Chapter 8

2D Game Programming Tricks

Chapter 7 demonstrated that OpenGL ES offers us quite a lot of features to exploit for 2D graphics programming, such as easy rotation and scaling, and automatic stretching of our view frustum to the viewport. It also offers performance benefits over using the `Canvas`.

Now it's time to look at some of the more advanced topics of 2D game programming. Some of these concepts we used intuitively when we wrote Mr. Nom, including time-based state updates and image atlases. A lot of what's to come is indeed very intuitive as well, and chances are high that you'd have come up with the same solution sooner or later. But it doesn't hurt to learn about these things explicitly.

We will look at a handful of crucial concepts for 2D game programming. Some of them will be graphics related, and others will deal with how we represent and simulate our game world. All of these have one thing in common: they rely on a little linear algebra and trigonometry. Fear not, the level of math we need to write games like Super Mario Brothers is not exactly mind blowing. Let's begin by reviewing some concepts of 2D linear algebra and trigonometry.

Before We Begin

As with the previous "theoretical" chapters, we are going to create a couple of examples to get a feel for what's happening. For this chapter we'll reuse what we developed in the last chapter, mainly the `GLGame`, `GLGraphics`, `Texture`, and `Vertices` classes, along with the rest of the framework classes.

Our demo project consists of a starter called `GameDev2DStarter`, which presents a list of tests to run. We can reuse the code of the `GLBasicsStarter` and simply replace the class names of the tests. We also have to add each of the tests to the manifest in the form of `<activity>` elements.

M. Zechner, *Beginning Android Games*
© Mario Zechner 2011

Each of the tests is again an instance of the `Game` interface, and the actual test logic is implemented in the form of a `Screen` contained in the `Game` implementation of the test, as in the previous chapter. I will only present the relevant portions of the `Screen` to conserve some pages. The naming conventions are again `XXXTest` and `XXXScreen` for the `GLGame` and `Screen` implementation of each test.

With that out of our way, let's talk about vectors.

In the Beginning There Was the Vector

In the last chapter I told you that vectors shouldn't be mixed up with positions. This is not entirely true, as we can (and will) represent a position in some space via a vector. A vector can actually have many interpretations:

- *Position*: We already used this in the previous chapters to encode the coordinates of our entities relative to the origin of the coordinate system.
- *Velocity and acceleration*: These are physical quantities we'll talk about in the next section. While we are used to thinking about velocity and acceleration as being a single value, they should actually be represented as 2D or 3D vectors. They encode not only the speed of an entity (e.g., a car driving at 100 kilometers per hour), but also the direction the entity is traveling in. Note that this kind of vector interpretation does not state that the vector is given relative to the origin. This makes sense since the velocity and direction of a car is independent of its position. Think of a car traveling northwest on a straight highway at 100 kilometers per hour. As long as its speed and direction don't change, the velocity vector won't change either.
- *Directions and distances*: Directions are similar to velocities but lack physical quantities in general. We can use such a vector interpretation to encode states such as *this entity is pointing southeast*. Distances just tell us the how far away and in what direction a position is from another position.

Figure 8–1 shows these interpretations in action.

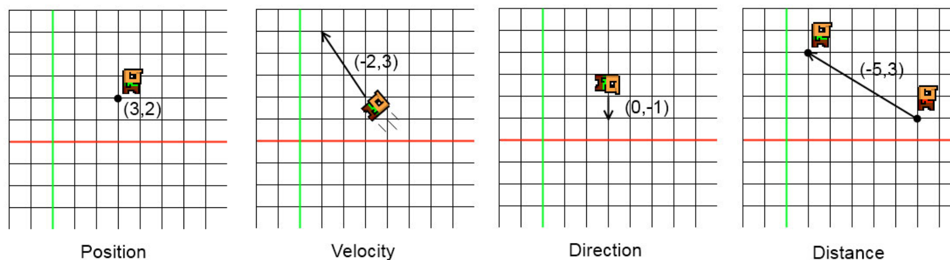

Figure 8–1. *Bob, with position, velocity, direction, and distance expressed as vectors*

Figure 8–1 is of course not exhaustive. Vectors can have a lot more interpretations. For our game development needs, however, these four basic interpretations suffice.

One thing that's left out from Figure 8–1 is what units the vector components have. We always have to make sure that those are sensible (e.g., Bob's velocity could be in meters per second, so he travels 2 meters to the left and 3 meters up in 1 second). The same is true for positions and distances, which we could also express in meters, for example. The direction of Bob is a special case, though—it is unitless. This will come in handy if we want to specify the general direction of an object while keeping the direction's physical features separate. We could do this for the velocity of Bob, storing the direction of his velocity as a direction vector and his speed as a single value. Single values are also known as *scalars*. For this, the direction vector must be of length 1, as we'll discuss later on.

Working with Vectors

The power of vectors stems from the fact that we can easily manipulate and combine them. Before we can do that, though, we need to define how we represent vectors:

```
v = (x,y)
```

Now, that wasn't a big surprise; we've done that a gazillion times already. Every vector has an x and a y component in our 2D space (yes, we'll be staying in two dimensions in this chapter). We can also add two vectors:

```
c = a + b = (a.x, a.y) + (b.x, b.y) = (a.x + b.x, a.y + b.y)
```

All we need to do is add the components together to arrive at the final vector. Try it out with the vectors given in Figure 8–1. Say we take Bob's position, $p = (3,2)$, and add his velocity, $v = (-2,3)$. We arrive at a new position, $p' = (3 + -2, 2 + 3) = (1,5)$. Don't get confused by the apostrophe behind the p here, it's just there to denote that we have a new vector p. Of course, this little operation only makes sense when the units of the position and velocity fit together. In this case we assume the position is given in meters (m) and the velocity is given in meters per second (m/s), which fits perfectly fine.

Of course, we can also subtract vectors:

```
c = a - b = (a.x, a.y) - (b.x, b.y) = (a.x - b.x, a.y - b.y)
```

Again, all we do is combine the components of the two vectors. Note, however, that the order in which we subtract one vector from the other is important. Take the rightmost image in Figure 8–1, for example. We have a green Bob at pg = (1,4) and a red Bob at pr = (6,1), where pg and pr stand for position green and red respectively.. When we take the distance vector from green Bob to red Bob, we calculate the following:

```
d = pg - pr = (1, 4) - (6, 1) = (-5, 3)
```

Now that is strange. That vector is actually pointing from red Bob to green Bob! To get the direction vector from green Bob to red Bob, we have to reverse the order of subtraction:

```
d = pr - pg = (6, 1) - (1, 4) = (5, -3)
```

If we want to find the distance vector from a position *a* to a position *b*, we use the following general formula:

```
d = b - a
```

In other words, we always subtract the start position from the end position. That's a little confusing at first, but if you think about it, it makes absolute sense. Try it out on some graph paper!

We can also multiply a vector by a scalar (remember, a scalar is just a single value):

```
a' = a * scalar = (a.x * scalar, a.y * scalar)
```

We multiply each of the components of the vector by the scalar. This allows us to scale the length of a vector. Take the direction vector in Figure 8–1 as an example. It's specified as $d = (0,-1)$. If we multiply it with the scalar $s = 2$, we effectively double its length: $d \times s = (0,-1 \times 2) = (0,-2)$. We can of course make it smaller as well, by using a scalar less than 1—for example, *d* multiplied by $s = 0.5$ creates a new vector $d' = (0,-0.5)$.

Speaking of length, we can also calculate the length of a vector (in the units it's given in):

```
|a| = sqrt(a.x*a.x + a.y*a.y)
```

The |a| notation just tells us that this represents the length of the vector. If you didn't sleep through your linear algebra class at school, you might recognize the formula for the vector length. It's the Pythagorean theorem applied to our fancy 2D vector. The x and y components of the vector form two sides of a right triangle, and the third side is the length of the vector. Figure 8–2 illustrates this.

