0], fdir[1], 0);
    var r3d = [];
    vec3.cross(r3d, f3d, dir3d);

    var rad = Math.acos(cosTheta);  // radian to roate
    if (r3d[2] < 0)
        rad = -rad;

    // Step E: rotate the facing direction with the angle and rate
    rad *= rate;  // actual angle need to rotate from Obj's front
    vec2.rotate(this.getCurrentFrontDir(), this.getCurrentFrontDir(), rad);
    this.getXform().incRotationByRad(rad);
};
```

The `rotateObjPointTo()` function rotates the `mCurrentFrontDir` to point to destination position `p` at the `rate` specified by the parameter. Here are the detailed operations:

a. Step A computes the distance between the current object and the destination position `p`. If this value is small, it means current object and the target position are close. The function returns without further processing.

b. Step B computes the dot product to determine the angle between the current front direction of the object (`fdir`) and the direction toward the destination position `p` (`dir`). If these two vectors are pointing in the same direction, the function returns. This computation is illustrated in Figure 6-10.

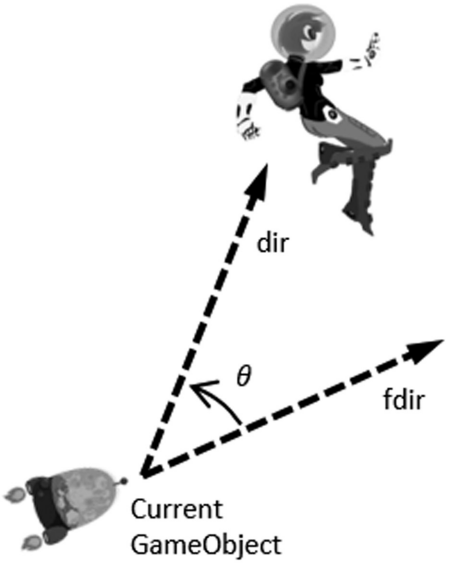

Figure 6-10. *A GameObject (Brain) chasing a target (Hero)*

c. Step C checks for the range of `cosTheta`. This is a step that must be performed because of the inaccuracy of floating-point operations in JavaScript.

d. Step D computes the results of the cross product to determine whether the current `GameObject` should be turning clockwise or counterclockwise to face toward the destination position `p`.

e. Step E rotates `mCurrentFrontDir` and sets the rotation in the `Transform` of the `Renderable` object.

4. Add a function to update the object's position with its direction and speed.

```
GameObject.prototype.update = function () {
    // simple default behavior
    var pos = this.getXform().getPosition();
    vec2.scaleAndAdd(pos, pos, this.getCurrentFrontDir(), this.getSpeed());
};
```

Notice that if the `mCurrentFrontDir` is modified by the `rotateObjPointTo()` function, then this `update()` function will move the object toward the target position `p`, and the object will behave as though it is chasing the target.

5. Add a function to draw the object based on the visibility setting.

```
GameObject.prototype.draw = function (aCamera) {
    if (this.isVisible())
        this.mRenderComponent.draw(aCamera);
};
```

Test the Chasing Functionality

The strategy and goals of this test case are to create a steerable `Brain` object for testing traveling along a predefined front direction and direct the `Brain` object to chase after the `Hero` object to test the chasing functionality.

Define the Brain GameObject

The `Brain` object will travel along its front direction under the control of the user's Left/Right Arrow keys for steering:

1. Create a new file in the `src/MyGame/Objects` folder and name it `Brain.js`. Define this as a subclass of `GameObject` and implement the constructor to initialize the appearance and initial behavior of the object.

```
function Brain(spriteTexture) {
    this.kDeltaDegree = 1;
    this.kDeltaRad = Math.PI * this.kDeltaDegree / 180;
    this.kDeltaSpeed = 0.01;
    this.mBrain = new SpriteRenderable(spriteTexture);
    this.mBrain.setColor([1, 1, 1, 0]);
    this.mBrain.getXform().setPosition(50, 10);
    this.mBrain.getXform().setSize(3, 5.4);
    this.mBrain.setElementPixelPositions(600, 700, 0, 180);
    GameObject.call(this, this.mBrain);
    this.setSpeed(0.05);
}
gEngine.Core.inheritPrototype(Brain, GameObject);
```

2. Override the `update()` function to support user steering and controlling the speed. Notice that the default `update()` function in the `GameObject` must be called to support the basic traveling of the object along the front direction according to its speed.

```
Brain.prototype.update = function () {
    GameObject.prototype.update.call(this);  // default moving forward
    var xf = this.getXform();
    var fdir = this.getCurrentFrontDir();
    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Left)) {
        xf.incRotationByDegree(this.kDeltaDegree);
        vec2.rotate(fdir, fdir, this.kDeltaRad);
    }
```

```
    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Right)) {
        xf.incRotationByRad(-this.kDeltaRad);
        vec2.rotate(fdir, fdir, -this.kDeltaRad);
    }
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.Up))
        this.incSpeedBy(this.kDeltaSpeed);
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.Down))
        this.incSpeedBy(-this.kDeltaSpeed);
};
```

The MyGame Scene

Modify the `MyGame` scene to test the `Brain` object movement. In this case, except for the `update()` function, the rest of the source code in `MyGame.js` is similar to previous projects. For this reason, only the details of the `update()` function are shown. Please refer to the source code in the `MyGame.js` file for the rest of the implementation details.

```
MyGame.prototype.update = function () {
    var msg = "Brain modes [H:keys, J:immediate, K:gradual]: ";
    var rate = 1;

    this.mHero.update();

    switch (this.mMode) {
    case 'H':
        this.mBrain.update();  // player steers with arrow keys
        break;
    case 'K':
        rate = 0.02;    // graduate rate
        // When "K" is typed, the following should also be executed.
    case 'J':
        this.mBrain.rotateObjPointTo(this.mHero.getXform().getPosition(), rate);
        GameObject.prototype.update.call(this.mBrain);
        break;
    }

    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.H))
        this.mMode = 'H';
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.J))
        this.mMode = 'J';=
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.K))
        this.mMode = 'K';

    this.mMsg.setText(msg + this.mMode);
};
```

In the `update()` function, the `switch` statement uses `mMode` to determine how to update the `Brain` object. In the cases of `J` and `K` modes, the `Brain` object turns toward the `Hero` object position with the `rotateObjPointTo()` function call. While in the `H` mode, the `Brain` object `update()` function is called for the user to steer the object with the arrow keys. The final three `if` statements simply set the `mMode` variable according to user input.

You can now try running the project. Initially the `Brain` object is under the user control. You can use the Left and Right Arrow keys to change the front direction of the `Brain` object to experience with steering of the object. Pressing the J key causes the `Brain` object to immediately point and move toward the `Hero` object. This is a result of the default turn `rate` value of 1.0. The K key causes a more natural behavior, where the `Brain` object continues to move forward and turns gradually to moves toward the `Hero` object. Feel free to change the values of the `rate` variable or modify the controls value of the `Brain` object; for example, change the `kDeltaRad` or `kDeltaSpeed` to experience with different behavior settings.

Collisions Between GameObjects

In the previous project, the `Brain` object would never stop traveling. Notice that under the J and K modes, the `Brain` object would orbit or rapidly flip directions when it reaches the target position. The `Brain` object misses the crucial ability to detect that it has collided with the `Hero` object, and thus it can stop moving. This section describes the axes-aligned bounding boxes, one of the most straightforward tools for approximating object collisions, and demonstrates the implementation of collision detection based on bounding boxes.

Bounding Box

A bounding box is an x/y-axes aligned rectangular box that bounds a given object. The term *x/y-axes aligned* refers to the fact that the four sides of a bounding box are parallel either to the horizontal x-axis or to the vertical y-axis. Figure 6-11 shows an example of representing the bounds to the `Brain` object by the lower-left corner (`mLL`), width, and height. This is a fairly common way to represent a bounding box because it uses only one position and two floating-point numbers to represent the dimensions.

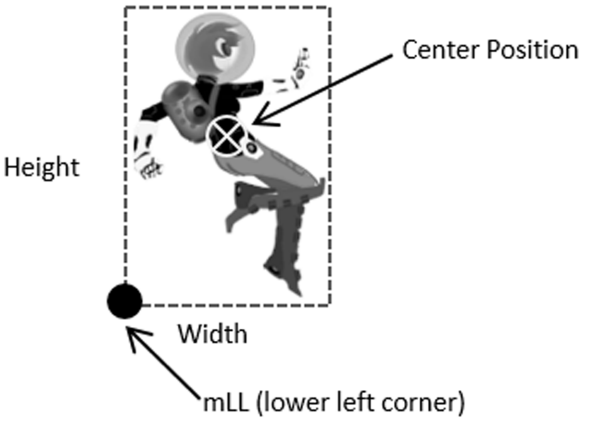

Figure 6-11. *The lower-left corner and size of the bounds for an object*

It is interesting to note that, in addition to representing the bounds of an object, bounding boxes can be used to represent the bounds of any given rectangular area. For example, recall that the WC visible through the `Camera` is a rectangular area with the camera's position located at the center and the WC width/height defined. A bounding box can be defined to represent the visible WC rectangular area, or the WC window, and used for detecting collision between the WC window and `GameObject` instances in the game world.

The Bounding Box and Collisions Project

This project demonstrates how to define a bounding box for a `GameObject` instance and detect collisions between two `GameObject` instances based on their bounding boxes. It is important to remember that bounding boxes are axes aligned, and thus the solution presented in this section does not support collision detections between rotated objects. You can see an example of this project running in Figure 6-12. The source code to this project is defined in the `Chapter6/6.3.BBoxAndCollisions` folder.

Figure 6-12. *Running the Bounding Box and Collisions project*

The controls of the project are identical to the previous project.

- *WASD keys*: Moves the `Hero` object
- *Left/Right Arrow keys*: Change the front direction of the `Brain` object when it is under user control
- *Up/Down Arrow keys***:** Increase/decrease the speed of the `Brain` object
- *H key*: Switches the `Brain` object to be under user arrow keys control

- *J key*: Switches the `Brain` object to always point at and move toward the current `Hero` object position
- *K key*: Switches the `Brain` object to turn and move gradually toward the current `Hero` object position

The goals of the project are as follows:

- To understand the implementation of a bounding box class
- To experience working with the bounding box of a `GameObject` instance
- To compute and work with the bounds of a `Camera` WC window
- To program with object collisions and object and camera WC window collisions

You can find the same external resource files as in the previous project in the `assets` folder.

Define a Bounding Box Object

Define a `BoundingBox` object to represent the bounds of a rectangular area.

1. Create a new file in the `src/Engine` folder, name it `BoundingBox.js`, and define its constructor with instance variables to represent a bound, as illustrated in Figure 6-11.

