Defining Behaviors and Detecting Collisions

Introduction

By this point, your game engine is capable of implementing games in convenient coordinate systems as well as 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 results of this shortcoming in the `init()` and `update()` functions of the `MyGame` objects in all the previous projects: the `init()` 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 encapsulate and hide the specifics of its initialization and behaviors. There are two main advantages to this approach. First, the `init()` 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 SimpleShader class 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 the 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 objects in the game. In general, resolving relative positions of all objects in a 2D world is nontrivial. Fortunately, typical video games require the knowledge of only those objects that are in close proximity to each other or are about to collide or have collided.

An efficient but somewhat crude approximation to detect collision is to compute the bounds of an object and approximate object collisions based on colliding bounding boxes. In the simplest cases, bounding boxes are rectangular boxes with edges that are aligned with the x/y axes. These are referred to as axis-aligned bounding boxes or AABBs. Because of the axis alignments, it is computationally efficient to detect when two AABBs overlap or when collision is about to occur.

Many 2D game engines can also 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 nontransparent 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 `init()` and `update()` functions of a game level and to facilitate reuse. This section introduces the simple `GameObject` class to illustrate how the cleaner and uncluttered `init()` 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.

Lab 6.1
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. This project leads you to create the infrastructure to support the many minions while keeping the logic in the `MyGame` level simple.

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 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 To introduce the `GameObjectSet` class demonstrating support for a set of homogenous objects with an identical interface

You can obtain the following external resource file from your instructor that you can put into the assets folder: `minion_sprite.png`; you'll also find the `fonts` folder that contains the default system fonts.

Note The `minion_sprite.png` file for this project is not the same image as for previous projects.

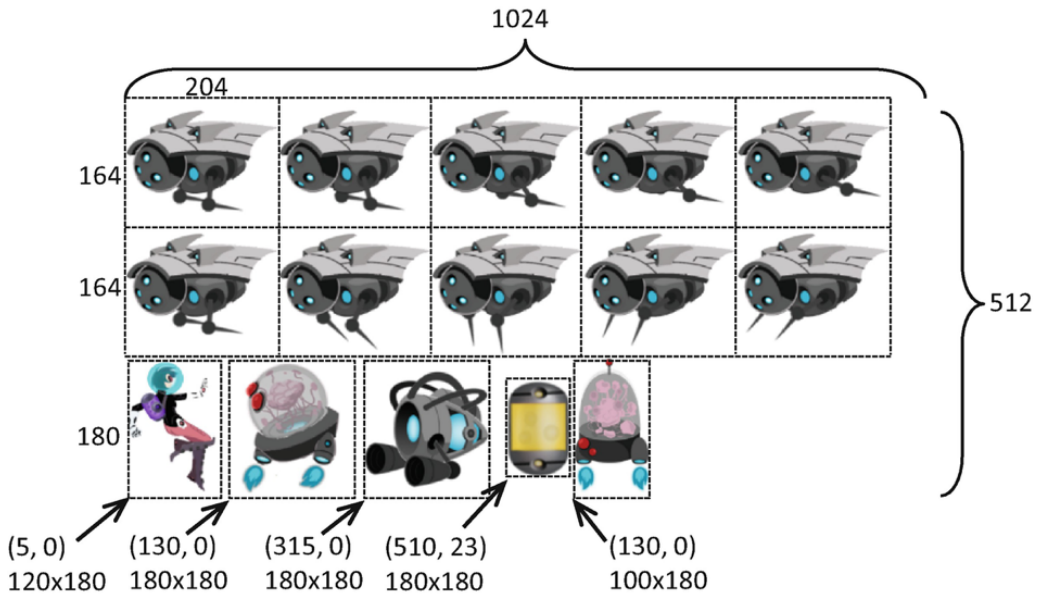

Figure 6.1. The minion_sprite.png file

Define the GameObject Class

The goal is to define a logical abstraction to encapsulate all relevant behavioral characteristics of a typical object in a game 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 only demonstrates the potential of the GameObject class with minimal behaviors defined.

1. Add a new folder src/engine/game_objects for storing GameObject-related files.

2. Create a new file in this folder, name it game_object.js, and add the following code:

```
class GameObject {
  constructor(renderable) {
      this.mRenderComponent = renderable;
  }
  getXform() { return this.mRenderComponent.getXform(); }
  getRenderable() { return this.mRenderComponent; }
  update() {  }
  draw(aCamera) {
      this.mRenderComponent.draw(aCamera);
  }
}
export default GameObject;
```

With the assessors to the Renderable and Transform 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/game_objects folder and name it game_object_set.js. Define the GameObjectSet class and the constructor to initialize an array for holding GameObject instances.

```
class GameObjectSet {
  constructor() {
      this.mSet = [];
  }
... implementation to follow ...
}
export default GameObjectSet;
```

2. Define functions for managing the set membership:

```
size() { return this.mSet.length; }
getObjectAt(index) { return this.mSet[index]; }
addToSet(obj) { this.mSet.push(obj); }
removeFromSet(obj) {
  let index = this.mSet.indexOf(obj);
  if (index > -1)
      this.mSet.splice(index, 1);
}
```

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

```
update() {
  let i;
  for (i = 0; i < this.mSet.length; i++) {
      this.mSet[i].update();
  }
}
draw(aCamera) {
  let i;
  for (i = 0; i < this.mSet.length; i++) {
      this.mSet[i].draw(aCamera);
  }
}
```

Export the Classes to the Client

The last step in integrating any new functionality into the engine involves modifying the engine access file, `index.js`. Edit `index.js` and add the following import and export statements to grant the client access to the `GameObject` and `GameObjectSet` classes:

```
... identical to previous code ...
// game objects
import GameObject from "./game_objects/game_object.js";
import GameObjectSet from "./game_objects/game_object_set.js";
... identical to previous code ...
export default {
  ... identical to previous code ...
  // Game Objects
  GameObject, GameObjectSet,
  ... identical to previous code ...
}
```

Note This process of import/export classes via the engine access file, `index.js`, must be repeated for every newly defined functionality. Henceforth, only a reminder will be provided and the straightforward code change will not be shown again.

Test the GameObject and GameObjectSet

The goals of this project 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 detailed implementation of these objects, follow good source code organization practice and create a new folder `src/my_game/objects` for storing the new object types.

The DyePack GameObject

The DyePack class derives from the `GameObject` class to demonstrate the most basic example of a `GameObject`: an object that has no behavior and is simply drawn to the screen.

Create a new file in the `src/my_game/objects` folder and name it `dye_pack.js`. Import from the engine access file, `index.js`, to gain access to all of the game engine functionality. Define `DyePack` as a subclass of `GameObject` and implement the constructor as follows:

```
import engine from "../../engine/index.js";
class DyePack extends engine.GameObject {
  constructor(spriteTexture) {
      super(null);
      this.kRefWidth = 80;
      this.kRefHeight = 130;
      this.mRenderComponent =
                          new engine.SpriteRenderable(spriteTexture);
      this.mRenderComponent.setColor([1, 1, 1, 0.1]);
      this.mRenderComponent.getXform().setPosition(50, 33);
      this.mRenderComponent.getXform().setSize(
                          this.kRefWidth / 50, this.kRefHeight / 50);
      this.mRenderComponent.setElementPixelPositions(510,595,23,153);
  }
}
export default DyePack;
```

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

Note The need to import from the engine access file, `index.js`, is true for almost all client source code file and will not be repeated.

The Hero GameObject

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

1. Create a new file in the `src/my_game/objects` folder and name it `hero.js`. Define `Hero` as a subclass of `GameObject`, and implement the constructor to initialize the sprite UV values, size, and position. Make sure to export and share this class.

```
class Hero extends engine.GameObject {
  constructor(spriteTexture) {
      super(null);
      this.kDelta = 0.3;
      this.mRenderComponent =
                          new engine.SpriteRenderable(spriteTexture);
      this.mRenderComponent.setColor([1, 1, 1, 0]);
      this.mRenderComponent.getXform().setPosition(35, 50);
      this.mRenderComponent.getXform().setSize(9, 12);
      this.mRenderComponent.setElementPixelPositions(0, 120, 0, 180);

    ... implementation to follow ...
    }
}
export default Hero;
```

2. Add a function to support the update of this object by user keyboard control. The `Hero` object moves at a `kDelta` rate based on WASD input from the keyboard.

```
update() {
  // control by WASD
  let xform = this.getXform();
  if (engine.input.isKeyPressed(engine.input.keys.W)) {
      xform.incYPosBy(this.kDelta);
  }
  if (engine.input.isKeyPressed(engine.input.keys.S)) {
      xform.incYPosBy(-this.kDelta);
  }
  if (engine.input.isKeyPressed(engine.input.keys.A)) {
      xform.incXPosBy(-this.kDelta);
  }
  if (engine.input.isKeyPressed(engine.input.keys.D)) {
          xform.incXPosBy(this.kDelta);
  }
}
```

The Minion GameObject

The Minion class demonstrates that simple autonomous behavior can also be hidden:

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

```
class Minion extends engine.GameObject {
  constructor(spriteTexture, atY) {
      super(null);
      this.kDelta = 0.2;
      this.mRenderComponent =
                    new engine.SpriteAnimateRenderable(spriteTexture);
      this.mRenderComponent.setColor([1, 1, 1, 0]);
      this.mRenderComponent.getXform().setPosition(
                                            Math.random() * 100, atY);
      this.mRenderComponent.getXform().setSize(12, 9.6);
      // first element pixel position: top-left 512 is top of image
      // 0 is left of the image
      this.mRenderComponent.setSpriteSequence(512, 0,
          204, 164,   // width x height in pixels
          5,          // number of elements in this sequence
          0);         // horizontal padding in between
      this.mRenderComponent.setAnimationType(
                                         engine.eAnimationType.eSwing);
      this.mRenderComponent.setAnimationSpeed(15);
      // show each element for mAnimSpeed updates
  }
  ... implementation to follow ...
}
export default Minion;
```

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

```
update() {
  // remember to update this.mRenderComponent's animation
  this.mRenderComponent.updateAnimation();
  // move towards the left and wraps
  let 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());
  }
}
```

The MyGame Scene

As in all cases, the `MyGame` level is implemented in the `my_game.js` file. With the three specific `GameObject` subclasses defined, follow these steps:

1. In addition to the engine access file, `index.js`, in order to gain access to the newly defined objects, the corresponding source code must be imported:

```
import engine from "../engine/index.js";
// user stuff
import DyePack from "./objects/dye_pack.js";
import Minion from "./objects/minion.js";
import Hero from "./objects/hero.js";
```

Note As is the case for other import/export statements, unless there are other specific reasons, this reminder will not be shown again.