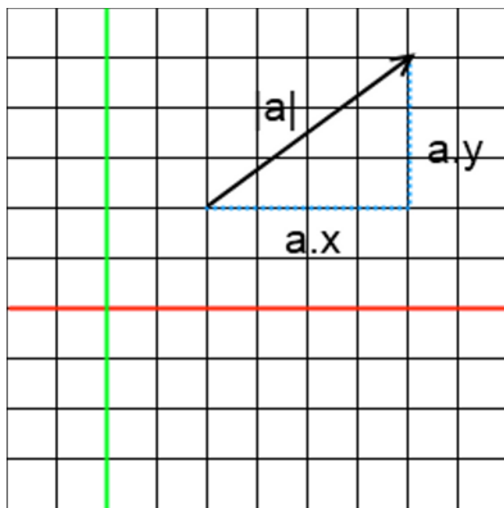

Figure 8–2. *Pythagoras would love vectors too*

The vector length is always positive or zero, given the properties of the square root. If we apply this to the distance vector between the red and green Bob, we can figure out that they are

```
|pr - pg| = sqrt(5*5 + -3*-3) = sqrt(25 + 9) = sqrt(34) ~= 5.83m
```

apart from each other (if their positions are given in meters). Note that if we calculated |*pg* – *pr*|, we'd arrive at the same value, as the length is independent of the direction of the vector. This new knowledge also has another implication: when we multiply a vector with a scalar, its length changes accordingly. Given a vector *d* = (0,–1) with an original length of 1 unit, we can multiply it by 2.5 and arrive at a new vector with a length of 2.5 units.

We discussed that direction vectors usually don't have any units associated with them. We can make them have a unit by multiplying them with a scalar—for example, we can multiply a direction vector *d* = (0,1) with a speed constant *s* = 100 m/s to get a velocity vector *v* = (0 × 100,1 × 100) = (0,100). It's therefore always a good idea to let our direction vectors have a length of 1. Vectors with a length of 1 are called *unit vectors*. We can make any vector a unit vector by dividing each of its components by its length:

```
d' = (d.x/|d|, d.y/|d|)
```

Remember that |d| just means the length of the vector *d*. Let's try it out. Say we want a direction vector that points exactly northeast: *d* = (1,1). It might seem that this vector is already unit length, as both components are 1, right? Wrong:

```
|d| = sqrt(1*1 + 1*1) = sqrt(2) ~= 1.44
```

We can easily fix that by making the vector a unit vector:

```
d' = (d.x/|d|, d.y/|d|) = (1/|d|, 1/|d|) ~= (1/1.44, 1/1.44) = (0.69, 0.69)
```

This is also called *normalizing* a vector, which just means that we make it have a length of 1. With this little trick we can create a unit-length direction vector out of a distance vector, for example. Of course, we have to watch out for zero-length vectors, as we'd have division by zero in that case!

A Little Trigonometry

Let's turn to trigonometry for a minute. There are two essential functions in trigonometry: *cosine* and *sine*. Each takes a single argument: an *angle*. We are used to specifying angles in degrees (e.g., 45° or 360°). In most math libraries, trigonometry functions expect the angle in radians, though. We can easily convert between degrees and radians with the following equations:

```
degreesToRadians(angleInDegrees) = angleInDegrees / 180 * pi
radiansToDegrees(angle) = angleInRadians / pi * 180
```

Here, `pi` is our beloved superconstant, with an approximate value of 3.14159265. `pi` radians equal 180 degrees, so that's how the preceding functions come to be.

So what do cosine and sine actually calculate given an angle? They calculate the x and y components of a unit-length vector relative to the origin. Figure 8–3 illustrates this.

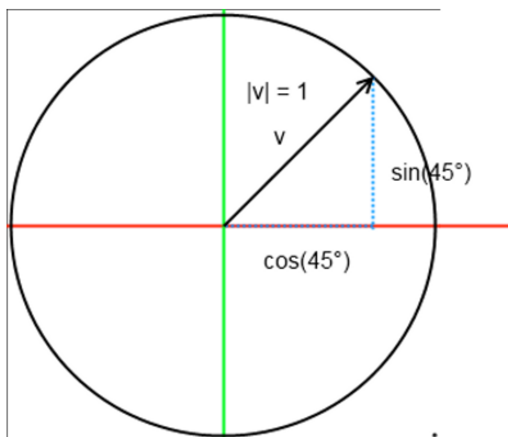

Figure 8–3. *Cosine and sine produce a unit vector with its endpoint lying on the unit circle*

Given an angle, we can therefore easily create a unit-length direction vector like this:

```
v = (cos(angle), sin(angle))
```

We can go the other way around as well, and calculate the angle of a vector with respect to the x-axis:

```
angle = atan2(v.y, v.x)
```

The atan2 function is actually an artificial construct. It uses the arcus tangent function (which is the inverse of the tangent function, which is another fundamental function in trigonometry) to construct an angle in the range of –180 degrees to 180 degrees (or –`pi` to `pi`, if the angle is returned in radians). The internals are a little involved and do not matter all that much for our discussion. The arguments are the y and x components of our vector. Note that the vector does not have to be a unit vector for the atan2 function to work. Also note that the y component is most often given first, and then the x component—but this depends on the math library we use. This is a common source for errors.

Let's try a few examples. Given a vector $v = (\cos(97°),\sin(97°))$, the result of atan2(sin(97°),cos(97°)) is 97°. Great, that was easy. Using a vector $v = (1,–1)$, we get atan2(–1,1) = –45°. So if our vector's y component is negative, we'll get a negative angle in the range 0° to –180°. We can fix this by adding 360° (or `2pi`) if the output of atan2 is negative. In the preceding example, we'd then get 315°.

The final operation we want to be able to apply to our vectors is rotating them by some angle. The derivation of the equations that follow are again a little involved. Luckily we can just use them as is without knowing about orthogonal base vectors (hint: that's the key phrase to search for on the Web if you want to know what's going on under the hood). Here's the magical pseudocode:

```
v.x' = cos(angle) * v.x - sin(angle) * v.y
v.y' = sin(angle) * v.x + cos(angle) * v.y
```

Woah, that was less complicated then expected. This will rotate any vector counterclockwise around the origin, no matter what interpretation we have of the vector.

Together with vector addition, subtraction, and multiplication by a scalar, we can actually implement all the OpenGL matrix operations ourselves. This is one part of the solution to further increase the performance of our `BobTest` in the last chapter. We'll talk about this in one of the following sections. For now, let's concentrate on what we discussed and transfer it to code.

Implementing a Vector Class

We want to create an easy-to-use vector class for 2D vectors. Let's call it `Vector2`. It should have two members, for holding the x and y components of the vector. Additionally it should have a couple of nice methods that allow us to the following:

- Add and subtract vectors
- Multiply the vector components with a scalar
- Measure the length of a vector
- Normalize a vector
- Calculate the angle between a vector and the x-axis
- Rotate the vector

Java lacks operator overloading, so we have to come up with a mechanism that makes working with the `Vector2` class less cumbersome. Ideally we should have something like the following:

```
Vector2 v = new Vector2();
v.add(10,5).mul(10).rotate(54);
```

We can easily achieve this by letting each of the `Vector2` methods return a reference to the vector itself. Of course, we also want to overload methods like `Vector2.add()` so that we can either pass in two floats or an instance of another `Vector2`. Listing 8–1 shows our `Vector2` class in its full glory.

Listing 8–1. *Vector2.java: Implementing Some Nice 2D Vector Functionality*

```
package com.badlogic.androidgames.framework.math;

import android.util.FloatMath;

public class Vector2 {
    public static float TO_RADIANS = (1 / 180.0f) * (float) Math.PI;
    public static float TO_DEGREES = (1 / (float) Math.PI) * 180;
    public float x, y;

    public Vector2() {
    }

    public Vector2(float x, float y) {
        this.x = x;
        this.y = y;
    }
```

```
    public Vector2(Vector2 other) {
        this.x = other.x;
        this.y = other.y;
    }
```

We put that class in the package `com.badlogic.androidgames.framework.math`, where we'll house any other math-related classes as well.