```
function BoundingBox(centerPos, w, h) {
    this.mLL = vec2.fromValues(0, 0);  // the lower-left corner position
    this.setBounds(centerPos, w, h);
}
```

2. The `setBounds()` function computes and sets the instance variables of the bounding box.

```
BoundingBox.prototype.setBounds = function (centerPos, w, h) {
    this.mWidth = w;
    this.mHeight = h;
    this.mLL[0] = centerPos[0] - (w / 2);
    this.mLL[1] = centerPos[1] - (h / 2);
};
```

3. Define an enumerated data type with values that identify the collision sides of a bounding box.

```
BoundingBox.eboundCollideStatus = Object.freeze({
    eCollideLeft: 1,
    eCollideRight: 2,
    eCollideTop: 4,
    eCollideBottom: 8,
    eInside : 16,
    eOutside: 0
});
```

Notice that each enumerated value is different and has only one nonzero bit. This allows the enumerated values to be combined with the bitwise-or operator to represent a multisided collision. For example, if an object collides with both the top and left sides of a bounding box, the collision status will be `eCollideLeft | eCollideTop = 4 | 1 = 5`.

4. Define a function to determine whether a given position, (x, y), is within the bounds of the box.

```
BoundingBox.prototype.containsPoint = function (x, y) {
    return ((x > this.minX()) && (x < this.maxX()) &&
           (y > this.minY()) && (y < this.maxY()));
};
```

5. Define a function to determine whether a given bound intersects with the current one.

```
BoundingBox.prototype.intersectsBound = function (otherBound) {
    return ((this.minX() < otherBound.maxX()) &&
           (this.maxX() > otherBound.minX()) &&
           (this.minY() < otherBound.maxY()) &&
           (this.maxY() > otherBound.minY()));
};
```

6. Define a function to compute the intersection status between a given bound and the current one.

```
BoundingBox.prototype.boundCollideStatus = function (otherBound) {
    var status = BoundingBox.eboundCollideStatus.eOutside;
    if (this.intersectsBound(otherBound)) {
        if (otherBound.minX() < this.minX())
            status |= BoundingBox.eboundCollideStatus.eCollideLeft;
        if (otherBound.maxX() > this.maxX())
            status |= BoundingBox.eboundCollideStatus.eCollideRight;
        if (otherBound.minY() < this.minY())
            status |= BoundingBox.eboundCollideStatus.eCollideBottom;
        if (otherBound.maxY() > this.maxY())
            status |= BoundingBox.eboundCollideStatus.eCollideTop;
        // if the bounds intersects and yet none of the sides overlaps
        // otherBound is completely inside thisBound
        if (status === BoundingBox.eboundCollideStatus.eOutside)
            status = BoundingBox.eboundCollideStatus.eInside;
    }
    return status;
};
```

Notice the subtle yet important difference between the `intersectsBound()` and `boundCollideStatus()` functions where the former is capable only of returning a `true` or `false` condition while the latter function encodes the colliding sides in the returned `status`.

7. Implement the functions that returns the X/Y values to the min and max bounds of the bounding box.

```
BoundingBox.prototype.minX = function () { return this.mLL[0]; };
BoundingBox.prototype.maxX = function () { return this.mLL[0] + this.mWidth; };
BoundingBox.prototype.minY = function () { return this.mLL[1]; };
BoundingBox.prototype.maxY = function () { return this.mLL[1] + this.mHeight;};
```

Use the BoundingBox in the Engine

Follow these steps:

1. Modify the `GameObject` class to return the bounding box of the unrotated `Renderable` object.

```
GameObject.prototype.getBBox = function () {
    var xform = this.getXform();
    var b = new BoundingBox(xform.getPosition(),xform.getWidth(), xform.getHeight());
    return b;
};
```

2. Modify the `Camera` object to compute the collision status between the bounds to a `Transform` object (typically defined in a `Renderable` object) and that of the WC window.

```
Camera.prototype.collideWCBound = function (aXform, zone) {
    var bbox = new BoundingBox(aXform.getPosition(),aXform.getWidth(), aXform.getHeight());
    var w = zone * this.getWCWidth();
    var h = zone * this.getWCHeight();
    var cameraBound = new BoundingBox(this.getWCCenter(), w, h);
    return cameraBound.boundCollideStatus(bbox);
};
```

Notice that the `zone` parameter defines the relative size of WC that should be used in the collision computation. For example, a `zone` value of 0.8 would mean computing for intersection status based on 80 percent of the current WC window size. Figure 6-13 shows how the camera collides with an object.

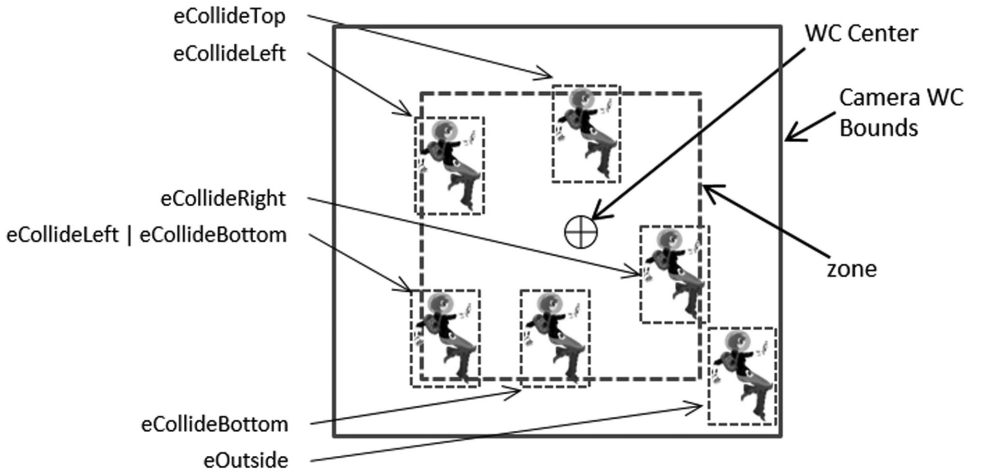

Figure 6-13. *Camera WC Bounds colliding with the bounds defined a Transform object*

Test Bounding Boxes with MyGame

The goal of this test case is to verify the correctness of the bounding box implementation in detecting object-object and object-camera intersections. Once again, with the exception of the `update()` function, the majority of the code in the `MyGame.js` file is similar to the previous projects and is not repeated here. The `update()` function is modified from the previous project to test for bounding box intersections.