2. Edit the `init()` function with the following code:

```
init() {
  // Step A: set up the cameras and set the background to gray
  this.mCamera = new engine.Camera(
    vec2.fromValues(50, 37.5),    // position of the camera
    100,                          // width of camera
    [0, 0, 640, 480]              // viewport (orgX, orgY, width, height)
  );
  this.mCamera.setBackgroundColor([0.8, 0.8, 0.8, 1]);

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

  // Step C: A set of Minions
  this.mMinionset = new engine.GameObjectSet();
  let 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 engine.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 similar to previous projects except for the dimensions of the viewport. 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, an instance of the `GameObjectSet` class. Notice that the `init()` function is free from the clustering of setting each object's textures, geometries, and so on.

3. The constructor and the `load()`, `unload()`, and `draw()` functions are similar as in previous projects.

TASK 6.1A complete the implementation for `constructor()`, `load()`, `unload()` and `draw()` functions.

4. Edit the `update()` function to update the game state :

```
update() {
  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.

Observation

You can now run the project and notice that the slightly more complex movements of six minions are accomplished with much cleaner `init()` and `update()` functions. The `init()` function consists of only logic and controls for 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 update()` function can focus on the details of the level. Note that the structure of this function clearly shows that the three objects are updated independently and do not interact with each other.

Note Throughout this book, in almost all cases, `MyGame` classes are designed to showcase the engine functionality. As a result, the source code organization in most `MyGame` classes may not represent the best practices for implementing games.

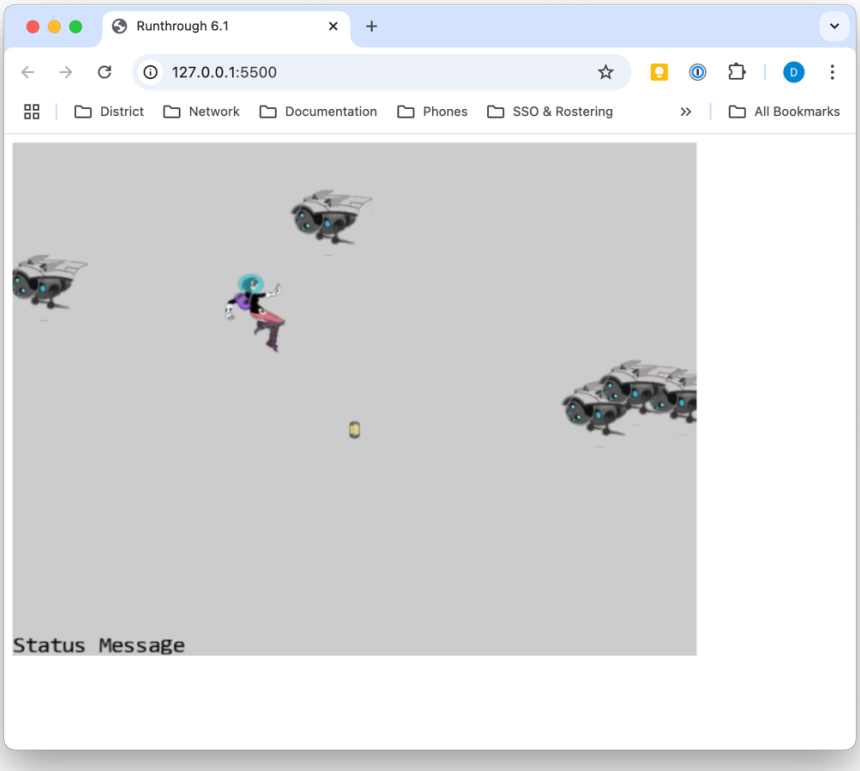

Figure 6-2 Running the Game Objects project

Creating a Chase Behavior

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 front direction and speed of the chaser such that 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 player is unable to avoid being hit, the game becomes impossibly difficult. 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 games, 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 games.

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.

Note For an introductory and comprehensive coverage of vectors, you can refer to www.storyofmathematics.com/vectors. For more detailed coverage of vector applications in games, you can refer to *Basic Math for Game Development with Unity 3D: A Beginner's Guide to Mathematical Foundations*, Apress, 2019.

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 the velocity or acceleration of an object. 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 Pa = (x_a, y_a) and Pb = (x_b, y_b), you can define the vector from P_a to P_b or $\overrightarrow{V_{ab}}$ as $P_b - P_a$. You can see this represented in the following equations and Figure 6-3:

- $P_a = (x_a, y_a)$
- $P_b = (x_b, y_b)$
- $\overrightarrow{V_{ab}} = P_b - P_a = (x_b - x_a, y_b - y_b)$

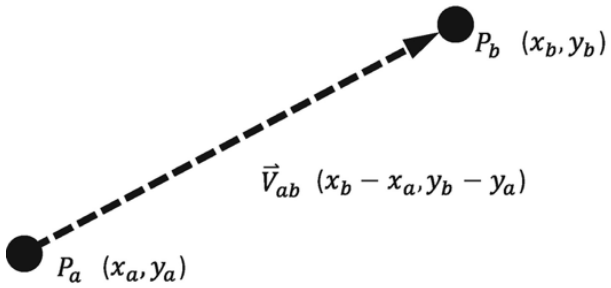

Figure 6-3 vector defined by two points

Now that you have a vector $\overrightarrow{V_{ab}}$, you can easily determine its length (or size) and direction. A vector's length is equal to the distance between the two points that created it. In this example, the length of or $\overrightarrow{V_{ab}}$ is equal to the distance between P_a and P_b, while the direction or $\overrightarrow{V_{ab}}$ 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 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 or $\overrightarrow{V_{ab}}$ can all be implemented as instances of the `vec2` object. However, $\overrightarrow{V_{ab}}$ 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, as shown in Figure 6-4. Notice that the mathematical symbol for a regular vector $\overrightarrow{V_a}$ and for a normalized vector is $\widehat{V_a}$:

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

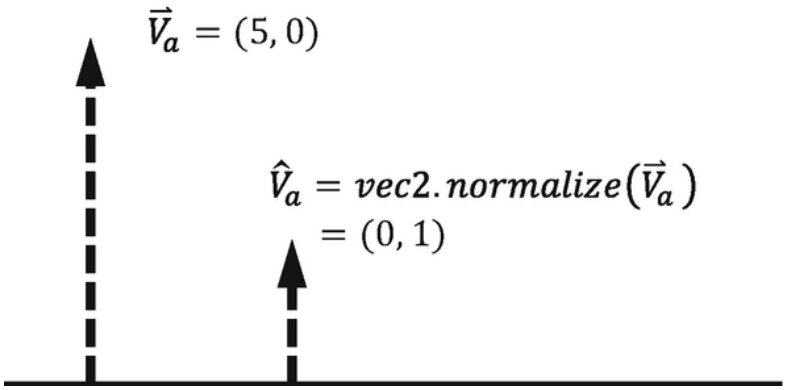

Figure 6-4 vector being normalized

Vectors to a position can also be rotated. If, for example, the vector $\overrightarrow{V_a} = (x_v, y_v)$ represents the direction from the origin to the position (x_v, y_v) and you want to rotate it by θ, then, as illustrated in Figure 6-5 , you can use the following equations to derive x_r and y_r:

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

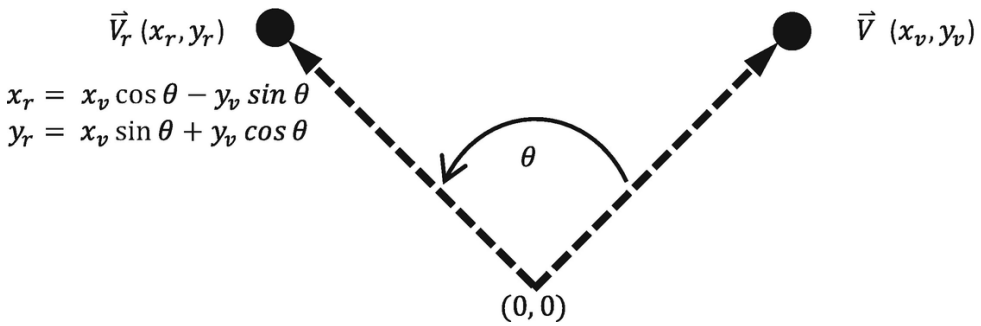

Figure 6-5 Vector from the origin to the position (x_v, y_v) being rotated by θ

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.

It is always 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 $\overrightarrow{V_a}$ and $\overrightarrow{V_{bc}}$ that are located at different positions but have the same direction and magnitude and thus are equal to each other. In contrast, the vector $\overrightarrow{V_d}$ is not the same because its direction and magnitude are different from the others.