We start off by defining two static constants, `TO_RADIANS` and `TO_DEGREES`. To convert an angle given in radians, we just need to multiply it by `TO_DEGREES`; to convert an angle given in degrees to radians, we multiply it by `TO_RADIANS`. You can double-check this by looking at the two previously defined equations that govern degree-to-radian conversion. With this little trick we can shave off a division to speed things up a little.

Next we define the two members x and y that store the components of the vector and a couple of constructors—nothing too complex:

```
    public Vector2 cpy() {
        return new Vector2(x, y);
    }
```

We also have a `cpy()` method that will create a duplicate instance of the current vector and return it. This might come in handy if we want to manipulate a copy of a vector, preserving the value of the original vector.

```
    public Vector2 set(float x, float y) {
        this.x = x;
        this.y = y;
        return this;
    }

    public Vector2 set(Vector2 other) {
        this.x = other.x;
        this.y = other.y;
        return this;
    }
```

The `set()` methods allow us to set the x and y components of a vector, based on either two float arguments or another vector. The methods return a reference to this vector, so we can chain operations as discussed previously.

```
    public Vector2 add(float x, float y) {
        this.x += x;
        this.y += y;
        return this;
    }

    public Vector2 add(Vector2 other) {
        this.x += other.x;
        this.y += other.y;
        return this;
    }

    public Vector2 sub(float x, float y) {
        this.x -= x;
```

```
        this.y -= y;
        return this;
    }

    public Vector2 sub(Vector2 other) {
        this.x -= other.x;
        this.y -= other.y;
        return this;
    }
```

The add() and sub() methods come in two flavors: in once case they work with two float arguments, and in the other case they take another Vector2 instance. All four methods return a reference to this vector so we can chain operations.

```
    public Vector2 mul(float scalar) {
        this.x *= scalar;
        this.y *= scalar;
        return this;
    }
```

The mul() method just multiplies the x and y components of the vector with the given scalar value, and again returns a reference to the vector itself for chaining.

```
    public float len() {
        return FloatMath.sqrt(x * x + y * y);
    }
```

The len() method calculates the length of the vector exactly like we defined it previously. Note that we use the FastMath class instead of the usual Math class that Java SE provides. This is a special Android API class that works with floats instead of doubles, and is a little bit faster than the Math equivalent.

```
    public Vector2 nor() {
        float len = len();
        if (len != 0) {
            this.x /= len;
            this.y /= len;
        }
        return this;
    }
```

The nor() method normalizes the vector to unit length. We use the len() method internally to calculate the length first. If it is zero, we can bail out early and avoid a division by zero. Otherwise we divide each component of the vector by its length to arrive at a unit-length vector. For chaining we return the reference to this vector again.

```
    public float angle() {
        float angle = (float) Math.atan2(y, x) * TO_DEGREES;
        if (angle < 0)
            angle += 360;
        return angle;
    }
```

The angle() method calculates the angle between the vector and the x-axis using the atan2() method, as discussed previously. We have to use the Math.atan2() method, as

the FastMath class doesn't have that method. The returned angle is given in radians, so we convert it to degrees by multiplying it by TO_DEGREES. If the angle is less than zero, we add 360 degrees to it so we can return a value in the range 0 to 360 degrees.

```
    public Vector2 rotate(float angle) {
        float rad = angle * TO_RADIANS;
        float cos = FloatMath.cos(rad);
        float sin = FloatMath.sin(rad);

        float newX = this.x * cos - this.y * sin;
        float newY = this.x * sin + this.y * cos;

        this.x = newX;
        this.y = newY;

        return this;
    }
```

The rotate() method simply rotates the vector around the origin by the give angle. Since the FastMath.cos() and FastMath.sin() methods expect the angle to be given in radians, we first convert from degrees to radians. Next we use the previously defined equations to calculate the new x and y components of the vector, and finally return the vector itself again for chaining.

```
    public float dist(Vector2 other) {
        float distX = this.x - other.x;
        float distY = this.y - other.y;
        return FloatMath.sqrt(distX * distX + distY * distY);
    }

    public float dist(float x, float y) {
        float distX = this.x - x;
        float distY = this.y - y;
        return FloatMath.sqrt(distX * distX + distY * distY);
    }
}
```

Finally we have two methods that calculate the distance between this and another vector.

And that's our shiny Vector2 class, which we'll use to represent positions, velocities, distances, and directions in the code that follows. To get a feeling for our new class, let's use it in a simple example.

A Simple Usage Example

Here's my proposal for a simple test:

- We'll create a sort of cannon represented by a triangle that has a fixed position in our world. The center of the triangle will be at (2.4,0.5).
- Each time we touch the screen, we want to rotate the triangle to face the touch point.

- Our view frustum will show us the region of our world between (0,0) and (4.8,3.2). We do not operate in pixel coordinates, but instead define our own coordinate system, were one unit equals one meter. Also, we'll be working in landscape mode.

There are a couple of things we need to think about. We already know how to define a triangle in model space—we can use a `Vertices` instance for this. Our cannon should point to the right at an angle of 0 degrees in its default orientation. Figure 8–4 shows the cannon triangle in model space.

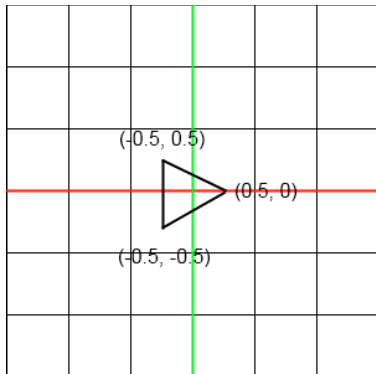

Figure 8–4. *The cannon triangle in model space*

When we render that triangle, we simply use `glTranslatef()` to move it to its place in the world at (2.4,0.5).

We also want to rotate the cannon so that its tip points in the direction of the point on the screen that we last touched. For this we need to figure out where the last touch event was touching our world. The `GLGame.getInput().getTouchX()` and `getTouchY()` methods will return the touch point in screen coordinates, with the origin in the top-left corner. We also said that the `Input` instance will not scale the events to a fixed coordinate system, as it did in Mr. Nom. Instead we will get the coordinates (479,319) when touching the bottom-right corner of the (landscape-oriented) screen on a Hero, and (799,479) on a Nexus One. We need to convert these touch coordinates to our world coordinates. We already did that in the touch handlers in Mr. Nom and the `Canvas`-based game framework; the only difference this time is that our coordinate system extents are a little smaller and our world's y-axis is pointing upward. Here's the pseudocode showing how we can achieve the conversion in the general case, which is nearly the same as in the touch handlers of Chapter 5:

```
worldX = (touchX / Graphics.getWidth()) * viewFrustmWidth
worldY = (1 - touchY / Graphics.getHeight()) * viewFrustumHeight
```

We normalize the touch coordinates to the range (0,1) by dividing them by the screen resolution. In the case of the y-coordinate, we subtract the normalized y-coordinate of the touch event from 1 to flip the y-axis. All that's left is scaling the x- and y-coordinates by the view frustum's width and height—in our case that's 4.8 and 3.2. From `worldX` and

worldY we can then construct a Vector2 that stores the position of the touch point in our world's coordinates.

The last thing we need to do is calculate the angle to rotate the canon with. Let's look at Figure 8–5, which shows our cannon and a touch point in world coordinates.