```
MyGame.prototype.update = function () {
    // ... identical code to previous project ...

    switch (this.mMode) {
    case 'H':
        this.mBrain.update();  // player steers with arrow keys
        break;
    case 'K':
        rate = 0.02;    // graduate rate
        // no break here on purpose
    case 'J':
        if (!hBbox.intersectsBound(bBbox)) {  // stop brain when touches hero
            this.mBrain.rotateObjPointTo(
                this.mHero.getXform().getPosition(),rate);
            GameObject.prototype.update.call(this.mBrain);
        }
        break;
    }

    // Check for hero going outside 80% of the WC Window bound
    var status = this.mCamera.collideWCBound(this.mHero.getXform(), 0.8);

    // ... identical code to previous project ...
    this.mMsg.setText(msg + this.mMode + " [Hero bound=" + status + "]");
};
```

The previous modification tests for bounding box collision between the `Brain` and `Hero` objects before invoking the `Brain.rotateObjPointTo()` and `update()` functions to cause the chasing behavior. In this way, the `Brain` object will stop moving as soon as it touches the bounds of the `Hero` object. In addition, the collision results between the `Hero` object and 80 percent of the camera WC window are computed and echoed.

You can now run the project and observe the `Brain` object stop moving as soon as it touches the `Hero` object. When you move the `Hero` object around, observe that the output `Hero Bound` message echoes WC window collisions before the `Hero` object actually touches the WC window bounds. This is a result of the 0.8, or 80 percent, parameter passed to the `mCamera.collideWCBound()` function, configuring the collision computation to 80 percent of the current WC window size. When the `Hero` object is completely within 80 percent of the WC window bounds, the output `Hero Bound` value is 16, or the value of `eboundcollideStatus.eInside`. Try moving the `Hero` object to touch the top 20 percent of the window bound and observe the `Hero Bound` value of 4, or the value of `eboundcollideStatus.eCollideTop`. Now move the `Hero` object toward the top-left corner of the window and observe the `Hero Bound` value of 5, or `eboundcollideStatus.eCollideTop | eboundcollideStatus.eCollideLeft`. In this way, the collision status is a bitwise-or result of all the colliding bounds.

Per-Pixel Collisions

In the previous example, you saw the results of bounding box collision approximation. Namely, the `Brain` object's motion stops as soon as its bounds overlap that of the `Hero` object. This is much improved over the original situation where the `Brain` object never stops moving. However, as illustrated in Figure 6-14, there are two serious limitations to the bounding box–based collisions.

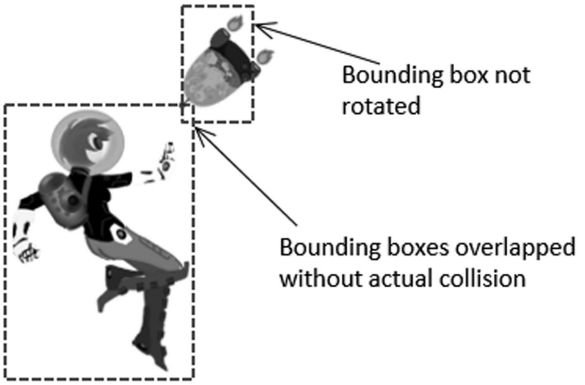

Figure 6-14. *Limitation with bounding box–based collision*

1. The `BoundingBox` object introduced in the previous example does not account for rotation. This is a well-known limitation for axes-aligned bounding boxes: although the approach is computationally efficient, it does not support rotation.
2. The two objects do not actually collide. The fact that the bounds of two objects overlap does not automatically equate to the two objects colliding.

In this project, you will see how to achieve per-pixel-accurate collision detection, pixel-accurate collision, or per-pixel collision, which identifies the overlapping of nontransparent pixels of two colliding objects. However, keep in mind that this is not an end-all solution. While the collision detection is precise, the trade-off is potential performance costs. As an image becomes larger and more complex, it also has more pixels that need to be checked for collisions. This is in contrast to the constant computation cost required for bounding box collision detection from the previous example.

The Per-Pixel Collisions Project

This project demonstrates how to detect collision between a large textured object, the `Collector` minion and a small textured object, the `Portal` minion. Both of the textures contain transparent and nontransparent areas. A collision occurs only when the nontransparent pixels overlap. In this project, when a collision occurs, a yellow `DyePack` appears at the collision point. You can see an example of this project in Figure 6-15. The source code to this project is defined in the `Chapter6/6.4.PerPixelCollisions` folder.

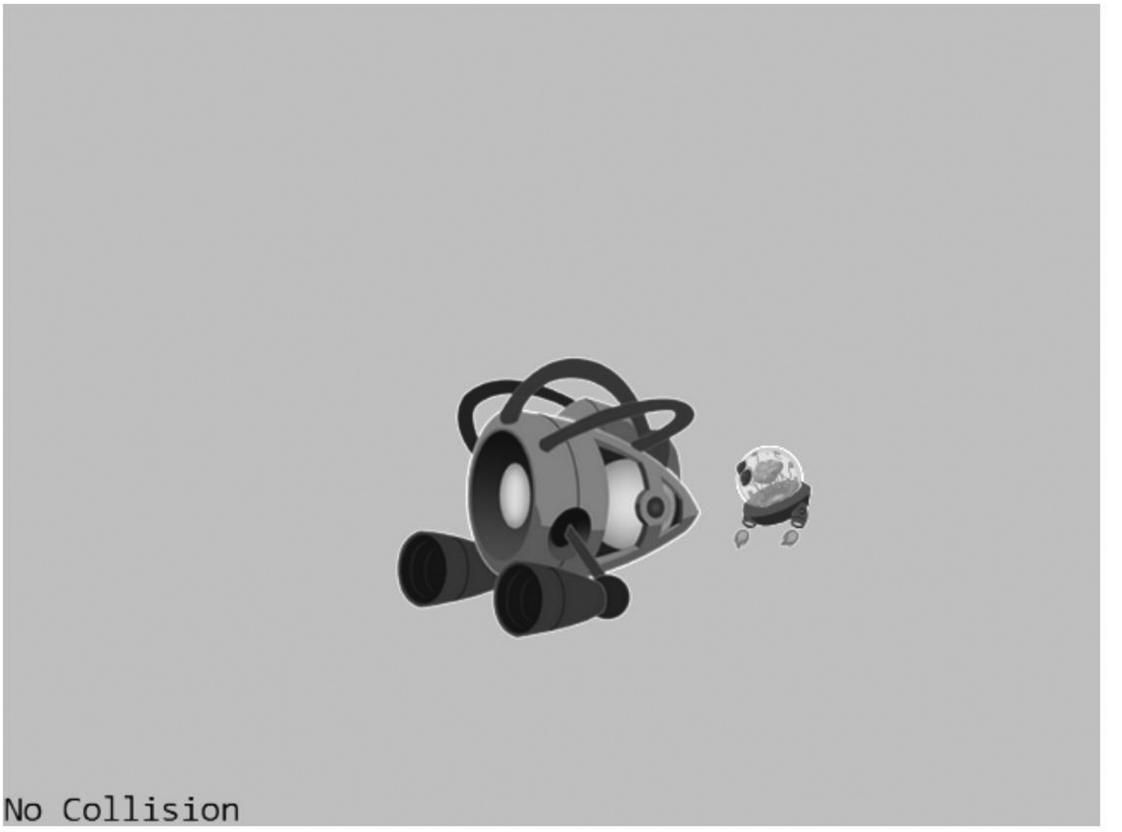

Figure 6-15. *Running the Per-Pixel Collisions project*

The controls of the project are as follows:

- *Arrow keys*: Move the small textured object, the `Portal` minion
- *WASD keys*: Move the large textured object, the `Collector` minion

The goals of the project are as follows:

- To demonstrate how to detect nontransparent pixel overlap
- To understand the pros and cons of using per-pixel-accurate collision detection

You can find the following external resources in the `assets` folder: the `fonts` folder that contains the default system fonts, `minion_collector.png`, `minion_portal.png`, and `minion_sprite.png`. Note that `minion_collector.png` is a large, 1024x1024 image, while `minion_portal.png` is a small, 64x64 image; `minion_sprite.png` defines the `DyePack` sprite element.

Overview of Per-Pixel Collision Algorithm

Before moving forward, it is important to identify the requirements for detecting a collision between two textured objects. Foremost is that the texture itself needs to contain an area of transparency in order for this type of collision detection to provide any increase in accuracy. Without transparency in the texture, you can and should use a simple bounding box collision detection. If one or both of the textures contain transparent areas, then you'll need to handle two cases of collision. The first case is to check whether the bounds of the two objects collide. You can see this reflected in Figure 6-16. Notice how the bounds of the objects overlap yet none of the nontransparent colored pixels are touching.

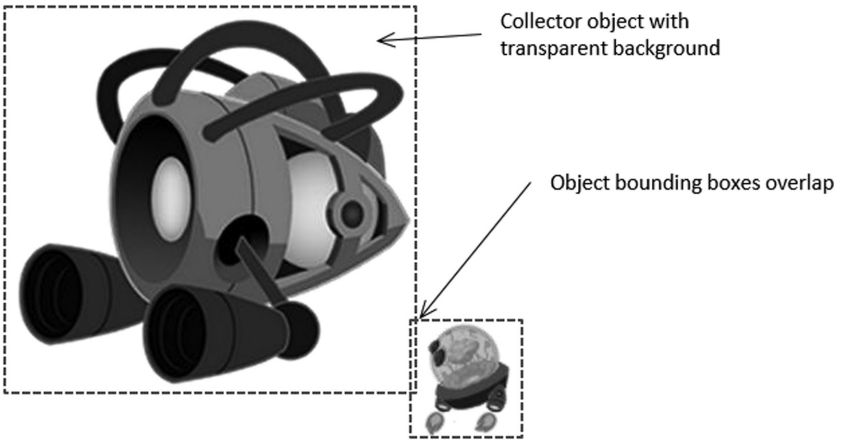

Figure 6-16. *Overlapping bounding boxes without actual collision*

The next case is to check whether the nontransparent pixels of the textures overlap. Take a look at Figure 6-17. Nontransparent pixels from the textures of the `Collector` and `Portal` objects are clearly in contact with one another.

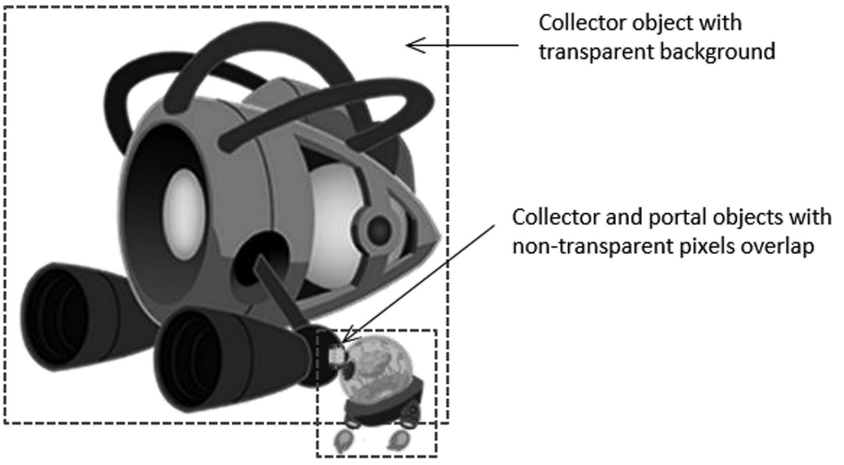

Figure 6-17. *Pixel collision occurring between the large texture and the small texture*

Now that the problem is clearly defined, here is the logic or pseudocode for per-pixel-accurate collision detection:

```
Given two images, Image-A and Image-B
If the bounds of the object of Image-A and Image-B collide then
    For each Pixel-A in Image-A
        pixelCameraSpace = Pixel-A position in camera space
        Transform pixelCameraSpace to Image-B space
        Read Pixel-B from Image-B
            If Pixel-A and Pixel-B are not both completely transparent then
                A collision has occurred
```

The per-pixel transformation to Image-B space from `pixelCameraSpace` is required because collision checking must be carried out within the same coordinate space. Additionally, pay attention to the runtime of this algorithm. Each pixel within Image-A must be checked, so the runtime is O(N), where N is equal to the number of pixels in Image-A, or Image-A's resolution. To mitigate this performance hit, you should use the smaller of the two images (the `portal` minion in this case) as Image-A. However, at this point, you can probably see why the performance of pixel-accurate collision detection is concerning. Checking for these collisions during every update with many high-resolution textures onscreen can quickly bog down performance. You are now ready to examine the implementation of per-pixel accurate collision.

Modify Engine_Textures to Load a Texture as an Array of Color

Recall that the `Engine_Texture` component reads image files from the server file system, loads the images to the WebGL context, and processes the images into WebGL textures. In this way, there is no actual storage of the file texture in the game engine. To support per-pixel collision detection, the color information must be retrieved from the WebGL context. The `Engine_Texture` component can be modified to support this requirement.

1. In the `Engine_Textures.js` file, expand the `TextureInfo` object to include a new variable for storing the color array of a file texture.

```
function TextureInfo(name, w, h, id) {
    this.mName = name;
    this.mWidth = w;
    this.mHeight = h;
    this.mGLTexID = id;
    this.mColorArray = null;
}
```

2. Define a function to retrieve the color array from the WebGL context and remember to add this new function to the public interface of `Engine_Texture`.

```
var getColorArray = function (textureName) {
    var texInfo = getTextureInfo(textureName);
    if (texInfo.mColorArray === null) {
        // create a framebuffer, bind it to the texture, and read the color content
        var gl = gEngine.Core.getGL();
        var fb = gl.createFramebuffer();
        gl.bindFramebuffer(gl.FRAMEBUFFER, fb);
        gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0, gl.TEXTURE_2D,
            texInfo.mGLTexID, 0);
```

```
        if (gl.checkFramebufferStatus(gl.FRAMEBUFFER) === gl.FRAMEBUFFER_COMPLETE) {
            var pixels = new Uint8Array(texInfo.mWidth * texInfo.mHeight * 4);
            gl.readPixels(0, 0, texInfo.mWidth, texInfo.mHeight, gl.RGBA,
                gl.UNSIGNED_BYTE, pixels);
            texInfo.mColorArray = pixels;
        } else {
            alert("WARNING: Engine.Textures.GetColorArray() failed!");
        }

        gl.bindFramebuffer(gl.FRAMEBUFFER, null);
        gl.deleteFramebuffer(fb);
    }
    return texInfo.mColorArray;
};

var mPublic = {
    loadTexture: loadTexture,
    unloadTexture: unloadTexture,
    activateTexture: activateTexture,
    deactivateTexture: deactivateTexture,
    getTextureInfo: getTextureInfo,
    getColorArray: getColorArray
};
```

The previous code creates a WebGL `FRAMEBUFFER`, fills the buffer with the desired texture, and retrieves the buffer content into the associated `texInfo.mColorArray`.

Modify TextureRenderable to Support Per-Pixel Collision

The `TextureRenderable` object is the most appropriate for implementing the per-pixel collision functionality. This is because the `TextureRenderable` object is the base class for all classes that render textures. Implementation in this base class means all subclasses can inherit the functionality with minimal additional changes.

1. Edit the `TextureRenderable.js` file and modify the constructor to add instance variables to hold texture information for supporting per-pixel collision detection and for later subclass overrides:

```
function TextureRenderable(myTexture) {
    Renderable.call(this);
    Renderable.prototype.setColor.call(this, [1, 1, 1, 0]);
    Renderable.prototype._setShader.call(this, gEngine.DefaultResources.getTextureShader());
    this.mTexture = null;
    // these two instance variables are to cache texture information
    // for supporting per-pixel accurate collision
    this.mTextureInfo = null;
    this.mColorArray = null;
    // defined for subclass to override
    this.mTexWidth = 0;
    this.mTexHeight = 0;
```

```
    this.mTexLeftIndex = 0;
    this.mTexBottomIndex = 0;
    this.setTexture(myTexture);
}
```

2. Modify the `setTexture()` function to initialize the instance variables accordingly.

```
TextureRenderable.prototype.setTexture = function (newTexture) {
    this.mTexture = newTexture;
    this.mTextureInfo = gEngine.Textures.getTextureInfo(newTexture);
    this.mColorArray = null;
    this.mTexWidth = this.mTextureInfo.mWidth;
    this.mTexHeight = this.mTextureInfo.mHeight;
    this.mTexLeftIndex = 0;
    this.mTexBottomIndex = 0;
};
```

Note that the `mTexWidth/Height` variables are defined for later subclass overrides such that the algorithm can support the collision of sprite elements.

Create TextureRenderable_PixelCollision.js File to Implement Per-Pixel Collision

To maintain the readability of the source code, create a new file in the `src/Engine/Renderables/` folder and name it `TextureRenderable_PixelCollision.js`. All per-pixel collision functionality for the `TextureRenderable` object will be implemented in the `TextureRenderable_PixelCollision.js` file. Remember to include commands to load this new file in `index.html`.

■ **Note** Source files with names in the form of `object_functionality.js` implement `functionality` for `object`. By separating complex implementations of related functionality into separate files, the readability of each source code file can be maintained.

---

1. Edit the `TextureRenderable_PixelCollision.js` file and define a function to set the `mColorArray`.

```
TextureRenderable.prototype.setColorArray = function () {
    if (this.mColorArray === null)
        this.mColorArray = gEngine.Textures.getColorArray(this.mTexture);
};
```

2. Define a function to return the alpha value, or the transparency, of any given pixel (`x`, `y`).

```
TextureRenderable.prototype._pixelAlphaValue = function (x, y) {
    x = x * 4;
    y = y * 4;
    return this.mColorArray[(y * this.mTextureInfo.mWidth) + x  + 3];
};
```

Notice that the mColorArray is a one-dimensional array where colors of pixels are stored as four floats and organized by rows.

3. Define a function to compute the WC position (returnWCPos) of a given pixel (i, j).