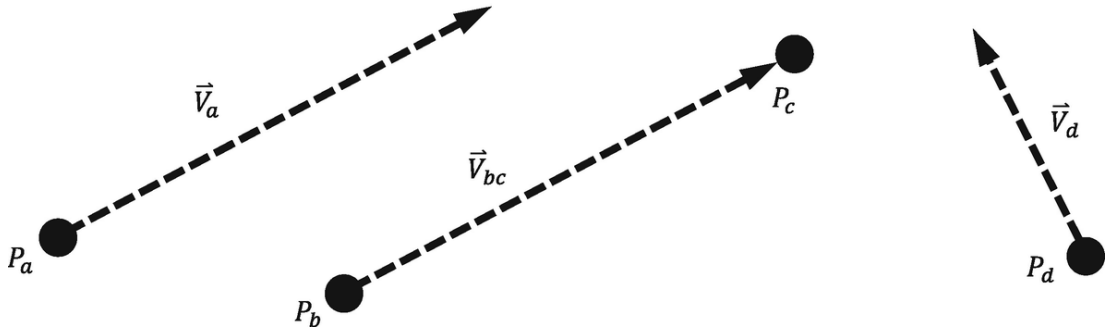

Figure 6-6 Three vectors represented in 2D space with two vectors equal to each other

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:

- $\overrightarrow{V_1} = (x_1, y_1)$
- $\overrightarrow{V_2} = (x_2, y_2)$

Then the following is true:

- $\overrightarrow{V_1} \cdot \overrightarrow{V_2} = \overrightarrow{V_2} \cdot \overrightarrow{V_1} = x_1x_2 + y_1y_2)$

Additionally, if both vectors $\overrightarrow{V_1}$ and $\overrightarrow{V_2}$ are normalized, then

- $\widehat{V_1} \cdot \widehat{V_2} = \cos\theta$

Figure 6-7 depicts an example of the $\overrightarrow{V_1}$ and $\overrightarrow{V_2}$ vectors with an angle θ in between them. It is also important to recognize that if $\overrightarrow{V_1} \cdot \overrightarrow{V_2} = 0$, then the two vectors are perpendicular.

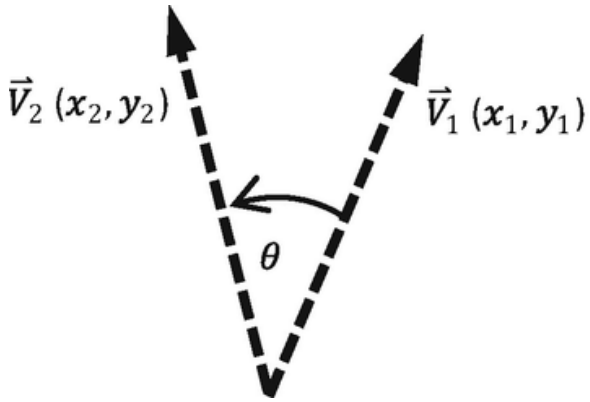

Figure 6-7 The angle between two vectors, which can be found through the dot product

Note If you need to review or refresh the concept of a dot product, please refer to 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 result of the cross product is 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 x/y plane results in a vector that lies in the third dimension or along the z axis. However, the resulting vector in the third dimension carries crucial information. For example, the direction of this vector in the third dimension can be used to determine whether the game object needs to rotate in the clockwise or counterclockwise direction. Take a closer look at the following:

- $\overrightarrow{V_1} = (x_1, y_1)$
- $\overrightarrow{V_2} = (x_2, y_2)$

Given the previous, the following is true:

- $\overrightarrow{V_3} = \overrightarrow{V_1} \times \overrightarrow{V_2}$ is a vecotr perpendicular to both $\overrightarrow{V_1}$ and $\overrightarrow{V_2}$.

Additionally, you know that the cross product of two vectors on the x/y plane results in a vector in the z direction. When $\overrightarrow{V_1} \times \overrightarrow{V_2} > 0$, you know that $\overrightarrow{V_1}$ is in the clockwise direction from $\overrightarrow{V_2}$ similarly, when $\overrightarrow{V_1} \times \overrightarrow{V_2} < 0$, you know that $\overrightarrow{V_1}$ is in the counterclockwise direction. Figure 6-8 should help clarify this concept.

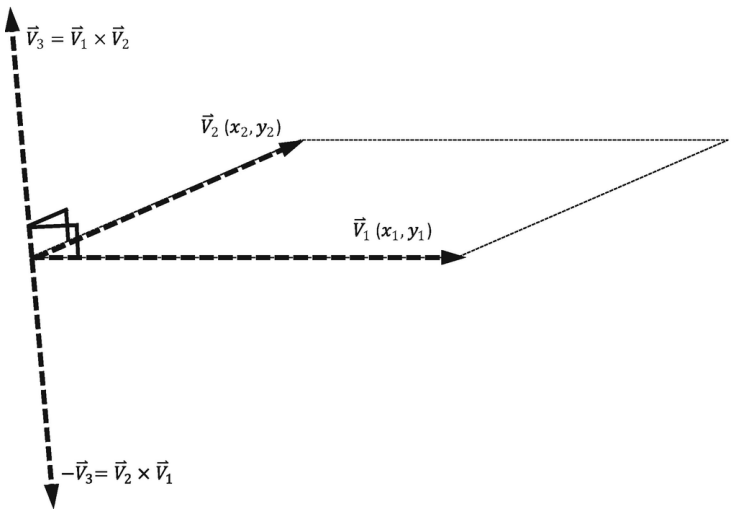

Figure 6-8 The cross product of two vectors

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

Lab 6.2
The Front and Chase Project

This project implements more interesting and sophisticated behaviors based on the vector concepts that have been reviewed. Instead of constant and aimless motions, you will experience the process of defining and varying the front direction of an object and guiding an object to chase after another object in the scene.

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

```
constructor(renderable) {
  this.mRenderComponent = renderable;
  this.mVisible = true;
  this.mCurrentFrontDir = vec2.fromValues(0, 1); // front direction
  this.mSpeed = 0;
}
```

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

```
getXform() { return this.mRenderComponent.getXform(); }
setVisibility(f) { this.mVisible = f; }
isVisible() { return this.mVisible; }
setSpeed(s) { this.mSpeed = s; }
getSpeed() { return this.mSpeed; }
incSpeedBy(delta) { this.mSpeed += delta; }
setCurrentFrontDir(f) { vec2.normalize(this.mCurrentFrontDir, f); }
getCurrentFrontDir() { return this.mCurrentFrontDir; }
getRenderable() { return this.mRenderComponent; }
```

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

```
rotateObjPointTo(p, rate) {
  // Step A: determine if reached the destination position p
  let dir = [];
  vec2.sub(dir, p, this.getXform().getPosition());
  let 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
  let fdir = this.getCurrentFrontDir();
  let cosTheta = vec2.dot(dir, fdir);
  if (cosTheta > 0.999999) { // almost exactly the same direction
      return;
  }

  // Step C: clamp the cosTheta 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
```

```
    let dir3d = vec3.fromValues(dir[0], dir[1], 0);
    let f3d = vec3.fromValues(fdir[0], fdir[1], 0);
    let r3d = [];
    vec3.cross(r3d, f3d, dir3d);
    let 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 the destination position `p` at a rate specified by the parameter `rate`. Here are the details of each operation:

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, as illustrated in Figure 6-9, computes the dot product to determine the angle θ 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 (cosθ is almost 1 or θ almost zero), the function returns.

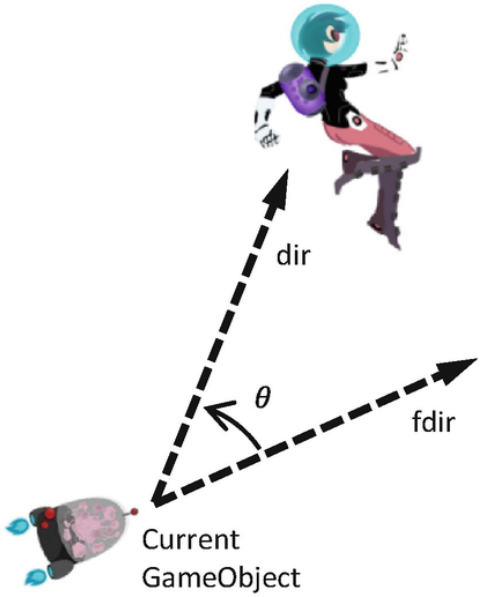

Figure 6-9 GameObject (Brain) chasing a target (Hero)

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 uses 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. It is important to recognize the two separate object rotation controls. The `Transform` controls the rotation of what is being drawn, and `mCurrentFrontDir` controls the direction of travel. In this case, the two are synchronized and thus must be updated with the new value simultaneously.

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

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

5. Modify the `draw()` function to draw the object based on the visibility setting:

```
draw(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 to demonstrate traveling along a predefined front direction and to direct the `Brain` to chase after the `Hero` to demonstrate 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/my_game/objects` folder and name it `brain.js`. Define `Brain` as a subclass of `GameObject`, and implement the constructor to initialize the appearance and behavior parameters.

```
class Brain extends engine.GameObject {
  constructor(spriteTexture) {
      super(null);
      this.kDeltaDegree = 1;
      this.kDeltaRad = Math.PI * this.kDeltaDegree / 180;
      this.kDeltaSpeed = 0.01;
      this.mRenderComponent =
                            new engine.SpriteRenderable(spriteTexture);
      this.mRenderComponent.setColor([1, 1, 1, 0]);
      this.mRenderComponent.getXform().setPosition(50, 10);
      this.mRenderComponent.getXform().setSize(3, 5.4);
      this.mRenderComponent.setElementPixelPositions(600, 700, 0, 180);
      this.setSpeed(0.05);
  }
  ... implementation to follow ...
}
export default Brain;
```

2. Override the `update()` function to support the 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.

```
update() {
  super.update();
  let xf = this.getXform();
  let fdir = this.getCurrentFrontDir();
  if (engine.input.isKeyPressed(engine.input.keys.Left)) {
      xf.incRotationByDegree(this.kDeltaDegree);
      vec2.rotate(fdir, fdir, this.kDeltaRad);
  }
  if (engine.input.isKeyPressed(engine.input.keys.Right)) {
      xf.incRotationByRad(-this.kDeltaRad);
      vec2.rotate(fdir, fdir, -this.kDeltaRad);
  }
  if (engine.input.isKeyClicked(engine.input.keys.Up)) {
      this.incSpeedBy(this.kDeltaSpeed);
  }
  if (engine.input.isKeyClicked(engine.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 `my_game.js` is similar to previous projects. For this reason, only the details of the `update()` function are shown.