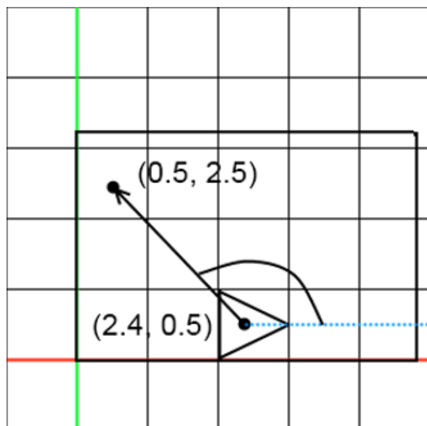

Figure 8–5. *Our cannon in its default state, pointing to the right (angle = 0°), a touch point, and the angle we need to rotate the cannon by. The rectangle is the area of the world that our view frustum will show on the screen: (0,0) to (4.8,3.2).*

All we need to do is create a distance vector from the cannon's center at (2.4,0.5) to the touch point (and remember, we have to subtract the cannon's center from the touch point, not the other way around). Once we have that distance vector we can calculate the angle with the `Vector2.angle()` method. This angle can then be used to rotate our model via `glRotatef()`.

Let's code that. Listing 8–2 shows the relevant portion of our `CannonScreen`, part of the `CannonTest` class.

Listing 8–2. *Excerpt from CannonTest.java; Touching the Screen Will Rotate the Cannon*

```
class CannonScreen extends Screen {
    float FRUSTUM_WIDTH = 4.8f;
    float FRUSTUM_HEIGHT = 3.2f;
    GLGraphics glGraphics;
    Vertices vertices;
    Vector2 cannonPos = new Vector2(2.4f, 0.5f);
    float cannonAngle = 0;
    Vector2 touchPos = new Vector2();
```

We start off with two constants that define our frustum's width and height, as discussed earlier. Next we have a `GLGraphics` instance, as well as a `Vertices` instance. We also store the cannon's position in a `Vector2` and its angle in a float. Finally we have another `Vector2`, which we'll use to calculate the angle between a vector from the origin to the touch point and the x-axis.

Why do we store the `Vector2` instances as class members? We could instantiate them every time we need them, but that would make the garbage collector angry. In general

we should try to instantiate all the `Vector2` instances once and reuse them as often as possible.

```
public CannonScreen(Game game) {
    super(game);
    glGraphics = ((GLGame) game).getGLGraphics();
    vertices = new Vertices(glGraphics, 3, 0, false, false);
    vertices.setVertices(new float[] { -0.5f, -0.5f,
                                        0.5f, 0.0f,
                                       -0.5f, 0.5f }, 0, 6);
}
```

In the constructor, we fetch the `GLGraphics` instance and create the triangle according to Figure 8–4.

```
@Override
public void update(float deltaTime) {
    List<TouchEvent> touchEvents = game.getInput().getTouchEvents();
    game.getInput().getKeyEvents();

    int len = touchEvents.size();
    for (int i = 0; i < len; i++) {
        TouchEvent event = touchEvents.get(i);

        touchPos.x = (event.x / (float) glGraphics.getWidth())
                * FRUSTUM_WIDTH;
        touchPos.y = (1 - event.y / (float) glGraphics.getHeight())
                * FRUSTUM_HEIGHT;
        cannonAngle = touchPos.sub(cannonPos).angle();
    }
}
```

Next up is the `update()` method. We simply loop over all `TouchEvents` and calculate the angle for the cannon. This is done in a couple of steps. First we transform the screen coordinates of the touch event to the world coordinate system, as discussed earlier. We store the world coordinates of the touch event in the `touchPoint` member. We then subtract the position of the cannon from the `touchPoint` vector, which will result in the vector depicted in Figure 8–5. We then calculate the angle between this vector and the x-axis. And that's all there is to it!

```
@Override
public void present(float deltaTime) {

    GL10 gl = glGraphics.getGL();
    gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
    gl.glMatrixMode(GL10.GL_PROJECTION);
    gl.glLoadIdentity();
    gl.glOrthof(0, FRUSTUM_WIDTH, 0, FRUSTUM_HEIGHT, 1, -1);
    gl.glMatrixMode(GL10.GL_MODELVIEW);
    gl.glLoadIdentity();

    gl.glTranslatef(cannonPos.x, cannonPos.y, 0);
    gl.glRotatef(cannonAngle, 0, 0, 1);
    vertices.bind();
    vertices.draw(GL10.GL_TRIANGLES, 0, 3);
```

```
        vertices.unbind();
    }
```

The `present()` method does the same boring things it did before. We set the viewport, clear the screen, set up the orthographic projection matrix using our frustum's width and height, and tell OpenGL ES that all subsequent matrix operations will work on the model-view matrix. We also load an identity matrix to the model-view matrix to "clear" it. Next we multiply the (identity) model-view matrix with a translation matrix, which will move the vertices of our triangle from model space to world space. We also call `glRotatef()` with the angle we calculated in the `update()` method so that our triangle gets rotated in model space before it is translated. Remember, transformations are applied in reverse order—the last specified transform is applied first. Finally we bind the vertices of the triangle, render it, and unbind it.

```
    @Override
    public void pause() {

    }

    @Override
    public void resume() {

    }

    @Override
    public void dispose() {

    }
}
```

Now we have a triangle that will follow our every touch. Figure 8–6 shows the output after touching the upper-left corner of the screen.

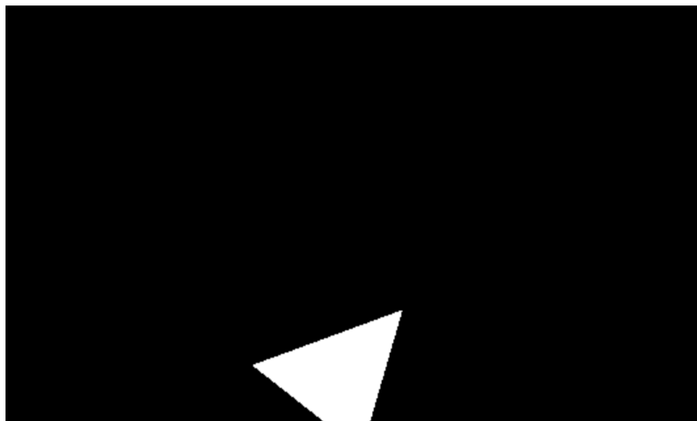

Figure 8–6. *Our triangle cannon reacting to a touch event in the upper-left corner*

Note that it doesn't really matter whether we render a triangle at the cannon position or a rectangle texture mapped to an image of a cannon—OpenGL ES doesn't really care. We also have all the matrix operations in the `present()` method again. The truth of the

matter is that it is easier to keep track of OpenGL ES states this way, and often we will use multiple view frustums in one `present()` call (e.g., one setting up a world in meters for rendering our world and another setting up a world in pixels for rendering UI elements). The impact on performance is not all that big, as described in the last chapter, so it's OK to do it this way most of the time. Just remember that we could optimize this if the need arises.

Vectors will be our best friends from now on. We'll use them to specify virtually everything in our world. We will also do some very basic physics with vectors. What's a cannon good for if it can't shoot, right?

A Little Physics in 2D

In this section we'll use a very simple and limited version of physics. Games are all about being good fakes. They cheat wherever possible to get rid of potentially heavy calculations. The behavior of objects in a game need not be 100 percent physically accurate, it just needs to be good enough to look believable. Sometimes we don't event want physically accurate behavior (e.g., one set of objects should fall downward and another, crazier, set of objects should fall upward).

Even a game like the original Super Mario Brothers uses at least some basic principles of Newtonian physics. These principles are really simple and easy to implement. We will only talk about the absolute minimum required to implement a very simple physics model for our game objects.

Newton and Euler, Best Friends Forever

We are mostly concerned with motion physics of so-called *point masses*, which refers to the change in position, velocity, and acceleration of an object over time. Point mass means that we approximate all our objects with an infinitesimally small point that has an associated mass. We do not deal with things like torque—the rotational velocity of an object around its center of mass—because that is a rather complex problem domain about which more than one complete book has been written. Let's just look at these three properties of an object:

- The position of an object is simply a vector in some space—in our case a 2D space. We represent it as a vector. Usually the position is given in meters.
- The velocity of an object is its change in position per second. Velocity is given as a 2D velocity vector, which is a combination of the unit-length direction vector the object is heading in and the speed that the object will move at, given in meters per seconds. Note that the speed just governs the length of the velocity vector; if we normalize the velocity vector by the speed, we get a nice unit-length direction vector.