```
TextureRenderable.prototype._indexToWCPosition = function (returnWCPos, i, j) {
    var x = i * this.mXform.getWidth() / (this.mTexWidth - 1);
    var y = j * this.mXform.getHeight() / (this.mTexHeight - 1);
    returnWCPos[0] = this.mXform.getXPos() + (x - (this.mXform.getWidth() * 0.5));
    returnWCPos[1] = this.mXform.getYPos() + (y - (this.mXform.getHeight() * 0.5));
};
```

4. Now, implement the inverse of the previous function and use a WC position (wcPos) to compute the texture pixel indices (returnIndex).

```
TextureRenderable.prototype._wcPositionToIndex = function (returnIndex, wcPos) {
    // use wcPos to compute the corresponding returnIndex[0 and 1]
    var delta = [];
    vec2.sub(delta, wcPos, this.mXform.getPosition());
    returnIndex[0] = this.mTexWidth  * (delta[0] / this.mXform.getWidth());
    returnIndex[1] = this.mTexHeight * (delta[1] / this.mXform.getHeight());

    // recall that xForm.getPosition() returns center, yet Texture origin is at lower-left corner!
    returnIndex[0] += this.mTexWidth / 2;
    returnIndex[1] += this.mTexHeight / 2;

    returnIndex[0] = Math.floor(returnIndex[0]);
    returnIndex[1] = Math.floor(returnIndex[1]);
};
```

5. Now, it is possible to implement the outlined per-pixel collision algorithm.

```
TextureRenderable.prototype.pixelTouches = function(other, wcTouchPos) {
    var pixelTouch = false;
    var xIndex = 0, yIndex;
    var otherIndex = [0, 0];

    while ((!pixelTouch) && (xIndex < this.mTexWidth)) {
        yIndex = 0;
        while ((!pixelTouch) && (yIndex < this.mTexHeight)) {
            if (this._pixelAlphaValue(xIndex, yIndex) > 0) {
                this._indexToWCPosition(wcTouchPos, xIndex, yIndex);
                other._wcPositionToIndex(otherIndex, wcTouchPos);
                if ((otherIndex[0] > 0) && (otherIndex[0] < other.mTexWidth) &&
                    (otherIndex[1] > 0) && (otherIndex[1] < other.mTexHeight)) {
                    pixelTouch = other._pixelAlphaValue(otherIndex[0], otherIndex[1]) > 0;
                }
            }
            yIndex++;
        }
        xIndex++;
    }
    return pixelTouch;
};
```

The parameter other is a reference to the other TextureRenderable object that is being tested for collision. If pixels do overlap between the objects, the returned value of wcTouchPos is the first detected colliding position in the WC space. Notice that the nested loops terminate as soon as one pixel overlap is detected or when pixelTouch becomes true. This is an important feature for efficiency concerns. However, this also means that the returned wcTouchPos is simply one of many potentially colliding points.

Support GameObject Per-Pixel Collision in GameObject_PixelCollision.js

The GameObject class must be modified to support per-pixel collision. Create a new file in the src/Engine/GameObjects/ folder and name it GameObject_PixelCollision.js. As always, remember to include the loading of this new file in index.html. This new file will implement the support for per-pixel collision of the GameObject class.

```
GameObject.prototype.pixelTouches = function (otherObj, wcTouchPos) {
    // only continue if both objects have GetColorArray defined
    var pixelTouch = false;
    var myRen = this.getRenderable();
    var otherRen = otherObj.getRenderable();

    if ((typeof myRen.pixelTouches === "function") &&
        (typeof otherRen.pixelTouches === "function")) {
        var otherBbox = otherObj.getBBox();
        if (otherBbox.intersectsBound(this.getBBox())) {
            myRen.setColorArray();
            otherRen.setColorArray();
            pixelTouch = myRen.pixelTouches(otherRen, wcTouchPos);
        }
    }
    return pixelTouch;
};
```

This function checks to ensure that the objects are colliding and delegates the actual per-pixel collision to TextureRenderable objects. Notice the intersectsBound() function for a bounding box intersection check before invoking the potentially expensive TextureRenderable.pixelTouches() function.

Test the Per-Pixel Collision in MyGame

As illustrated in Figure 6-15, the testing of per-pixel collision is rather straightforward, involving three instances of GameObject instances: the large Collector minion, the small Portal minion, and the DyePack. The Collector and Portal minions are controlled by arrow and WASD keys, respectively. The detail of the implementation of MyGame is similar to previous projects and is not shown. The only noteworthy code fragment is the collision testing in the update() function, as shown here:

```
MyGame.prototype.update = function () {
    var msg = "No Collision";

    this.mPortal.update(gEngine.Input.keys.W, gEngine.Input.keys.S,
                    gEngine.Input.keys.A, gEngine.Input.keys.D);
    this.mCollector.update(gEngine.Input.keys.Up, gEngine.Input.keys.Down,
                      gEngine.Input.keys.Left, gEngine.Input.keys.Right);
```