```
update() {
  let msg = "Brain [H:keys J:imm K:gradual]: ";
  let rate = 1;
  this.mHero.update();
  switch (this.mMode) {
      case 'H':
          this.mBrain.update();  // player steers with arrow keys
          break;
      case 'K':
          rate = 0.02;    // gradual rate
          // In gradual mode, the following should also be executed
      case 'J':
          this.mBrain.rotateObjPointTo(
              this.mHero.getXform().getPosition(), rate);
          // the default GameObject: only move forward
          engine.GameObject.prototype.update.call(this.mBrain);
          break;
      }
  if (engine.input.isKeyClicked(engine.input.keys.H)) {
      this.mMode = 'H';
  }
  if (engine.input.isKeyClicked(engine.input.keys.J)) {
      this.mMode = 'J';
  }
  if (engine.input.isKeyClicked(engine.input.keys.K)) {
      this.mMode = 'K';
  }
  this.mMsg.setText(msg + this.mMode);
}
```

TASK: Complete the implementation of `my_game.js` using the previous project and the following `update()` method as references to identify elements for the `constructor()`, `init()`, `load()`, `unload()` and `draw()`.

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's `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.

Note that in the cases of `J` and `K` modes, in order to bypass the user control logic after the `rotateObjPointTo()`, the `update()` function being called is the one defined by the `GameObject` and not by the Brain.

Note The JavaScript syntax, `ClassName.prototype.FunctionName.call(anObj)`, calls `FunctionName` defined by `ClassName`, where `anObj` is a subclass of `ClassName`.

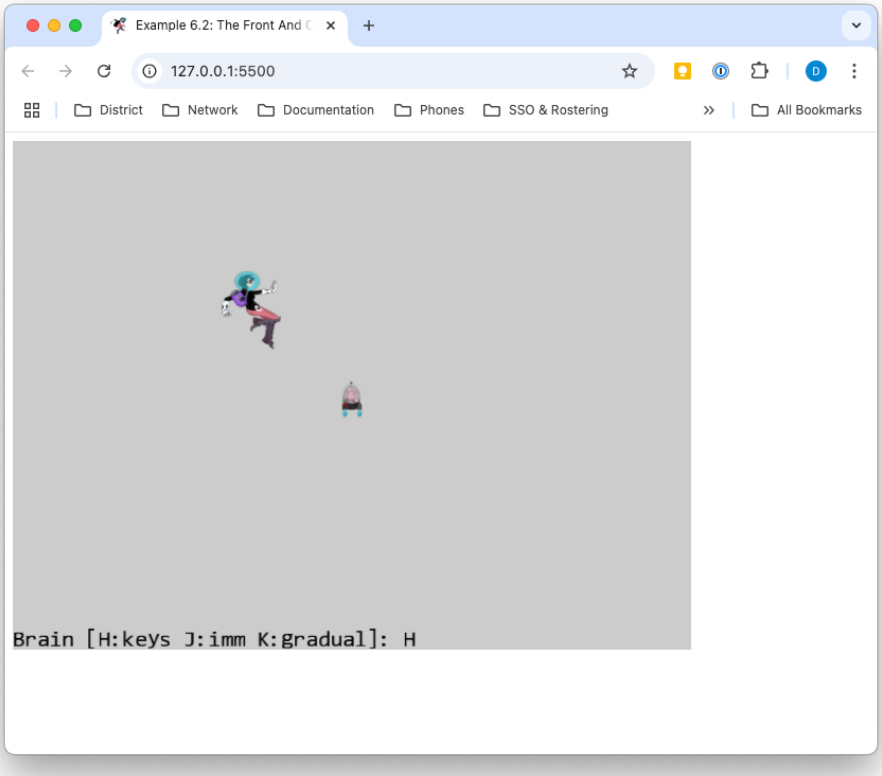

Figure 6-10 GameObject (Brain) chasing a target (Hero)

Observation

You can now try running the project. Initially, the `Brain` object is under the user's control. You can use the left- and right-arrow keys to change the front direction of the `Brain` object and experience steering 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 gradually turns to move toward the `Hero` object. Feel free to change the values of the rate variable or modify the control value of the `Brain` object. For example, change the `kDeltaRad` or `kDeltaSpeed` to experiment with different settings for the behavior.

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 is missing the critical ability to detect that it has collided with the `Hero` object, and as a result, it never stops moving. This section describes axis-aligned bounding boxes (AABBs), one of the most straightforward tools for approximating object collisions, and demonstrates the implementation of collision detection based on AABB.

Axis-Aligned Bounding Box (AABB)

An AABB is an x/y axis–aligned rectangular box that bounds a given object. The term *x/y axis aligned* refers to the fact that the four sides of an AABB 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 `Hero` object by the lower-left corner (`mLL`), width, and height. This is a fairly common way to represent an AABB because it uses only one position and two floating-point numbers to represent the dimensions.

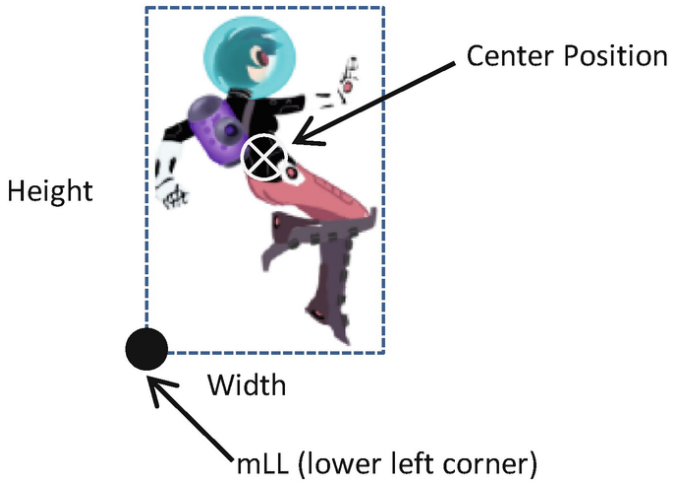

Figure 6-11 The lower-left corner and size of the bounds for an object

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 by the game developer. An AABB 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.

Note In this book, AABB and "bounding box" are used interchangeably.

Lab 6.3
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.

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 the 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 Class

Define a `BoundingBox` class to represent the bounds of a rectangular area:

1. Create a new file in the `src/engine` folder; name it `bounding_box.js`. First, define an enumerated data type with values that identify the colliding sides of a bounding box.

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

Notice that each enumerated value 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 = 1 | 4 = 5`.

2. Now, define the `BoundingBox` class and the constructor with instance variables to represent a bound, as illustrated in Figure 6-10. Notice that the `eBoundCollideStatus` must also be exported such that the rest of the engine, including the client, can also have access.

```
class BoundingBox {
  constructor(centerPos, w, h) {
      this.mLL = vec2.fromValues(0, 0);
      this.setBounds(centerPos, w, h);
  }
  ... implementation to follow ...
}
export {eBoundCollideStatus}
export default BoundingBox;
```

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

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

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

```
containsPoint(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:

```
intersectsBound(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:

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

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

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

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

TASK 6.3A Remember to update the engine access file, `index.js`, to forward the newly defined functionality to the client. This task requires two import statements and corresponding exports. One for **BoundingBox** and one for the `eBoundCollideStatus` constants.

Use the BoundingBox in the Engine

The newly defined functionality will be used to detect collisions between objects and between objects and the WC bounds. In order to accomplish this, the `GameObject` and `Camera` classes must be modified.

1. Edit `game_object.js` to import the newly defined functionality and modify the `GameObject` class; implement the `getBBox()` function to return the bounding box of the unrotated `Renderable` object:

```
import BoundingBox from "../bounding_box.js";
class GameObject {
  ... identical to previous code ...
  getBBox() {
      let xform = this.getXform();
      let b = new BoundingBox(
                          xform.getPosition(),
                          xform.getWidth(),
                          xform.getHeight());
      return b;
  }
  ... identical to previous code ...
}
```

2. Edit `camera.js` to import from bounding box, and modify the `Camera` class to compute the collision status between the bounds of a Transform object (typically defined in a `Renderable` object) and that of the WC window:

```
import BoundingBox from "./bounding_box.js";
class Camera {
  ... identical to previous code ...
  collideWCBound(aXform, zone) {
      let bbox = new BoundingBox(
                          aXform.getPosition(),
                          aXform.getWidth(),
                          aXform.getHeight());
      let w = zone * this.getWCWidth();
      let h = zone * this.getWCHeight();
      let 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 6-12 shows how the camera collides with an object.