- The acceleration of an object is its change in velocity per second. We can either represent this as a scalar that only affects the speed of the velocity (the length of the velocity vector), or as a 2D vector, so that we can have different acceleration in the x- and y-axes. Here we'll choose the latter, as it allows us to use things such as ballistics more easily. Acceleration is usually given in meters per second per second (m/s^2). No, that's not a typo—we change the velocity by some amount given in meters per second, each second.

When we know these properties of an object for a given point in time, we can integrate them to simulate the object's path through the world over time. This may sound scary, but we already did this with Mr. Nom and our Bob test. In those cases we just didn't use acceleration; we set the velocity to a fixed vector. Here's how we can integrate the acceleration, velocity and position of an object in general:

```
Vector2 position = new Vector2();
Vector2 velocity = new Vector2();
Vector2 acceleration = new Vector2(0, -10);
while(simulationRuns) {
   float deltaTime = getDeltaTime();
   velocity.add(acceleration.x * deltaTime, acceleration.y * deltaTime);
   position.add(velocity.x * deltaTime, velocity.y * deltaTime);
}
```

This is called *numerical Euler integration*, and is the most intuitive of the integration methods used in games. We start off with a position at (0,0), a velocity given as (0,0), and an acceleration of (0,–10), which means that the velocity will increase by 1 meter per second on the y-axis. There will be no movement on the x-axis. Before we enter the integration loop, our object is standing still. Within the loop we first update the velocity based on the acceleration multiplied by the delta time, and then update the position based on the velocity times the delta time. That's all there is to the big, scary word *integration*.

NOTE: As usual, that's not even half of the story. Euler integration is an "unstable" integration method and should be avoided when possible. Usually one would employ a variant of the so-called *verlet integration*, which is just a bit more complex. For our purposes, the easier Euler integration is sufficient though.

Force and Mass

You might wonder where the acceleration comes from. That's a good question with many answers. The acceleration of a car comes from its engine. The engine applies a force to the car that causes it to accelerate. But that's not all. Our car will also accelerate toward the center of earth due to gravity. The only thing that keeps it from falling through the center of the earth is the ground it can't pass through. The ground cancels out this gravitational force. The general idea is this:

```
force = mass × acceleration
```

We can rearrange this to the following equation:

```
acceleration = force / mass
```

Force is given in the SI unit *Newton* (guess who came up with this). If we specify acceleration as a vector, then we also have to specify the force as a vector. A force can thus have a direction. For example, the gravitational force pulls downward in the direction (0,–1). The acceleration is also dependent on the mass of an object. The more mass an object has, the more force we need to apply to make it accelerate as fast as an object of less weight. This is a direct consequence of the preceding equations.

For simple games we can, however, ignore the mass and force, and just work with velocity and acceleration directly. In the preceding pseudocode, we set the acceleration to (0,–10) m/s per second (again, not a typo), which is roughly the acceleration an object will experience when it is falling toward earth, no matter its mass (ignoring things like air resistance). It's true, ask Galileo!

Playing Around, Theoretically

So let's use our preceding example to play with an object falling toward earth. Let's assume that we let the loop iterate ten times and that `getDeltaTime()` will always return 0.1 seconds. We'll get the following positions and velocities for each iteration:

```
time=0.1, position=(0.0,-0.1), velocity=(0.0,-1.0)
time=0.2, position=(0.0,-0.3), velocity=(0.0,-2.0)
time=0.3, position=(0.0,-0.6), velocity=(0.0,-3.0)
time=0.4, position=(0.0,-1.0), velocity=(0.0,-4.0)
time=0.5, position=(0.0,-1.5), velocity=(0.0,-5.0)
time=0.6, position=(0.0,-2.1), velocity=(0.0,-6.0)
time=0.7, position=(0.0,-2.8), velocity=(0.0,-7.0)
time=0.8, position=(0.0,-3.6), velocity=(0.0,-8.0)
time=0.9, position=(0.0,-4.5), velocity=(0.0,-9.0)
time=1.0, position=(0.0,-5.5), velocity=(0.0,-10.0)
```

After 1 second, our object falls 5.5 meters and has a velocity of (0,–10) m/s, straight down to the core of the earth (until it hits the ground, of course).

Our object will increase its downward speed without end, as we don't factor in air resistance. (As I said before, we can easily cheat our own system.) We can just enforce a maximum velocity by checking the current velocity length, which equals the speed of our object.

All-knowing Wikipedia tells us that a human in free fall can have a maximum, or terminal, velocity of roughly 125 miles per hour. Converting that to meters per second (125 × 1.6 × 1000 / 3600), we get 55.5 m/s. To make our simulation more realistic, we can modify the loop as follows:

```
while(simulationRuns) {
   float deltaTime = getDeltaTime();
   if(velocity.len() < 55.5)
      velocity.add(acceleration.x * deltaTime, acceleration.y * deltaTime);
   position.add(velocity.x * deltaTime, velocity.y * deltaTime);

}
```

As long as the speed of our object (the length of the velocity vector) is smaller than 55.5 m/s, we increase the velocity by the acceleration. When we've reached the terminal velocity, we simply don't increase it by the acceleration anymore. That simple capping of velocities is a trick used heavily in many games.

We could add some wind to the equation by adding another acceleration in the x direction, say (–1,0) m/s^2. For this we just need to add up the gravitational acceleration and the wind acceleration before we add it to the velocity:

```
Vector2 gravity = new Vector2(0,-10);
Vector2 wind = new Vector2(-1,0);
while(simulationRuns) {
   float deltaTime = getDeltaTime();
   acceleration.set(gravity).add(wind);
   if(velocity.len() < 55.5)
      velocity.add(acceleration.x * deltaTime, acceleration.y * deltaTime);
   position.add(velocity.x * deltaTime, velocity.y * deltaTime);
}
```

We can also ignore acceleration altogether and let our objects have a fixed velocity. We did exactly this in the `BobTest` earlier. We changed the velocity of each Bob only if he hit an edge, and we did so instantly.

Playing Around, Practically

The possibilities, even with this simple model, are endless. Let's extend our little `CannonTest` so we can actually shoot a cannonball. Here's what we want to do:

- As long as the user drags his finger over the screen, the canon will follow it. That's how we'll specify the angle at which we'll shoot the ball.
- As soon as we receive a touch-up event, we'll fire a cannonball in the direction the cannon is pointing. The initial velocity of the cannonball will be a combination of the cannon's direction and the speed the cannonball will have from the start. The speed is equal to the distance between the cannon and the touch point. The further away we touch, the faster the cannonball will fly.
- The cannonball will fly for as long as there's no new touch-up event.
- We'll double the size of our view frustum to (0,0) to (9.6, 6.4) so that we can see more of our world. Additionally we'll place the cannon at (0,0). Note that all units of our world are now given in meters.
- We'll render the cannonball as a red rectangle of the size 0.2×0.2 meters, or 20x20 centimeters. Close enough to a real cannonball, I guess. The pirates among you may choose a more realistic size, of course.

Initially the position of the cannonball will be (0,0)—the same as the cannon's position. The velocity will also be (0,0). Since we apply gravity in each update, the cannonball will just fall straight downward.

Once a touch-up event is received, we set the ball's position back to (0,0) and its initial velocity to (Math.cos(cannonAngle),Math.sin(cannonAngle)). This will ensure that our cannonball flies in the direction the cannon is pointing. We also set the speed by simply multiplying the velocity by the distance between the touch point and the cannon. The closer the touch point to the cannon, the more slowly the cannonball will fly.

Sounds easy enough, so let's implement it. I copied over the code from the CannonTest to a new file, called CannonGravityTest.java. I renamed the classes contained in that file to CannonGravityTest and CannonGravityScreen. Listing 8–3 shows the CannonGravityScreen.