```
    var h = [];
    // Portal's resolution is 1/16 that of Collector!
    // if (this.mCollector.pixelTouches(this.mPortal, h)) { // VERY EXPENSIVE!!
    if (this.mPortal.pixelTouches(this.mCollector, h)) {
        msg = "Collided!: (" + h[0].toPrecision(4) + " " + h[1].toPrecision(4) + ")";
        this.mDyePack.setVisibility(true);
        this.mDyePack.getXform().setXPos(h[0]);
        this.mDyePack.getXform().setYPos(h[1]);
    } else {
        this.mDyePack.setVisibility(false);
    }
    this.mMsg.setText(msg);
};
```

You can now test the collision accuracy by moving the two minions and intersecting them at different locations (for example, top colliding with the bottom, left colliding with the right) or moving them such that there are large overlapping areas. Notice that it is rather difficult, if not impossible, to predict the actual reported intersection position (position of the `DyePack`). It is important to remember that the per-pixel collision function is mainly a function that returns `true` or `false` indicating whether there is a collision. You cannot rely on this function to compute the actual collision position. Lastly, try switching to calling the `Collector.pixelTouches()` function to detect collisions. Notice the less than real-time performance! In this case, the computation cost of the `Collector.pixelTouches()` function is 16×16=256 times that of the `Portal.pixelTouches()` function.

Generalized Per-Pixel Collisions

In the previous section, you saw the basic operations required to achieve per-pixel-accurate collision detection. However, as you may have noticed, the previous project applies only when the textures are aligned along the x/y-axes. This means that your implementation does not support collisions between rotated objects.

This section explains how you can achieve per-pixel-accurate collision detection when objects are rotated. The fundamental concepts of this project are the same as in the previous project; however, this version involves working with vector decomposition, and a quick review can be helpful.

Vector Review: Components and Decomposition

Recall that two perpendicular directions can be used to decompose a vector into corresponding components. For example, Figure 6-18 contains two normalized vectors, or the component vectors, that can be used to decompose the vector $\vec{V} = (2,3)$: the normalized component vectors $\hat{i} = (1,0)$ and $\hat{j} = (1,0)$ decompose the vector $\vec{V}$ into the components $2\hat{i}$ and $3\hat{j}$.

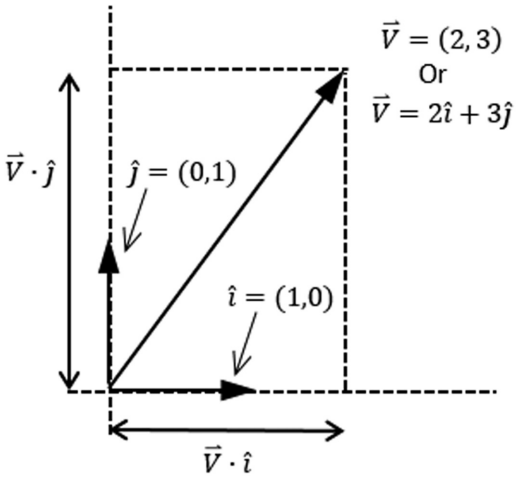

Figure 6-18. *The decomposition of vector* $\vec{V}$

With that in mind, given the normalized perpendicular component vectors $\hat{L}$ and $\hat{M}$ and any vector $\vec{V}$, the following formulas will always be true. You can see a representation of this principle in Figure 6-19.

$$\vec{V} = \left(\vec{V} \cdot \hat{i}\right)\hat{i} + \left(\vec{V} \cdot \hat{j}\right)\hat{j}$$

$$\vec{V} = \left(\vec{V} \cdot \hat{L}\right)\hat{L} + \left(\vec{V} \cdot \hat{M}\right) \cdot \hat{M}$$

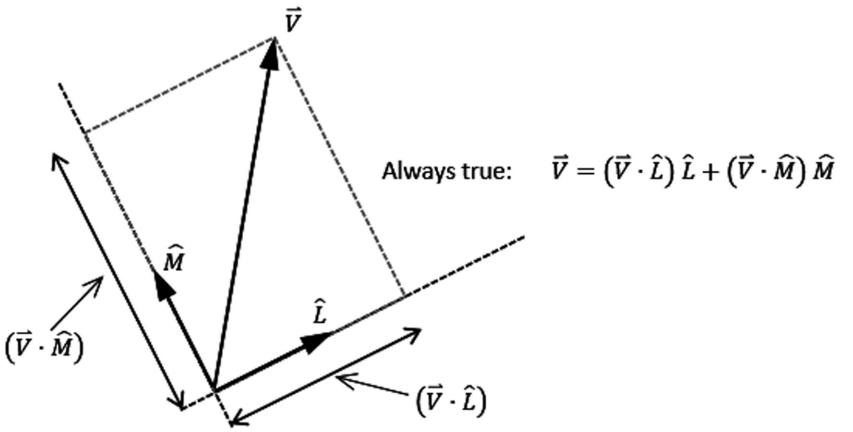

Figure 6-19. *Decomposing a vector by two normalized component vectors*

Vector decomposition is relevant to this project because of the rotated image axes. Without rotation, an image can be referenced by the familiar normalized perpendicular set of vectors along the default x-axis ($\hat{i}$) and y-axis ($\hat{j}$). You handled this case in the previous project. You can see an example of this in Figure 6-20.

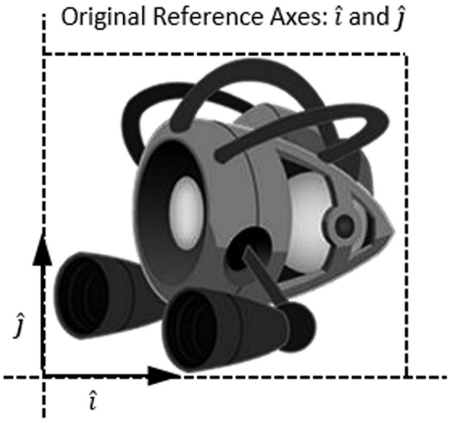

Figure 6-20. *An axes-aligned texture*

However, after the image has been rotated, the reference vector set no longer resides along the x/y-axes. Therefore, the collision computation must take into account the newly rotated axes $\hat{L}$ and $\hat{M}$, as shown in Figure 6-21.

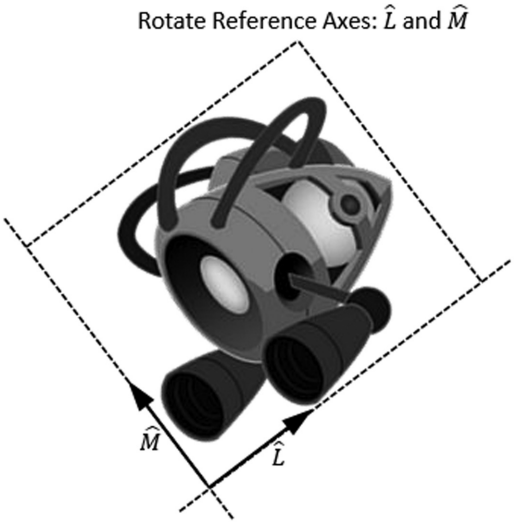

Figure 6-21. *A rotated texture and its component vectors*

The General Pixel Collisions Project

This project demonstrates how to detect a collision between two rotated `TextureRenderable` objects with per-pixel accuracy. As in the previous project, a yellow `DyePack` object (as a test confirmation) will be displayed at the detected colliding position. You can see an example of this project running in Figure 6-22. The source code to this project is defined in the `Chapter6/6.5.GeneralPixelCollisions` folder.

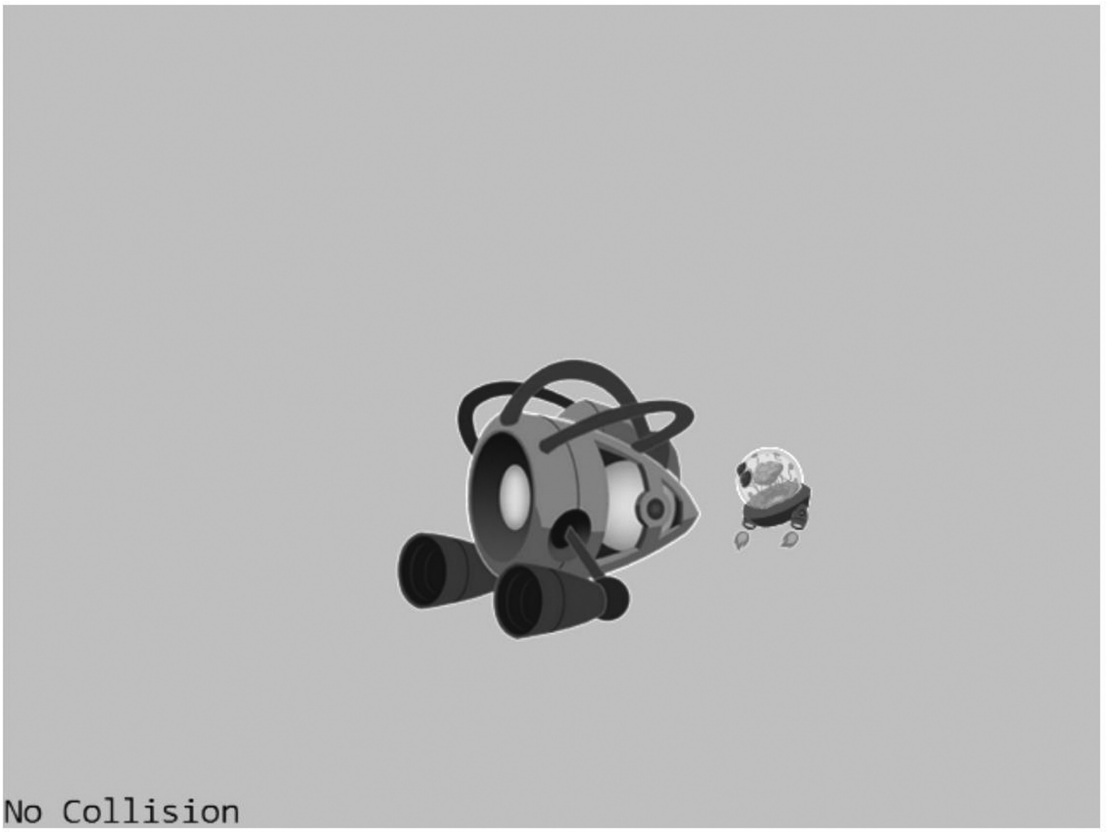

Figure 6-22. *Running the General Pixel Collisions project*

The controls of the project are as follows:

- *Arrow keys*: Move the small textured object, the `Portal` minion
- *P key*: Rotates the small textured object, the `Portal` minion
- *WASD keys*: Move the large textured object, the `Collector` minion
- *E key*: Rotates the large textured object, the `Collector` minion

The goals of the project are as follows:

- To access pixels of a rotated image via vector decomposition
- To support per-pixel accurate collision detection between two rotated textured objects

You can find the same external resource files as in the previous project in the `assets` folder.

Modify TextureRenderable_PixelCollision to Support Rotation

1. Edit the TextureRenderable_PixelCollision.js file and modify the _indexToWCPosition() function accordingly.

```
TextureRenderable.prototype._indexToWCPosition = function (returnWCPos, i, j, xDir, yDir) {
    var x = i * this.mXform.getWidth() / (this.mTexWidth - 1);
    var y = j * this.mXform.getHeight() / (this.mTexHeight - 1);
    var xDisp = x - (this.mXform.getWidth() * 0.5);
    var yDisp = y - (this.mXform.getHeight() * 0.5);
    var xDirDisp = [];
    var yDirDisp = [];

    vec2.scale(xDirDisp, xDir, xDisp);
    vec2.scale(yDirDisp, yDir, yDisp);
    vec2.add(returnWCPos, this.mXform.getPosition(), xDirDisp);
    vec2.add(returnWCPos, returnWCPos, yDirDisp);
};
```

In the previous code, xDir and yDir are the $\hat{L}$ and $\hat{M}$ normalized component vectors. The variables xDisp and yDisp are the displacements to be offset along the xDir and yDir, respectively. The returned value of returnWCPos is a simple displacement from the object's center position along the xDirDisp and yDirDisp vectors, the scaled xDir and yDir vectors.

2. In a similar fashion, modify the _wcPositionToIndex() function to support the rotated normalized vector components.

```
TextureRenderable.prototype._wcPositionToIndex = function (returnIndex, wcPos, xDir, yDir) {
    // use wcPos to compute the corresponding returnIndex[0 and 1]
    var delta = [];
    vec2.sub(delta, wcPos, this.mXform.getPosition());
    var xDisp = vec2.dot(delta, xDir);
    var yDisp = vec2.dot(delta, yDir);
    returnIndex[0] = this.mTexWidth  * (xDisp / this.mXform.getWidth());
    returnIndex[1] = this.mTexHeight * (yDisp / this.mXform.getHeight());

    // recall that xForm.getPosition() returns lower-left corner, yet
    // Texture origin is at lower-left corner!
    returnIndex[0] += this.mTexWidth / 2;
    returnIndex[1] += this.mTexHeight / 2;

    returnIndex[0] = Math.floor(returnIndex[0]);
    returnIndex[1] = Math.floor(returnIndex[1]);
};
```

3. The pixelTouches() function needs to be modified to compute the rotated normalized component vectors.