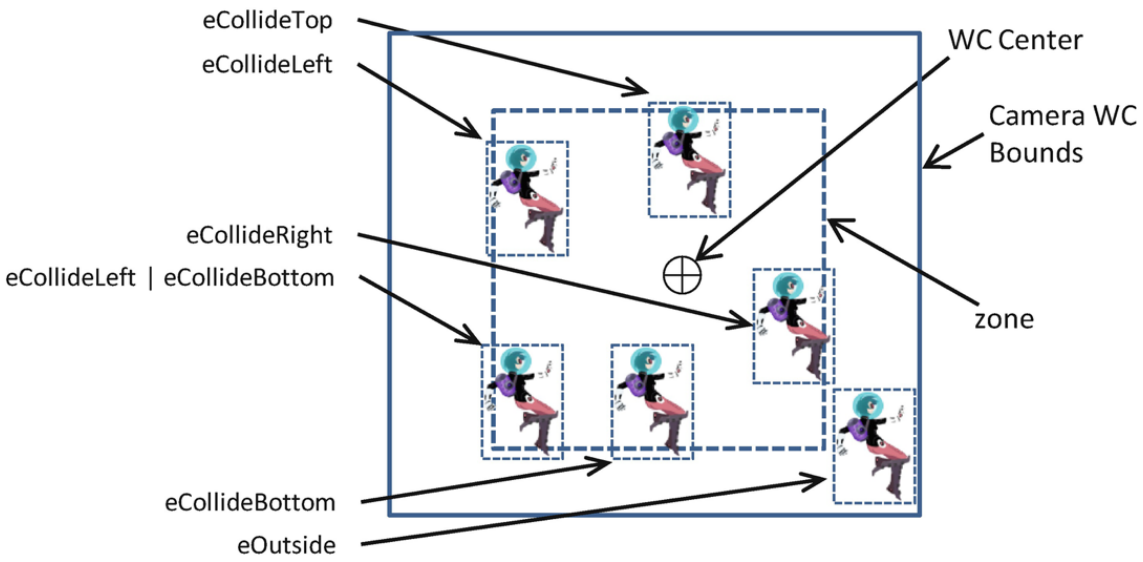

Figure 6-12 Camera WC bounds colliding with the bounds defining a Transform object

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 `my_game.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.

TASK 6.3B: The `update()` function that follows is missing key elements and will result in errors. Complete the implementation for this method and the rest of `my_game.js`.

```
update() {
  ... identical to previous code ...
  switch (this.mMode) {
      case 'H':
          this.mBrain.update();  // player steers with arrow keys
          break;
      case 'K':
          rate = 0.02;    // gradual rate
          // no break here on purpose
      case 'J':
          // stop the brain when it touches hero bound
          if (!hBbox.intersectsBound(bBbox)) {
              this.mBrain.rotateObjPointTo(
                      this.mHero.getXform().getPosition(), rate);
              // the default GameObject: only move forward
              engine.GameObject.prototype.update.call(this.mBrain);
          }
```

```
            break;
    }
    // Check for hero going outside 80% of the WC Window bound
    let status = this.mCamera.collideWCBound(this.mHero.getXform(), 0.8);
    ... identical to previous code ...
    this.mMsg.setText(msg + this.mMode + " [Hero bound=" + status + "]");
}
```

In the `switch` statement's `J` and `K` cases, the modification tests for bounding box collision between the `Brain` and `Hero` objects before invoking `Brain.rotateObjPointTo()` and `update()` to cause the chasing behavior. In this way, the `Brain` object will stop moving as soon as it touches the bound of the `Hero` object. In addition, the collision results between the Hero object and 80 percent of the camera WC window are computed and displayed.

Observation

You can now run the project and observe that the `Brain` object, when in autonomous mode (`J` or `K` keys), stops moving as soon as it touches the `Hero` object. When you move the `Hero` object around, observe the `Hero` bound output message begins to echo 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.

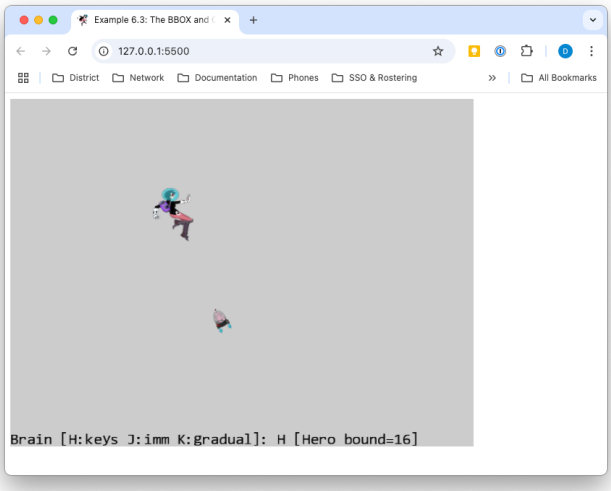

Figure 6-13 Collision Detection

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.

1. The BoundingBox object introduced in the previous example does not account for rotation. This is a well-known limitation for AABB: although the approach is computationally efficient, it does not support rotated objects.
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.

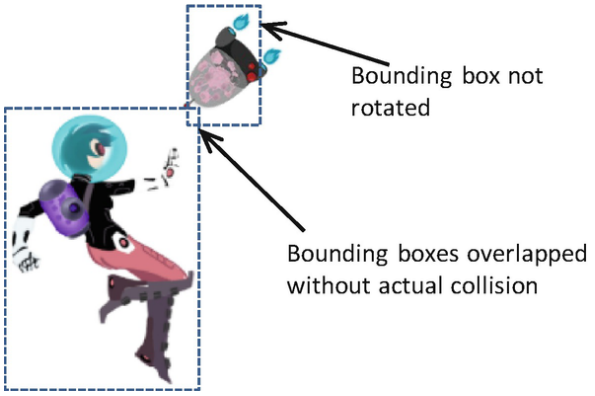

Figure 6-14 Limitation with bounding box–based collision

In the next project, you will implement per-pixel-accurate collision detection, pixel-accurate collision detection, or per-pixel collision detection, to detect the overlapping of nontransparent pixels of two colliding objects. However, keep in mind that this is not an end-all solution. While the per-pixel 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.

Lab 6-4
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.

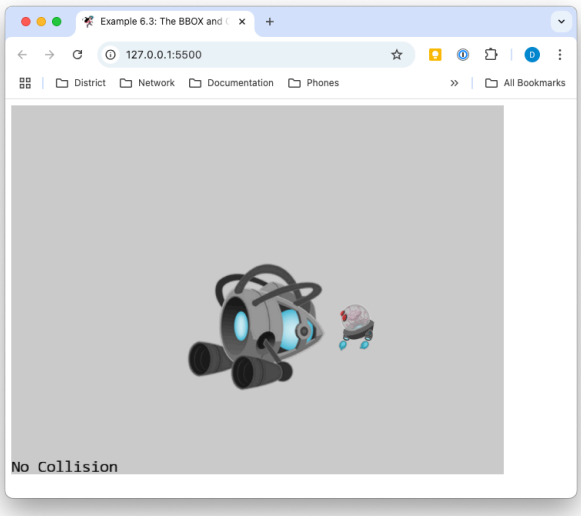

Figure 6-15 No collision

In this project, you will be able to use WASD keys to move the large `Portal` object and the arrow keys to move the small `Brain` object.

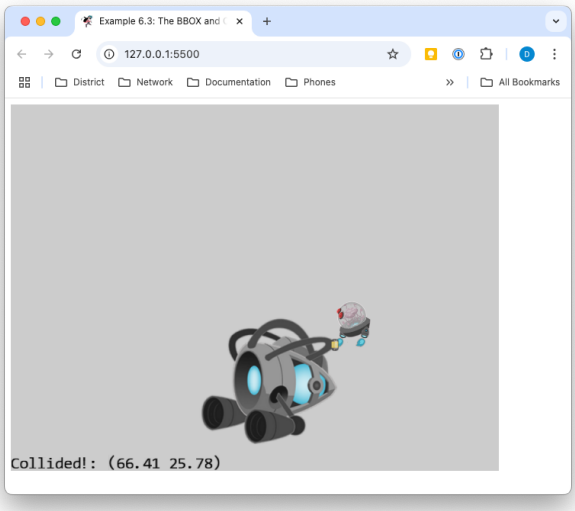

Figure 6-16 Collision Detected

When a collision occurs, a small yellow `DyePack` will appear at the collision point as shown in figure 6-16.

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

Note A "transparent" pixel is one you can see through completely and, in the case of this engine, has an alpha value of 0. A "nontransparent" pixel has a greater than 0 alpha value, or the pixel does not completely block what is behind it; it may or may not occlude. An "opaque" pixel occludes what is behind it, is "nontransparent," and has an alpha value of 1. For example, notice that you can "see through" the top region of the Portal object. These pixels are nontransparent but not opaque and should cause a collision when an overlap occurs based on the parameters defined by the project.

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 6-17. Notice how the bounds of the objects overlap, yet none of the nontransparent colored pixels are touching.

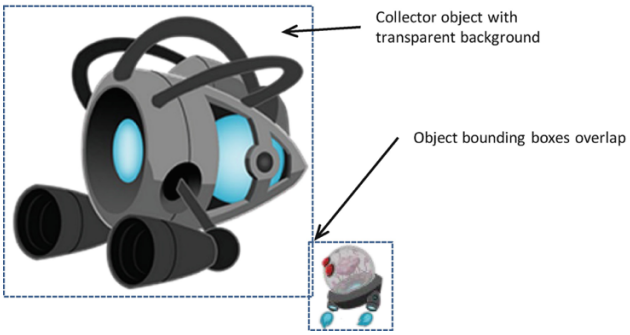

Figure 6-17 Overlapping bounding boxes without actual collision

The next case is to check whether the nontransparent pixels of the textures overlap. Take a look at Figure 6-18. Nontransparent pixels from the textures of the Collector and Portal objects are clearly in contact with one another.

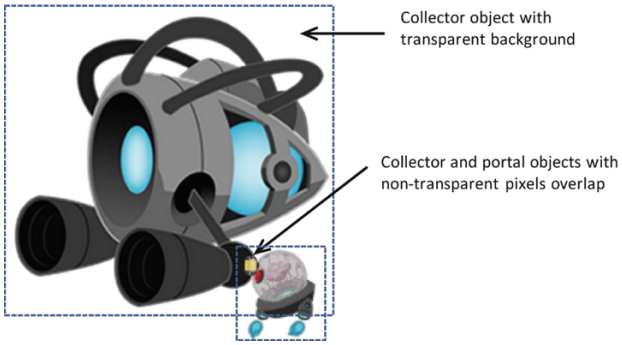

Figure 6-18 Pixel collision occurring between the large texture and the small texture

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 two collide then
    For each Pixel-A in Image-A
        If Pixel-A is not completely transparent
            pixelCameraSpace = Pixel-A position in camera space
            Transform pixelCameraSpace to Image-B space
            Read Pixel-B from Image-B
            If Pixel-B is not 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.