Listing 8–3. *Excerpt from CannonGravityTest*

```
class CannonGravityScreen extends Screen {
    float FRUSTUM_WIDTH = 9.6f;
    float FRUSTUM_HEIGHT = 6.4f;
    GLGraphics glGraphics;
    Vertices cannonVertices;
    Vertices ballVertices;
    Vector2 cannonPos = new Vector2();
    float cannonAngle = 0;
    Vector2 touchPos = new Vector2();
    Vector2 ballPos = new Vector2(0,0);
    Vector2 ballVelocity = new Vector2(0,0);
    Vector2 gravity = new Vector2(0,-10);
```

Not a lot has changed. We simply double the size of the view frustum, and reflect that by setting FRUSTUM_WIDTH and FRUSTUM_HEIGHT to 9.6 and 6.2, respectively. This means that we can see a rectangle of 9.2×6.2 meters of our world. Since we also want to draw the cannonball, I added another Vertices instance, called ballVertices, that will hold the four vertices and six indices of the rectangle of the cannonball. The new members ballPos and ballVelocity store the position and velocity of the cannonball, and the member gravity is the gravitational acceleration, which will stay at a constant (0,–10) m/s^2 over the lifetime of our program.

```
    public CannonGravityScreen(Game game) {
        super(game);
        glGraphics = ((GLGame) game).getGLGraphics();
        cannonVertices = new Vertices(glGraphics, 3, 0, false, false);
        cannonVertices.setVertices(new float[] { -0.5f, -0.5f,
                                                  0.5f, 0.0f,
                                                 -0.5f, 0.5f }, 0, 6);
        ballVertices = new Vertices(glGraphics, 4, 6, false, false);
        ballVertices.setVertices(new float[] { -0.1f, -0.1f,
                                                0.1f, -0.1f,
                                                0.1f,  0.1f,
                                               -0.1f,  0.1f }, 0, 8);
        ballVertices.setIndices(new short[] {0, 1, 2, 2, 3, 0}, 0, 6);
    }
```

In the constructor we simply create the additional Vertices instance for the rectangle of the cannonball. We again define it in model space with the vertices (–0.1,–0.1), (0.1,–0.1), (0.1,0.1), and (–0.1,0.1). We use indexed drawing, so we also specify six vertices in this case.

```
@Override
public void update(float deltaTime) {
    List<TouchEvent> touchEvents = game.getInput().getTouchEvents();
    game.getInput().getKeyEvents();

    int len = touchEvents.size();
    for (int i = 0; i < len; i++) {
        TouchEvent event = touchEvents.get(i);

        touchPos.x = (event.x / (float) glGraphics.getWidth())
                * FRUSTUM_WIDTH;
        touchPos.y = (1 - event.y / (float) glGraphics.getHeight())
                * FRUSTUM_HEIGHT;
        cannonAngle = touchPos.sub(cannonPos).angle();

        if(event.type == TouchEvent.TOUCH_UP) {
            float radians = cannonAngle * Vector2.TO_RADIANS;
            float ballSpeed = touchPos.len();
            ballPos.set(cannonPos);
            ballVelocity.x = FloatMath.cos(radians) * ballSpeed;
            ballVelocity.y = FloatMath.sin(radians) * ballSpeed;
        }
    }

    ballVelocity.add(gravity.x * deltaTime, gravity.y * deltaTime);
    ballPos.add(ballVelocity.x * deltaTime, ballVelocity.y * deltaTime);
}
```

The update() method has also only changed slightly. The calculation of the touch point in world coordinates and the angle of the cannon are still the same. The first addition is the if statement inside the event-processing loop. In case we get a touch-up event, we prepare our cannonball to be shot. We first transform the cannon's aiming angle to radians, as we'll use FastMath.cos() and FastMath.sin() later on. Next we calculate the distance between the cannon and the touch point. This will be the speed of our cannonball. We then set the ball's position to the cannon's position. Finally we calculate the initial velocity of the cannonball. We use sine and cosine, as discussed in the previous section, to construct a direction vector from the cannon's angle. We multiply this direction vector by the cannonball's speed to arrive at our final cannonball velocity. This is interesting, as the cannonball will have this velocity from the start. In the real world, the cannonball would of course accelerate from 0 m/s to whatever it can reach given air resistance, gravity, and the force applied to it by the cannon. We can cheat here, though, as that acceleration would happen in a very tiny time window (a couple hundred milliseconds). The last thing we do in the update() method is update the velocity of the cannonball, and based on that, adjust its position.

```
@Override
    public void present(float deltaTime) {

        GL10 gl = glGraphics.getGL();
        gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();
        gl.glOrthof(0, FRUSTUM_WIDTH, 0, FRUSTUM_HEIGHT, 1, -1);
        gl.glMatrixMode(GL10.GL_MODELVIEW);

        gl.glLoadIdentity();
        gl.glTranslatef(cannonPos.x, cannonPos.y, 0);
        gl.glRotatef(cannonAngle, 0, 0, 1);
        gl.glColor4f(1,1,1,1);
        cannonVertices.bind();
        cannonVertices.draw(GL10.GL_TRIANGLES, 0, 3);
        cannonVertices.unbind();

        gl.glLoadIdentity();
        gl.glTranslatef(ballPos.x, ballPos.y, 0);
        gl.glColor4f(1,0,0,1);
        ballVertices.bind();
        ballVertices.draw(GL10.GL_TRIANGLES, 0, 6);
        ballVertices.unbind();
    }
```

In the `present()` method, we simply add the rendering of the cannonball rectangle. We do this after rendering the cannon's triangle, which means that we have to "clean" the model-view matrix before we can render the rectangle. We do this with `glLoadIdentity()`, and then use `glTranslatef()` to convert the cannonball's rectangle from model space to world space at the ball's current position.

```
    @Override
    public void pause() {

    }

    @Override
    public void resume() {

    }

    @Override
    public void dispose() {

    }
}
```

If you run the example and touch the screen a couple of times, you'll get a pretty good feel for how the cannonball will fly. Figure 8–7 shows the output (which is not all that impressive, since it is a still image).

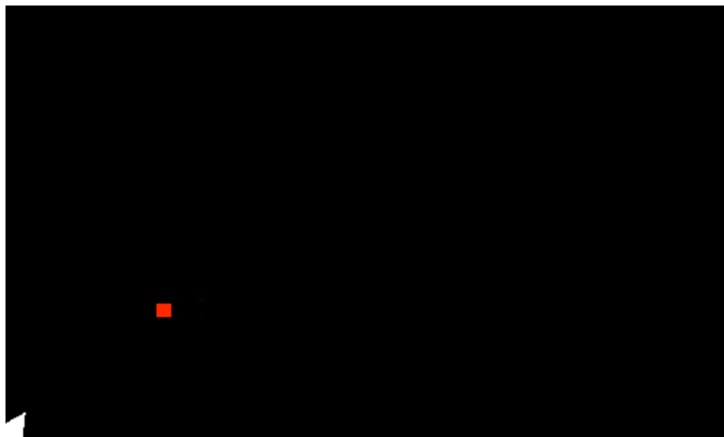

Figure 8–7. *A triangle cannon that shoots red rectangles. Impressive!*

That's enough physics for our purposes. With this simple model, we can simulate much more than cannonballs. Super Mario, for example, could be simulated much in the same way. If you have ever played Super Mario Brothers, then you will notice that Mario takes a little bit of time before he reaches his maximum velocity when running. This can be implemented with a very fast acceleration and velocity capping, as in the preceding pseudocode. Jumping can be implemented in much the same way as we shot the cannonball. Mario's current velocity would be adjusted by an initial jump velocity on the y-axis (remember that we can add velocities like any other vectors). If there were no ground beneath his feet, we would apply gravitational acceleration so that he would actually fall back to the ground. The velocity in the x direction is not influenced by what's happening on the y-axis. We could still press left and right to change the velocity on the x-axis. The beauty of this simple model is that it allows us to implement very complex behavior with very little code. We'll actually use a similar this type of physics when we write our next game.