```
TextureRenderable.prototype.pixelTouches = function(other, wcTouchPos) {
    var pixelTouch = false;
    var xIndex = 0, yIndex;
    var otherIndex = [0, 0];
```

```
    var xDir = [1, 0];
    var yDir = [0, 1];
    var otherXDir = [1, 0];
    var otherYDir = [0, 1];
    vec2.rotate(xDir, xDir, this.mXform.getRotationInRad());
    vec2.rotate(yDir, yDir, this.mXform.getRotationInRad());
    vec2.rotate(otherXDir, otherXDir, other.mXform.getRotationInRad());
    vec2.rotate(otherYDir, otherYDir, other.mXform.getRotationInRad());

    while ((!pixelTouch) && (xIndex < this.mTexWidth)) {
        yIndex = 0;
        while ((!pixelTouch) && (yIndex < this.mTexHeight)) {
            if (this._pixelAlphaValue(xIndex, yIndex) > 0) {
                this._indexToWCPosition(wcTouchPos, xIndex, yIndex,
                    xDir, yDir);
                other._wcPositionToIndex(otherIndex, wcTouchPos,
                    otherXDir, therYDir);
                if ((otherIndex[0] > 0) && (otherIndex[0] < other.mTexWidth) &&
                    (otherIndex[1] > 0) && (otherIndex[1] < other.mTexHeight)) {
                    pixelTouch = other._pixelAlphaValue (otherIndex[0], otherIndex[1]) > 0;
                }
            }
            yIndex++;
        }
        xIndex++;
    }
    return pixelTouch;
};
```

The variables `xDir` and `yDir` are the rotated normalized component vectors $\hat{L}$ and $\hat{M}$ of this `TextureRenderable` object, while `otherXDir` and `otherYDir` are those of the colliding object. These vectors are used as references for computing transformations from texture index to WC and from WC to texture index.

Modify the GameObject_PixelCollision.js to Support Rotation

Recall that the `GameObject` class first tests for the bounding-box collision between two objects before it actually invokes the much more expensive per-pixel collision computation. As illustrated in Figure 6-14, the `BoundingBox` object does not support object rotation correctly, and the following code remedies this shortcoming:

```
GameObject.prototype.pixelTouches = function (otherObj, wcTouchPos) {
    // only continue if both objects have getColorArray defined
    // if defined, should have other texture intersection support!
    var pixelTouch = false;
    var myRen = this.getRenderable();
    var otherRen = otherObj.getRenderable();

    if ((typeof myRen.pixelTouches === "function") &&
        (typeof otherRen.pixelTouches === "function")) {
        if ((myRen.getXform().getRotationInRad() === 0) &&
            (otherRen.getXform().getRotationInRad() === 0)) {
```

```
            // no rotation, we can use bbox ...
            var otherBbox = otherObj.getBBox();
            if (otherBbox.intersectsBound(this.getBBox())) {
                myRen.setColorArray();
                otherRen.setColorArray();
                pixelTouch = myRen.pixelTouches(otherRen, wcTouchPos);
            }
        } else {
            // One or both are rotated, compute an encompassing circle by using the
            hypertenuse as radius
            var mySize = myRen.getXform().getSize();
            var otherSize = otherRen.getXform().getSize();
            var myR = Math.sqrt(0.5*mySize[0]*0.5*mySize[0] + 0.5*mySize[1]*0.5*mySize[1]);
            var otherR = Math.sqrt(0.5*otherSize[0]*0.5*otherSize[0] +
                                   0.5*otherSize[1]*0.5*otherSize[1]);
            var d = [];
            vec2.sub(d, myRen.getXform().getPosition(), otherRen.getXform().getPosition());
            if (vec2.length(d) < (myR + otherR)) {
                myRen.setColorArray();
                otherRen.setColorArray();
                pixelTouch = myRen.pixelTouches(otherRen, wcTouchPos);
            }
        }
    }
    return pixelTouch;
};
```

The previous code shows that if either of the colliding objects is rotated, then two encompassing circles are used to determine whether the objects are sufficiently close for the expensive per-pixel collision computation. The two circles are defined with radii equal to the hypotenuse of the x/y size of the corresponding `TextureRenderable` objects. The per-pixel collision detection is invoked only if the distance between these two circles is less than the sum of the radii.

Test Generalized Per-Pixel Collision

The code for testing the rotated `TextureRenderable` objects is essentially identical to that from the previous project, with the exception of the two added controls for rotations. The details of the implementation are not shown. You can now run the project, rotate the two objects, and observe the accurate collision results.

Per-Pixel Collisions for Sprites

The previous project implicitly assumes that the `Renderable` object is covered by the entire texture map. This assumption means that the per-pixel collision implementation does not support sprite or animated sprite objects. In this section, you will see how to remedy this deficiency.

The Sprite Pixel Collisions Project

This project demonstrates how to move an animated sprite object around the screen and perform per-pixel collision detection with other objects. The project tests for the correctness between collisions of `TextureRenderable`, `SpriteRenderable`, and `SpriteAnimateRenderable` objects. You can see an example of this project running in Figure 6-23. The source code to this project is defined in the `Chapter6/6.6.SpritePixelCollisions` folder.

Figure 6-23. *Running the Sprite Pixel Collisions project*

The controls of the project are as follows:

- *Arrow and P keys*: Move and rotate the `Portal` minion
- *WASD keys*: Move the `Hero`
- *L, R, H, B keys*: Select the target for colliding with the `Portal` minion

The goal of the project is as follow:

- To implement per-pixel collision detection for sprite and animated sprite objects

You can find the following external resource file in the `assets` folder: the `fonts` folder that contains the default system fonts and `minion_sprite.png`, and `minion_portal.png`.

Create SpriteRenderable_PixelCollision.js file to Implement Per-Pixel Collision

Create a new file in the src/Engine/Renderables/ folder and name it SpriteRenderable_PixelCollision.js. This file implements the per-pixel specific support for SpriteRenderable objects. Remember to include commands to load this new file in index.html. In this file, define the _setTexInfo() function to override instance variables defined in the superclass, TextureRenderable, such that these variables identify the currently active sprite element.

```
SpriteRenderable.prototype._setTexInfo = function () {
    var imageW = this.mTextureInfo.mWidth;
    var imageH = this.mTextureInfo.mHeight;

    this.mTexLeftIndex = this.mTexLeft * imageW;
    this.mTexBottomIndex = this.mTexBottom * imageH;

    this.mTexWidth = ((this.mTexRight - this.mTexLeft) * imageW) + 1;
    this.mTexHeight = ((this.mTexTop - this.mTexBottom) * imageH) + 1;
};
```

Notice that instead of the dimension of the entire texture map, mTexWidth/Height now contain values that correspond to the dimension of a single sprite element in the sprite sheet.

Modify Existing SpriteRenderable Functions to Support Per-Pixel Collision

Follow these steps:

1. Edit the SpriteRenderable.js file and modify the constructor to call the newly defined _setTexInfo() function to properly record the dimension of a sprite element.

```
function SpriteRenderable(myTexture) {
    TextureRenderable.call(this, myTexture);
    Renderable.prototype._setShader.call(this, gEngine.DefaultResources.getSpriteShader());
    this.mTexLeft = 0.0;   // bounds of texture coordinate (0 is left, 1 is right)
    this.mTexRight = 1.0;  //
    this.mTexTop = 1.0;    //   1 is top and 0 is bottom of image
    this.mTexBottom = 0.0; //
    this._setTexInfo();
}
```

2. Remember to call the _setTexInfo() function when the current sprite element is updated in the setElementUVCooridnate() and setElementPixelPositions() functions.

```
SpriteRenderable.prototype.setElementUVCoordinate = function (left, right, bottom, top) {
    this.mTexLeft = left;
    this.mTexRight = right;
    this.mTexBottom = bottom;
    this.mTexTop = top;
    this._setTexInfo();
};
```

```
SpriteRenderable.prototype.setElementPixelPositions = function (left, right, bottom, top) {
    var imageW = this.mTextureInfo.mWidth;
    var imageH = this.mTextureInfo.mHeight;
    this.mTexLeft = left / imageW;
    this.mTexRight = right / imageW;
    this.mTexBottom = bottom / imageH;
    this.mTexTop = top / imageH;
    this._setTexInfo();
};
```

Modify TextureRenderable to Support Accesses to Sprite Pixels

Edit the `TextureRenderable_PixelCollision.js` file and modify the `_pixelAlphaValue()` function to support pixel accesses with a sprite element index offset.