Notice that in the algorithm Image-A and Image-B are exchangeable. That is, when testing for collision between two images, it does not matter which image is Image-A or Image-B. The collision result will be the same. Either the two images do overlap, or they do not. Additionally, pay attention to the runtime of this algorithm. Each pixel within Image-A must be processed; thus, the runtime is O(N), where N is the number of pixels in Image-A or Image-A's resolution. For this reason, for performance reason, it is important to choose the smaller of the two images (the `Portal` minion in this case) as Image-A.

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 can quickly bog down performance. You are now ready to examine the implementation of per-pixel-accurate collision.

Modify Texture to Load a Texture as an Array of Colors

Recall that the Texture component reads image files from the server file system, loads the images to the GPU memory, and processes the images into WebGL textures. In this way, texture images are stored on the GPU and are not accessible by the game engine which is running on the CPU. To support per-pixel collision detection, the color information must be retrieved from the GPU and stored in the CPU. The Texture component can be modified to support this requirement.

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

```
class TextureInfo {
  constructor(w, h, id) {
      this.mWidth = w;
      this.mHeight = h;
      this.mGLTexID = id;
      this.mColorArray = null;
  }
}
```

2. Define and export a function to retrieve the color array from the GPU memory:

```
function getColorArray(textureName) {
  let gl = glSys.get();
  let texInfo = get(textureName);
  if (texInfo.mColorArray === null) {
      // create framebuffer bind to texture and read the color content
      let 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) {
          let 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 {
          throw new Error("...");
          return null;
      }
      gl.bindFramebuffer(gl.FRAMEBUFFER, null);
      gl.deleteFramebuffer(fb);
  }
  return texInfo.mColorArray;
}

export {has, get, load, unload,
```

```
    TextureInfo,
    activate, deactivate,
    getColorArray
  }
```

The `getColorArray()` function creates a WebGL `FRAMEBUFFER`, fills the buffer with the desired texture, and retrieves the buffer content into the CPU memory referenced by `texInfo.mColorArray`.

Modify TextureRenderable to Support Per-Pixel Collision

The `TextureRenderable` is the most appropriate class for implementing the per-pixel collision functionality. This is because `TextureRenderable` 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.

As the functionality of the `TextureRenderable` class increases, so will the complexity and size of the implementation source code. For readability and expandability, it is important to maintain the size of source code files. An effective approach is to separate the source code of a class into multiple files according to their functionality.

Organize the Source Code

In the following steps, the `TextureRenderable` class will be separated into three source code files: `texture_renderable_main.js` for implementing the basic functionality from previous projects, `texture_renderable_pixel_collision.js` for implementing the newly introduced per-pixel-accurate collision, and `texture_renderable.js` for serving as the class access point.

1. Rename `texture_renderable.js` to `texture_renderable_main.js`. This file defines the basic functionality of the `TextureRenderable` class.

2. Create a new file in `src/engine/renderables` and name it `texture_renderable_pixel_collision.js`. This file will be used to extend the `TextureRenderable` class functionality in supporting per-pixel-accurate collision. Add in the following code to import from the `Texture` module and the basic `TextureRenderable` class, and re-export the `TextureRenderable` class. For now, this file does not serve any purpose; you will add in the appropriate extending functions in the following subsection.

```
"use strict";
import TextureRenderable from "./texture_renderable_main.js";
import * as texture from "../resources/texture.js";
export default TextureRenderable;
```

3. Create a new `texture_renderable.js` file to serve as the `TextureRenderable` access point by adding the following code:

```
"use strict";
import TextureRenderable from "./texture_renderable_pixel_collision.js";
export default TextureRenderable;
```

With this structure, the `texture_renderable_main.js` file implements all the basic functionality and exports to `texture_renderable_pixel_collision.js`, which appends additional functionality to the `TextureRenderable` class. Finally, `texture_renderable.js` imports the extended functions from `texture_renderable_pixel_collision.js`. The users of the `TextureRenderable` class can simply import from `texture_renderable.js` and will have access to all of the defined functionality.

In this way, from the perspective of the game developer, `texture_renderable.js` serves as the access point to the `TextureRenderable` class and hides the details of the implementation source code structure. At the same time, from the perspective of you as the engine developer, complex implementations are separated into source code files with names indicating the content achieving readability of each individual file.

Define Access to the Texture Color Array

Recall that you began this project by first editing the Texture module to retrieve, from the GPU to the CPU, the color array that represents a texture. You must now edit `TextureRenderable` to gain access to this color array.

1. Edit the `texture_renderable_main.js` file, and modify the constructor to add instance variables to hold texture information, including a reference to the retrieved color array, for supporting per-pixel collision detection and for later subclass overrides:

```
class TextureRenderable extends Renderable {
  constructor(myTexture) {
      super();
      // Alpha of 0: switch off tinting of texture
      super.setColor([1, 1, 1, 0]);
      super._setShader(shaderResources.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.mElmWidthPixels = 0;
      this.mElmHeightPixels = 0;
      this.mElmLeftIndex = 0;
```

```
        this.mElmBottomIndex = 0;
        // texture for this object, cannot be a "null"
        this.setTexture(myTexture);
    }
```

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

```
setTexture(newTexture) {
    this.mTexture = newTexture;
    // these two instance variables are to cache texture information
    // for supporting per-pixel accurate collision
    this.mTextureInfo = texture.get(newTexture);
    this.mColorArray = null;
    // defined for one sprite element for subclass to override
    // For texture_renderable, one sprite element is the entire texture
    this.mElmWidthPixels = this.mTextureInfo.mWidth;
    this.mElmHeightPixels = this.mTextureInfo.mHeight;
    this.mElmLeftIndex = 0;
    this.mElmBottomIndex = 0;
}
```

Note that by default, the mColorArry is initialized to null. For CPU memory optimization, the color array is fetched from the GPU only for textures that participate in per-pixel collision. The `mElmWidthPixels` and `mElmHeightPixels` variables are the width and height of the texture. These variables are defined for later subclass overrides such that the algorithm can support the collision of sprite elements.

Implement Per-Pixel Collision

You are now ready to implement the per-pixel collision algorithm in the newly created `texture_renderable_pixel_collision.js` file.

1. Edit the `texture_renderable_pixel_collision.js` file, and define a new function for the `TextureRenderable` class to set the `mColorArray`:

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

Note JavaScript classes are implemented based on prototype chains. After class construction, instance methods can be accessed and defined via the prototype of the class or `aClass.prototype.method`. For more information on JavaScript classes and prototypes, please refer to https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain

2. Define a new 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 `mColorArray` is a 1D 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) {
  let x = i * this.mXform.getWidth() / this.mElmWidthPixels;
  let y = j * this.mXform.getHeight() / this.mElmHeightPixels;
  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]
  let delta = [];
  vec2.sub(delta, wcPos, this.mXform.getPosition());
  returnIndex[0] = this.mElmWidthPixels *
                   (delta[0] / this.mXform.getWidth());
  returnIndex[1] = this.mElmHeightPixels *
                   (delta[1] / this.mXform.getHeight());
  // recall that xForm.getPosition() returns center, yet
  // Texture origin is at lower-left corner!
  returnIndex[0] += this.mElmWidthPixels / 2;
  returnIndex[1] += this.mElmHeightPixels / 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) {
  let pixelTouch = false;
  let xIndex = 0, yIndex;
  let otherIndex = [0, 0];
  while ((!pixelTouch) && (xIndex < this.mElmWidthPixels)) {
      yIndex = 0;
      while ((!pixelTouch) && (yIndex < this.mElmHeightPixels)) {
          if (this._pixelAlphaValue(xIndex, yIndex) > 0) {
              this._indexToWCPosition(wcTouchPos, xIndex, yIndex);
              other._wcPositionToIndex(otherIndex, wcTouchPos);
              if ((otherIndex[0] >= 0) &&
                  (otherIndex[0] < other.mElmWidthPixels) &&
                  (otherIndex[1] >= 0) &&
                  (otherIndex[1] < other.mElmHeightPixels)) {
                  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 the many potentially colliding points.

Support Per-Pixel Collision in GameObject

Edit the `game_object.js` file to add the `pixelTouches()` function to the `GameObject` class:

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

To test per-pixel collision, we are going to create a new `GameObject` type to store information regarding the `TextureRenderable`.