Just shooting a cannonball is not a lot of fun. We want to be able to hit objects with the cannonball. For this we need something called collision detection, which we'll investigate in the next section.

Collision Detection and Object Representation in 2D

Once we have moving objects in our world, we want them to interact as well. One such mode of interaction is simple collision detection. Two objects are said to be colliding when they overlap in some way. We already did a little bit of collision detection in Mr. Nom when we checked whether Mr. Nom bit himself or ate an ink stain.

Collision detection is accompanied by collision response: once we determine that two objects have collided, we need to respond to that collision by adjusting the position and/or movement of our objects in a sensible manner. For example, when Super Mario jumps on a Goomba, the Goomba goes to Goomba heaven and Mario performs another little jump. A more elaborate example is the collision and response of two or more

billiard balls. We won't go into this kind of collision response, as it is overkill for our purposes. Our collision responses will usually only consist of changing the state of an object (e.g., letting an object explode or die, collecting a coin and setting the score, etc.). This type of response is game dependent, so we won't talk about it in this section.

So how do we figure out whether two objects have collided? First we need to think about when to check for collisions. If our objects follow some sort of simple physics model, as discussed in the last section, we could check for collisions after we move all our objects for the current frame and time step.

Bounding Shapes

Once we have the final positions for our objects, we can perform the collision tests, which boil down to testing for overlap. But what overlaps? Each of our objects needs to have some mathematically defined form or shape that bounds it. The correct term in this case is *bounding shape*. Figure 8–8 shows a couple of choices we have for bounding shapes.

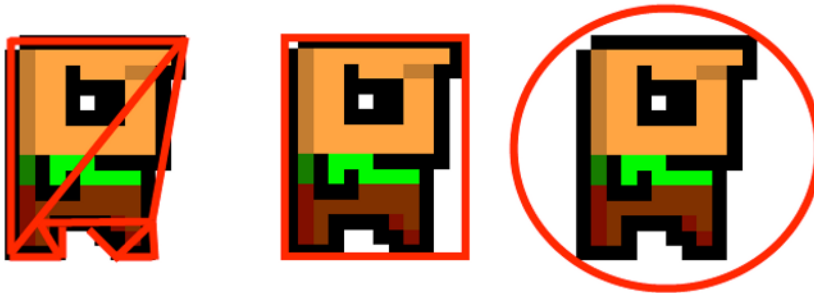

Figure 8–8. *Various bounding shapes around Bob*

The properties of the three types of bounding shapes in Figure 8–8 are as follows:

- *Triangle mesh*: This bounds the object as tightly as possible by approximating its silhouette with a couple of triangles. It requires the most storage space, and it's hard to construct and expensive to test against. It gives the most precise results, though. We won't necessarily use the same triangles for rendering, but instead just store them for collision detection. The mesh can be stored as a list of vertices, where each subsequent three vertices form a triangle. To conserve memory, we could also use indexed vertex lists.

- *Axis-aligned bounding box*: This bounds the object via a rectangle that is axis aligned, which means that the bottom and top edges are always aligned with the x-axis, and the left and right edges are aligned with the y-axis. This is also fast to test against, but less precise than a triangle mesh. A bounding box is usually stored in the form of the position of its lower-left corner plus its width and height. (In the case of 2D, these are also referred to as *bounding rectangles*).
- *Bounding circle*: This bounds the object with the smallest circle that can contain the object. It's very fast to test against, but it is the least precise bounding shape of them all. The circle is usually stored in the form of its center position and its radius.

Every object in our game gets a bounding shape that encloses it, in addition to its position, scale, and orientation. Of course, we need to adjust the bounding shape's position, scale, and orientation according to the object's position, scale, and orientation when we move the object, say, in a physics integration step.

Adjusting for position changes is easy: we simple move the bounding shape accordingly. In the case of the triangle mesh we just move each vertex, in the case of the bounding rectangle we move the lower-left corner, and in the case of the bounding circle we just move the center.

Scaling a bound shape is a little harder. We need to define the point around which we scale. Usually this is the object's position, which is often given as the center of the object. If we use this convention, then scaling is easy as well. For the triangle mesh we scale the coordinates of each vertex; for the bounding rectangle we scale its width, height, and lower-left corner position; and for the bounding circle we scale its radius (the circle center is equal to the object's center).

Rotating a bounding shape is also dependent on the definition of a point around which to rotate. Using the convention just mentioned (where the object center is the rotation point), rotation becomes easy as well. In the case of the triangle mesh, we simple rotate all vertices around the object's center. In the case of the bounding circle, we do not have to do anything, as the radius will stay the same no matter how we rotate our object. The bounding rectangle is a little bit more involved. We need to construct all four corner points, rotate them, and then find the axis-aligned bounding rectangle that encloses those four points. Figure 8–9 shows the three bounding shapes after rotation.

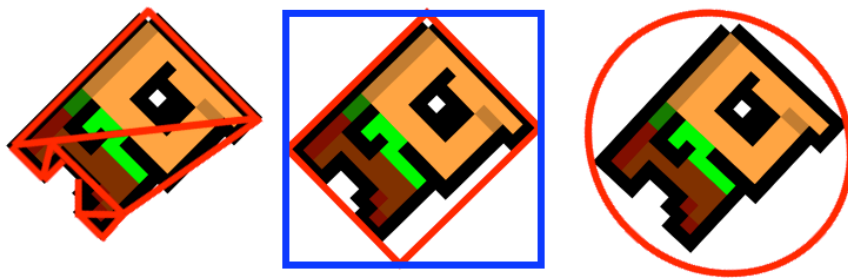

Figure 8–9. *Rotated bounding shapes with the center of the object as the rotation point*

While rotating a triangle mesh or a bounding circle is rather easy, the results for the axis-aligned bounding box are not all that satisfying. Notice that the bounding box of the original object fits tighter than its rotated version. This leads us to the question of how we got our bounding shapes for Bob in the first place.

Constructing Bounding Shapes

In this example, I simply constructed the bounding shapes by hand based on Bob's image. But Bob's image is given in pixels, and our world might operate in, say, meters. The solutions to this problem involve normalization and model space. Imagine the two triangles we'd use for Bob in model space when we'd render him with OpenGL. The rectangle is centered at the origin in model space and has the same aspect ratio (width/height) as Bob's texture image (e.g., 32×32 pixels in the texture map as compared to 2×2 meters in model space). Now we can apply Bob's texture and figure out where in model space the points of the bounding shape are. Figure 8–10 shows how we can construct the bounding shapes around Bob in model space.

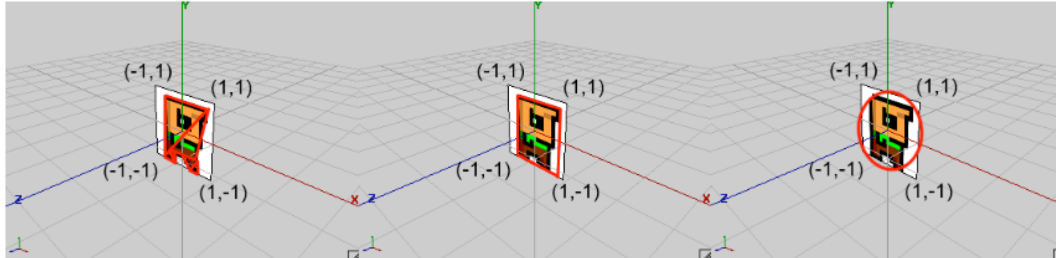

Figure 8–10. *Bounding shapes around Bob in model space*

This process may seem a little cumbersome, but the steps involved are not all that hard. The first thing we have to remember is how texture mapping works. We specify the texture coordinates for each vertex of Bob's rectangle (which is composed of two triangles) in texture space. The upper-left corner of the texture image in texture space is at (0,0), and the lower-left corner is at (1,1), no matter the actual width and height of the image in pixels. To convert from the pixel space of our image to texture space, we can thus use this simple transformation:

```
u = x / imageWidth
v = y / imageHeight
```

where u and v are the texture coordinates of the pixel given by x and y in image space. The `imageWidth` and `imageHeight` are set to the image's dimensions in pixels (32×32 in Bob's case). Figure 8–11 shows how the center of Bob's image maps to texture space.