```
TextureRenderable.prototype._pixelAlphaValue = function (x, y) {
    y += this.mTexBottomIndex;
    x += this.mTexLeftIndex;
    x = x * 4;
    y = y * 4;
    return this.mColorArray[(y * this.mTextureInfo.mWidth) + x  + 3];
};
```

Test Per-Pixel Collision for Sprites in MyGame

The code for testing this project is a simple modification from previous projects, and the details are not listed. It is important to note the different object types in the scene.

- *`Portal minion`*: A simple `TextureRenderable` object
- *`Hero and Brain`*: `SpriteRenderable` objects where the textures shown on the geometries are sprite elements defined in the `minion_sprite.png` sprite sheet
- *`Left and Right minions`*: `SpriteAnimateRenderable` objects with sprite elements defined in the top two rows of the `minion_sprite.png` animated sprite sheet

You can now run this project and observe the correct results from the collisions of the different object types.

1. Try moving the `Hero` object and observe how the `Brain` object constantly seeks out and collides with it. This is the case of collision between two `SpriteRenderable` objects.
2. Press the L/R keys and then move the `Portal` minion with the WASD keys to collide with the `Left` or `Right` minions. Remember that you can rotate the `Portal` minion with the P key. This is the case of collision between `TextureRenderable` and `SpriteAnimatedRenderable` objects.
3. Type the H key and then move the `Portal` minion to collide with the `Hero` object. This is the case of collision between `TextureRenderable` and `SpriteRenderable` objects.
4. Type the B key and then move the `Portal` minion to collide with the `Brain` object. This is the case of collision between rotated `TextureRenderable` and `SpriteRenderable` objects.

Summary

This chapter showed you how to encapsulate common behaviors of objects in games and demonstrated the benefits of the encapsulation in the forms of simpler and better organized control logic in the client `MyGame` test levels. You reviewed vectors in 2D space. A vector is defined by its direction and magnitude. Vectors are convenient for describing defined displacements (velocities). You reviewed some foundational vector operations, including normalization of a vector and how to calculate dot and cross products. You worked with these operators to implement the front-facing direction capability and create simple autonomous behaviors such as pointing toward a specific object and chasing.

The need for detecting object collisions becomes prominent as the objects' behaviors increase in sophistication. The axes-aligned bounding boxes were introduced as a crude and yet computationally efficient solution for approximating object collisions. You learned the algorithm for per-pixel-accurate collision detection and that it is accurate but computationally expensive. You now understand how to mitigate the computational cost in two ways. First, you invoke the pixel-accurate procedure only when the objects are sufficiently close to each other, such as when their bounding boxes collide. Second, you execute the pixel iteration process based on the texture with lower resolution. When implementing pixel-accurate collision, you began with tackling the basic case of working with aligned textures. After that implementation, you went back and added support for collision detection between rotated textures. Finally, you generalized the implementation to support collisions between sprite elements. Solving the easiest case first lets you test and observe the results and helps define what you might need for the more advanced problems (rotation and subregions of a texture in this case).

At the beginning of this chapter, your game engine supported interesting sophistication in drawing ranging from the abilities to define WC space, to view the WC space with the `Camera` object, and to draw visually pleasing textures and animations on objects. However, there was no infrastructure for supporting the behaviors of the objects. This shortcoming resulted in clustering of initialization and control logic in the client-level implementations. With the object behavior abstraction, mathematics, and collision algorithms introduced and implemented in this chapter, your game engine functionality is now better balanced. The clients of your game engine now have tools for encapsulating specific behaviors and detecting collisions. The next chapter reexamines and enhances the functionality of the `Camera` object. You will learn to control and manipulate the `Camera` object and work with multiple `Cameras` in the same game.

Game Design Considerations

Chapters 1–5 introduced foundation techniques for drawing, moving, and animating objects on the screen. The Scene Objects project from Chapter 4 described a simple interaction behavior and showed you how to change the game screen based on the location of a rectangle. Recall that moving the rectangle to the left boundary caused the level to visually change, and the Audio Support project added contextual sound to reinforce the overall sense of presence. Although it's possible to build an intriguing (albeit simple) puzzle game using only the elements from Chapters 1–5, things get much more interesting when you can integrate object detection and collision triggers; these behaviors form the basis for many common game mechanics and provide opportunities to design a wide range of interesting gameplay options.

Starting with the Game Objects project, you can see how the screen elements start working together to convey the game setting. Even with the interaction in this project limited to character movement, the setting is beginning to resolve into something that feels coherent. The hero character appears to be flying through a moving scene populated by a number of mechanized robots; there's also a small object in the center of the screen that you might imagine could become some kind of special pickup.

Even at this basic stage of development, it's possible to brainstorm game mechanics that could potentially form the foundation for a full game. If you were designing a simple game mechanic based on only the screen elements found in the Game Object Abstraction project, what kind of behaviors would you choose and what kind of interactions would you require? As one example, imagine that the hero character must avoid colliding with the flying robots and that perhaps some of the robots will detect and follow the hero; maybe the hero is also "zapped" if she comes into contact with a robot. Imagine also that the small

object in the center of the screen allows the hero to be invincible for a certain duration. With these few basic interactions, you have opened opportunities to explore mechanics and levels typically associated with familiar game genres, all with just the inclusion of the object detection, chase, and collision behaviors covered in Chapter 6. Try this design exercise yourself using just the elements shown in the Game Object Abstraction project. What kinds of simple conditions and behaviors might you design to make your experience unique? How many ways can you think of to use the small object in the center of the screen? The final design project in Chapter 11 will explore these themes in greater detail.

This is also a good opportunity to brainstorm some of the other nine elements of game design discussed in Chapter 1. What if the game wasn't set in space with robots? Perhaps the setting is in a forest, or under water, or even something completely abstract. How might you incorporate audio to enhance the sense of presence and reinforce the game setting? You'll probably be surprised by the variety of settings and scenarios you come up with, and keeping yourself limited to just the elements and interactions covered through Chapter 6 is actually a beneficial exercise; design constraints often help the creative process by shaping and guiding your ideas.

The Vectors: Front and Chase project is interesting from both a game mechanic and presence perspective. Many games, of course, require objects in the game world to detect the hero character and will either chase or try to avoid the player (or both if the object has multiple states). The project also demonstrates two different approaches to chase behavior, instant and smooth pursuit, and the game setting will typically influence which behavior you choose to implement. The choice between instant and smooth pursuit is a great example of subtle behaviors that can significantly influence the sense of presence. If you were designing a game where ships were interacting on the ocean, for example, you would likely want their pursuit behavior to take real-world inertia and momentum into consideration because ships can't instantly turn and respond to changes in movement; rather, they move smoothly and gradually, demonstrating a noticeable delay in how quickly they can respond to a moving target. Most objects in the physical world will display the same inertial and momentum constraint to some degree, but there are also situations where you may want game objects to respond directly to path changes (or, perhaps, you want to intentionally flout real-world physics and create a behavior that isn't based on the limitations of physical objects). The key is to always be intentional about your design choices, and it's good to remember that virtually no implementation details are too small to be noticed by players.

The Bounding Box and Collisions project introduces the key element of detection to your design arsenal. Detection allows you to begin including more robust cause-and-effect mechanics that form the basis for many game interactions. Chapter 6 discusses the trade-offs of choosing between the less precise but more performant bounding box collision detection method versus the precise but resource-intensive per-pixel detection method. There are situations where the bounding-box approach is sufficient, but if players perceive collisions to be arbitrary because the bounding boxes are too removed from the actual visual objects, it can negatively impact the sense of presence. Detection and collision are even more powerful design tools when coupled with the result from the Per Pixel Collisions project. Although the dye pack in this example was used to indicate the first point of collision, you can imagine building interesting casual chains around a new object being produced as the result of two objects colliding (for example, player pursues object, player collides with object, object "drops" a new object that enables the player to do something they couldn't do before). Game objects that move around the game screen will typically be animated, of course, so the Sprite Pixel Collisions project reviewed how to implement collision detection when the object boundaries aren't stationary.

With the addition of the techniques in Chapter 6, you now have a critical mass of behaviors that can be combined to create truly interesting game mechanics relevant to many genres from action games to puzzlers. Of course, game mechanic behaviors are only one of the nine elements of game design and typically aren't sufficient on their own to create a magical gameplay experience: the setting, visual style, meta game elements, and the like all have something important to contribute. The good news is that creating a memorable game experience need not be as elaborate as you often believe. Great games continue being produced based on relatively basic combinations of the behaviors and techniques covered in Chapters 1–6. The games that often shine the brightest aren't always the most complex, but rather they're often the games where every aspect of each of the nine elements of design is intentional and working together in harmony. If you give the appropriate attention and focus to all aspects of the game design, you're on a great track to produce something great whether you're working on your own or you're part of a large team.