1. Create a new file in `src/my_game/objects` called `texture_object.js`. Modify the file to include the following:

```
"use strict";

import engine from "../../engine/index.js";

class TextureObject extends engine.GameObject {
    constructor(texture, x, y, w, h) {
        super(null);
        this.kDelta = 0.2;

        this.mRenderComponent = new engine.TextureRenderable(texture);
        this.mRenderComponent.setColor([1, 1, 1, 0.1]);
        this.mRenderComponent.getXform().setPosition(x, y);
        this.mRenderComponent.getXform().setSize(w, h);
    }

    update(up, down, left, right) {
        let xform = this.getXform();
        if (engine.input.isKeyPressed(up)) {
            xform.incYPosBy(this.kDelta);
        }
        if (engine.input.isKeyPressed(down)) {
            xform.incYPosBy(-this.kDelta);
        }
        if (engine.input.isKeyPressed(left)) {
            xform.incXPosBy(-this.kDelta);
        }
        if (engine.input.isKeyPressed(right)) {
            xform.incXPosBy(this.kDelta);
        }
    }
}
```

```
export default TextureObject;
```

2. In `my_game.js`, modify the `init()` function:

```
init() {
  // Step A: set up the cameras
  this.mCamera = new engine.Camera(
    vec2.fromValues(50, 37.5), // position of the camera
          100,                 // width of camera
          [0, 0, 640, 480]     // viewport (orgX, orgY, width, height)
      );
  this.mCamera.setBackgroundColor([0.8, 0.8, 0.8, 1]);
  // sets the background to gray

  this.mDyePack = new DyePack(this.kMinionSprite);
  this.mDyePack.setVisibility(false);

  this.mCollector = new TextureObject(this.kMinionCollector,
          50, 30, 30, 30);
  this.mPortal = new TextureObject(this.kMinionPortal, 70, 30, 10, 10);

  this.mMsg = new engine.FontRenderable("Status Message");
  this.mMsg.setColor([0, 0, 0, 1]);
  this.mMsg.getXform().setPosition(1, 2);
  this.mMsg.setTextHeight(3);
}
```

3. Modify `update()` function as shown here:

```
update() {
  let msg = "No Collision";
  this.mCollector.update(engine.input.keys.W, engine.input.keys.S,
          engine.input.keys.A, engine.input.keys.D);
  this.mPortal.update(engine.input.keys.Up, engine.input.keys.Down,
          engine.input.keys.Left, engine.input.keys.Right);
  let h = [];
  // Portal's resolution is 1/16 x 1/16 that of Collector!
  // VERY EXPENSIVE!!
  // if (this.mCollector.pixelTouches(this.mPortal, h)) {
  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);
}
```

The testing of per-pixel collision is rather straightforward, involving three instances of `GameObject`: the large `Collector` minion, the small `Portal` minion, and the `DyePack`. The `Collector` and `Portal` minions are controlled by the arrow and WASD keys, respectively.

TASK 6.4A The details of the implementation of `MyGame` are similar to the previous projects and are to be completed by the student. You must adapt the `my_game.js` file using previous projects as examples to complete this project. Be sure to implement:

- identify and include and import statements
- constructor to identify the paths to the needed assets and initialize the variables (`mMsg`, `mCollector`, `mPortal`)
- `load` and `unload` to obtain and clear the assets
- `draw()` function

Observation

You can now test the collision accuracy by moving the two minions and intersecting them at different locations (e.g., 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 positions.

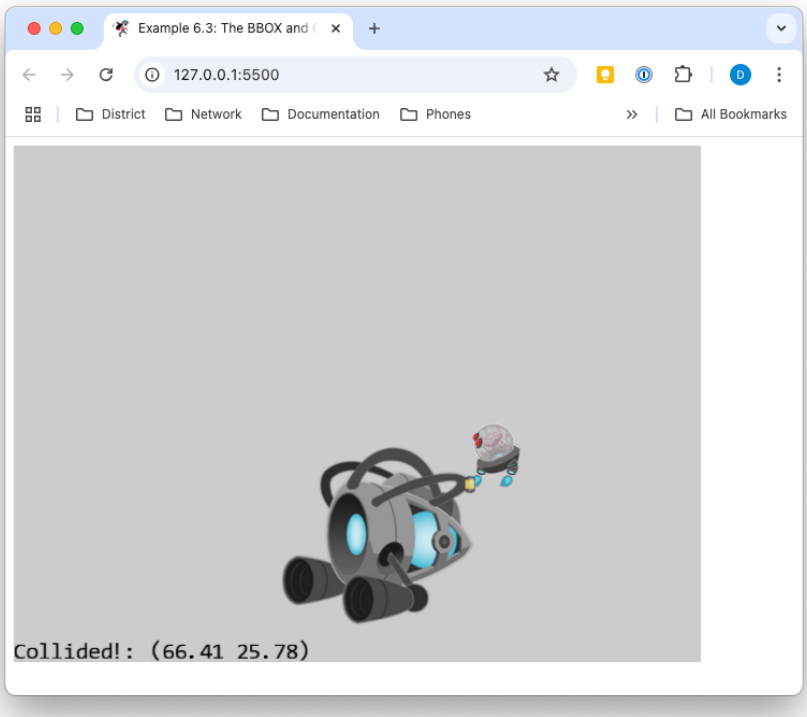

Figure 6-19 Collision Detected

TASK 6.4B 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 6-20 contains two normalized vectors, or the component

vectors, that can be used to decompose the vector $\vec{V} = (2,3)$: the normalized component vectors $\hat{\imath} = (1,0)$ and $\hat{\jmath} = (1,0)$ decompose the vector $\vec{V}$ into the components $2\hat{\imath}$ and $3\hat{\jmath}$.

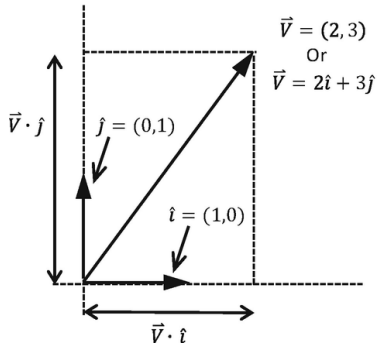

Figure 6-20 The decomposition of vector

In general, as illustrated in Figure 6-21, given the normalized perpendicular component vectors $\hat{L}$ and $\hat{M}$ and any vector $\vec{V}$, the following formulae are always true:

$$\vec{V} = (\vec{V} \cdot \hat{\imath})\hat{\imath} + (\vec{V} \cdot \hat{\jmath})\hat{\jmath}$$
$$\vec{V} = (\vec{V} \cdot \hat{L})\hat{L} + (\vec{V} \cdot \hat{M})\hat{M}$$

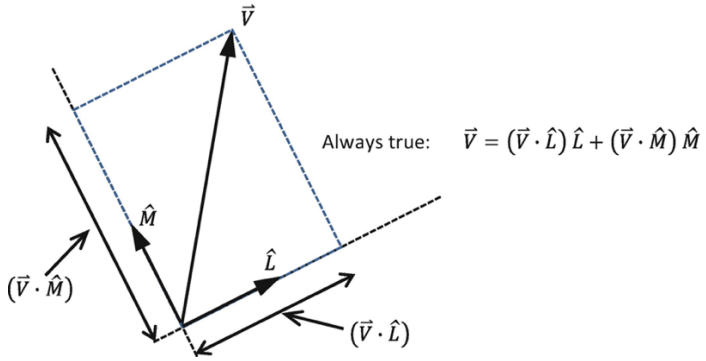

Figure 6-21 Decomposing a vector by two normalized component vectors

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 ($\hat{\imath}$) and y axis ($\hat{\jmath}$). You handled this case in the previous project. You can see an example of this in Figure 6-22.

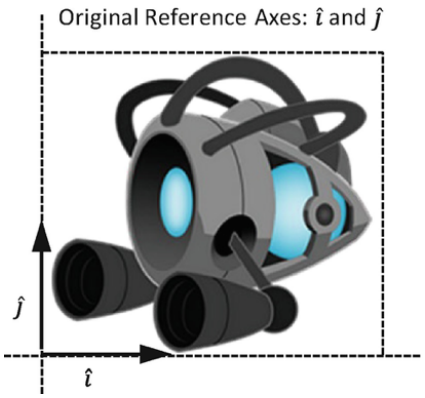

Figure 6-22 An axes-aligned texture

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 6-23.

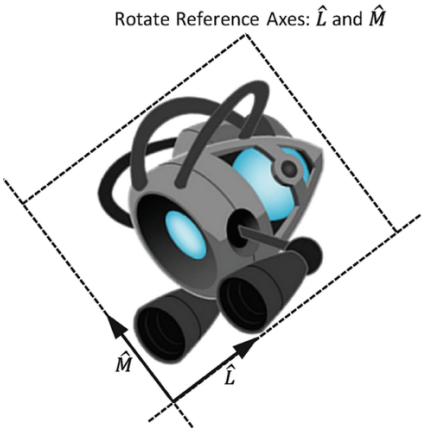

Figure 6-23 A rotated texture and its component vectors

Lab 6-5
The General Pixel Collisions Project

This project demonstrates how to detect a collision between two rotated `TextureRenderable` objects with per-pixel accuracy. Similar to the previous project, a yellow `DyePack` object (as a test confirmation) will be displayed at the detected colliding position.

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 Pixel Collision to Support Rotation

1. Edit the `texture_renderable_pixel_collision.js` file and modify the `_indexToWCPosition()` function:

```
TextureRenderable.prototype._indexToWCPosition =
function (returnWCPos, i, j, xDir, yDir) {
  let x = i * this.mXform.getWidth() / this.mElmWidthPixels;
  let y = j * this.mXform.getHeight() / this.mElmHeightPixels;
  let xDisp = x - (this.mXform.getWidth() * 0.5);
  let yDisp = y - (this.mXform.getHeight() * 0.5);
  let xDirDisp = [];
  let 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 listed 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 `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. Note that `xDirDisp` and `yDirDisp` are 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]
  let delta = [];
  vec2.sub(delta, wcPos, this.mXform.getPosition());
  let xDisp = vec2.dot(delta, xDir);
  let yDisp = vec2.dot(delta, yDir);
  returnIndex[0] = this.mElmWidthPixels *
                   (xDisp / this.mXform.getWidth());
  returnIndex[1] = this.mElmHeightPixels *
                   (yDisp / this.mXform.getHeight());
  // recall that xForm.getPosition() returns center, yet
  // Texture origin is at lower-left corner!
  returnIndex[0] += this.mElmWidthPixels / 2;
  returnIndex[1] += this.mElmHeightPixels / 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) {
  let pixelTouch = false;
  let xIndex = 0, yIndex;
  let otherIndex = [0, 0];
  let xDir = [1, 0];
  let yDir = [0, 1];
  let otherXDir = [1, 0];
  let 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.mElmWidthPixels)) {
      yIndex = 0;
      while ((!pixelTouch) && (yIndex < this.mElmHeightPixels)) {
          if (this._pixelAlphaValue(xIndex, yIndex) > 0) {
              this._indexToWCPosition(wcTouchPos,
                                      xIndex, yIndex, xDir, yDir);
              other._wcPositionToIndex(otherIndex, wcTouchPos,
                                       otherXDir, otherYDir);
              if ((otherIndex[0] >= 0) &&
                  (otherIndex[0] < other.mElmWidthPixels) &&
                  (otherIndex[1] >= 0) &&
                  (otherIndex[1] < other.mElmHeightPixels)) {
                  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 GameObject 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 6-13, the `BoundingBox` object does not support object rotation correctly, and the following code remedies this shortcoming:

```
pixelTouches(otherObj, wcTouchPos) {
    // only continue if both objects have getColorArray defined
    // if defined, should have other texture intersection support!
    let pixelTouch = false;
    let myRen = this.getRenderable();
    let 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 ...
            let 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 hypotenuse as radius
            let mySize = myRen.getXform().getSize();
            let otherSize = otherRen.getXform().getSize();
            let myR = Math.sqrt(0.5*mySize[0]*0.5*mySize[0] +
                                0.5*mySize[1]*0.5*mySize[1]);
            let otherR = Math.sqrt(0.5*otherSize[0]*0.5*otherSize[0] +
                                   0.5*otherSize[1]*0.5*otherSize[1]);
            let 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 listed 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.

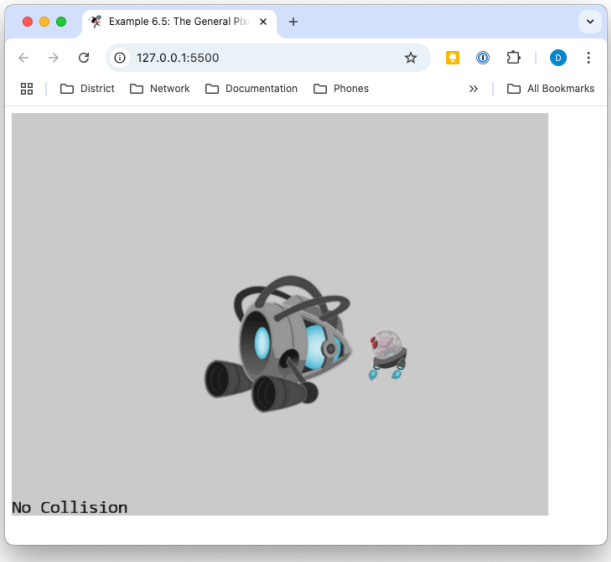

Figure 6-24. Running the General Pixel Collisions Project

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 remedy this deficiency.

Lab 6-6
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.

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 follows:

> To generalize the per-pixel collision implementation for sprite and animated sprite objects

You can need to put the following external resource files in the assets folder: the `fonts` folder that contains the default system fonts, `minion_sprite.png`, and `minion_portal.png`.

Implement Per-Pixel Collision for SpriteRenderable

Edit `sprite_renderable.js` to implement the per-pixel-specific support for `SpriteRenderable` objects:

1. Modify the `SpriteRenderable` constructor to call the `_setTexInfo()` function to initialize per-pixel collision parameters; this function is defined in the next step:

```
  constructor(myTexture) {
    super(myTexture);
    super._setShader(shaderResources.getSpriteShader());
    // sprite coordinate
    // bounds of texture coordinate (0 is left, 1 is right)
    this.mElmLeft = 0.0;
    this.mElmRight = 1.0;
    this.mElmTop = 1.0;    //   1 is top and 0 is bottom of image
    this.mElmBottom = 0.0; //
    // sets info to support per-pixel collision
    this._setTexInfo();
}
```

2. Define the `_setTexInfo()` function to override instance variables defined in the `TextureRenderable` superclass. Instead of the entire texture image, the instance variables now identify the currently active sprite element.

```
_setTexInfo() {
  let imageW = this.mTextureInfo.mWidth;
  let imageH = this.mTextureInfo.mHeight;
  this.mElmLeftIndex = this.mElmLeft * imageW;
  this.mElmBottomIndex = this.mElmBottom * imageH;
  this.mElmWidthPixels = ((this.mElmRight - this.mElmLeft)*imageW)+1;
  this.mElmHeightPixels = ((this.mElmTop - this.mElmBottom)*imageH)+1;
}
```

Notice that instead of the dimension of the entire texture map, `mElmWidthPixel` and `mElmHeightPixel` now contain pixel values that correspond to the dimension of a single sprite element in the sprite sheet.

3. Remember to call the `_setTexInfo()` function when the current sprite element is updated in the `setElementUVCoordinate()` and `setElementPixelPositions()` functions:

```
setElementUVCoordinate(left, right, bottom, top) {
  this.mElmLeft = left;
  this.mElmRight = right;
  this.mElmBottom = bottom;
  this.mElmTop = top;
  this._setTexInfo();
}
setElementPixelPositions(left, right, bottom, top) {
  // entire image width, height
  let imageW = this.mTextureInfo.mWidth;
  let imageH = this.mTextureInfo.mHeight;
  this.mElmLeft = left / imageW;
  this.mElmRight = right / imageW;
  this.mElmBottom = bottom / imageH;
  this.mElmTop = top / imageH;
  this._setTexInfo();
}
```

Support Accesses to Sprite Pixels in TextureRenderable

Edit the `texture_renderable_pixel_collision.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.mElmBottomIndex;
  x += this.mElmLeftIndex;
  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

Observation

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. Press 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. Press 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 a simpler and better organized control logic in the client's `MyGame` test levels. You reviewed vectors in 2D space. A vector is defined by its direction and magnitude. Vectors are convenient for describing displacements (velocities). You reviewed some foundational vector operations, including normalization of a vector and how to calculate both 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 became a prominent omission as the behaviors of objects increased in sophistication. The axis-aligned bounding boxes, or AABBs, were introduced as a crude, yet computationally efficient solution for approximating object collisions. You learned the algorithm for per-pixel-accurate collision detection and that its accuracy comes at the cost of performance. 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 invoke the pixel iteration process based on the texture with a lower resolution.

When implementing pixel-accurate collision, you began with tackling the basic case of working with axis-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 Camera objects 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, while 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 to 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 scenarios.

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 conveys a sense of place. The hero character appears to be flying through a moving scene populated by a number of mechanized robots, and there's 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 Objects project, what kind of behaviors would you choose and what kind of actions would you require the player to perform? 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 pursue the hero in an attempt to stop the player's progress; maybe the hero is also penalized in some way if they come into contact with a robot. Imagine perhaps that the small object in the center of the screen allows the hero to be invincible for a fixed period of time and that we've designed the level to require temporary invincibility to reach the goal, thus creating a more complex and interesting game loop (e.g., avoid the pursuing robots to reach the power up, activate the power up and become temporarily invincible, use invincibility to reach the goal). With these few basic interactions, we've opened opportunities to explore mechanics and level designs that will feel very familiar from many different kinds of games, 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 Objects 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 12 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. Limiting yourself to just the elements and interactions covered through Chapter 6 is actually a beneficial exercise as design constraints often help the creative process by shaping and guiding your ideas. Even the most advanced video games typically have a fairly basic set of core game loops as their foundation.

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, allowing 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 and the precise but resource-intensive per-pixel detection method. There are many situations where the bounding-box approach is sufficient, but if players perceive collisions to be arbitrary because the bounding boxes are too different 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 causal chains around a new object being produced as the result of two objects colliding (e.g., 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 describes 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 covering the spectrum 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 and 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.

Appendix A.
6.1 my_game.js

```
"use strict";  // Operate in Strict mode such that variables must be declared before used!

import engine from "../engine/index.js";

// user stuff
import DyePack from "./objects/dye_pack.js";
import Minion from "./objects/minion.js";
import Hero from "./objects/hero.js";

class MyGame extends engine.Scene {
   constructor() {
      super();
      this.kMinionSprite = "assets/minion_sprite.png";
      // The camera to view the scene
      this.mCamera = null;

      // For echo message
      this.mMsg = null;

      // the hero and the support objects
      this.mHero = null;
      this.mMinionset = null;
      this.mDyePack = null;
   }

   load() {
      engine.texture.load(this.kMinionSprite);
   }

   unload() {
      engine.texture.unload(this.kMinionSprite);
   }

   init() {
      // Step A: set up the cameras
      this.mCamera = new engine.Camera(
         vec2.fromValues(50, 37.5),   // position of the camera
         100,                     // width of camera
         [0, 0, 640, 480]         // viewport (orgX, orgY, width, height)
      );
      this.mCamera.setBackgroundColor([0.8, 0.8, 0.8, 1]);
```

```
    // sets the background to gray
    //
    // Step B: The dye pack: simply another GameObject
    this.mDyePack = new DyePack(this.kMinionSprite);

    // Step C: A set of Minions
    this.mMinionset = new engine.GameObjectSet();
    let 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 engine.FontRenderable("Status Message");
    this.mMsg.setColor([0, 0, 0, 1]);
    this.mMsg.getXform().setPosition(1, 2);
    this.mMsg.setTextHeight(3);
}

// This is the draw function, make sure to setup proper drawing environment, and more
// importantly, make sure to _NOT_ change any state.
draw() {
    // Step A: clear the canvas
    engine.clearCanvas([0.9, 0.9, 0.9, 1.0]); // clear to light gray

    // Step  B: Activate the drawing Camera
    this.mCamera.setViewAndCameraMatrix();

    // Step  C: draw everything
    this.mHero.draw(this.mCamera);
    this.mMinionset.draw(this.mCamera);
    this.mDyePack.draw(this.mCamera);
    this.mMsg.draw(this.mCamera);
}

// The Update function, updates the application state. Make sure to _NOT_ draw
// anything from this function!
update() {
```

```
        this.mHero.update();
        this.mMinionset.update();
        this.mDyePack.update();
    }
}

window.onload = function () {
    engine.init("GLCanvas");

    let myGame = new MyGame();
    myGame.start();
}
```