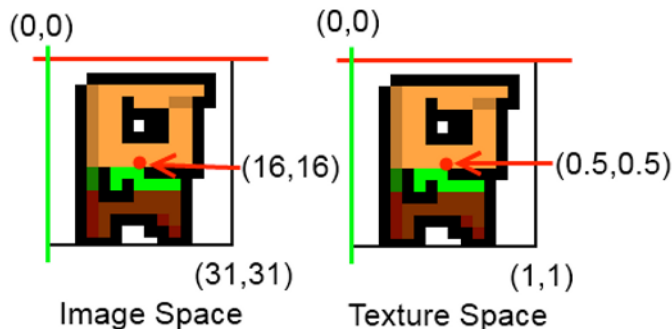

Figure 8–11. *Mapping a pixel from image space to texture space*

The texture is applied to a rectangle that we define in model space. In Figure 8–10 we have an example with the upper-left corner at (–1,1) and the lower-right corner at (1,–1). We use meters as the units in our world, so the rectangle has a width and height of 2 meters each. Additionally we know that the upper-left corner has the texture coordinates (0,0) and the lower-right corner has the texture coordinates (1,1), so we map the complete texture to Bob. This won't always be the case, as you'll see in one of the next sections.

So let's come up with a generic way to map from texture to model space. We can make our lives a little easier by constraining our mapping to only axis-aligned rectangles in texture and model space. This means we assume that an axis-aligned rectangular region in texture space is mapped to an axis-aligned rectangle in model space. For the transformation we need to know the width and height of the rectangle in model space and the width and height of the rectangle in texture space. In our Bob example we have a 2×2 rectangle in model space and a 1×1 rectangle in texture space (since we map the complete texture to the rectangle). We also need to know the coordinates of the upper-left corner of each rectangle in its respective space. For the model space rectangle, that's (–1,1), and for the texture space rectangle it's (0,0) (again, since we map the complete texture, not just a portion). With this information and the u- and v-coordinates of the pixel we want to map to model space, we can do the transformation with these two equations:

```
mx = (u - minU) / (tWidth) × mWidth + minX
my = (1 - ((v - minV) / (tHeight)) × mHeight - minY
```

The variables `u` and `v` are the coordinates we calculated in the last transformation from pixel to texture space. The variables `minU` and `minV` are the coordinates of the top-left corner of the region we map from texture space. The variables `tWidth` and `tHeight` are the width and height of our texture space region. The variables `mWidth` and `mHeight` are the width and height of our model space rectangle. The variables `minX` and `minY` are—you guessed it—the coordinates of the top-left corner of the rectangle in model space. Finally we have `mx` and `my`, which are the transformed coordinates in model space.

These equations take the u- and v-coordinates, map them to the range 0 to 1, and then scale and position them in model space. Figure 8–12 shows a texel in texture space and

how it is mapped to a rectangle in model space. On the sides you see tWidth and tHeight, and mWidth and mHeight, respectively. The top-left corner of each rectangle corresponds to (minU, minV) in texture space and (minX, minY) in model space.

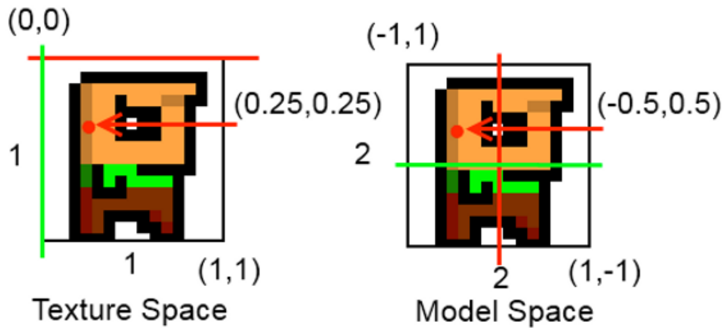

Figure 8–12. *Mapping from texture space to model space*

Substituting the first two equations we can directly go from pixel space to model space:

```
mx = ((x/imageWidth) - minU) / (tWidth) * mWidth + minX
my = (1 - (((y/imageHeight) - minV) / (tHeight)) * mHeight - minY
```

We can use these two equations to calculate the bounding shapes of our objects based on the image we map to their rectangles via texture mapping. In the case of the triangle mesh, this can get a little tedious; the bounding rectangle and bounding circle cases are a lot easier. Usually we don't go this hard route, but instead try to create our textures so that at least the bounding rectangles have the same aspect ratio as the rectangle we render for the object via OpenGL ES. This way we can construct the bounding rectangle from the object's image dimension directly. The same is true for the bounding circle. I just wanted to show you how you can construct an arbitrary bounding shape given an image that gets mapped to a rectangle in model space.

You should now know how to construct a nicely fitting bounding shape for your 2D objects. But remember, we define those bounding shape sizes manually, when we create our graphical assets and define the units and sizes of our objects in the game world. We then use these sizes in our code to collide objects with each other.

Game Object Attributes

Bob just got fatter. In addition to the mesh we use for rendering (the rectangle mapping to Bob's image texture), we now have a second data structure holding his bounds in some form. It is crucial to realize that while we model the bounds after the mapped version of Bob in model space, the actual bounds are independent of the texture region we map Bob's rectangle to. Of course, we try to have as close a match to the outline of Bob's image in the texture as possible when we create the bounding shape. It does not matter, however, whether the texture image is 32×32 or 128×128 pixels. An object in our world thus has three attribute groups:

- Its position, orientation, scale, velocity, and acceleration. With these we can apply our physics model from the previous section. Of course, some objects might be static, and thus will only have position, orientation, and scale. Often we can even leave out orientation and scale. The position of the object usually coincides with the origin in model space, as in Figure 8–10. This makes some calculations easier.
- Its bounding shape (usually constructed in model space around the object's center), which coincides with its position and is aligned with the object's orientation and scale, as shown in Figure 8–10. This gives our object a boundary and defines its size in the world. We can make this shape as complex as we want. We could, for example, make it a composite of several bounding shapes.
- Its graphical representation. As shown in Figure 8–12, we still use two triangles to form a rectangle for Bob and texture-map his image onto the rectangle. The rectangle is defined in model space but does not necessarily equal the bounding shape, as shown in Figure 8–10. The graphical rectangle of Bob that we send to OpenGL ES is slightly larger than Bob's bounding rectangle.

This separation of attributes allows us to apply our Model-View-Controller (MVC) pattern again.

- On the model side we simply have Bob's physical attributes, composed of his position, scale, rotation, velocity, acceleration, and bounding shape. Bob's position, scale, and orientation govern where his bounding shape is located in world space.
- The view just takes Bob's graphical representation (e.g., the two texture-mapped triangles defined in model space) and renders them at their world space position according to Bob's position, rotation, and scale. Here we can use the OpenGL ES matrix operations as we did previously.
- The controller is responsible for updating Bob's physical attributes according to user input (e.g., a left button press could move him to the left), and according to physical forces, such as gravitational acceleration (like we applied to the cannonball in the previous section).

Of course, there's some correspondence between Bob's bounding shape and his graphical representation in the texture, as we base the bounding shape on that graphical representation. Our MVC pattern is thus not entirely clean, but we can live with that.

Broad-Phase and Narrow-Phase Collision Detection

We still don't know how to check for collisions between our objects and their bounding shapes. There are two phases of collision detection:

Broad phase: In this phase we try to figure out which objects can potentially collide. Imagine having 100 objects that could each collide with each other. We'd need to perform 100 × 100 / 2 overlap tests if we chose to naively test each object against each other object. This naïve overlap testing approach is of $O(n^2)$ asymptotic complexity, meaning it would take n^2 steps to complete (it actually finished in half that many steps, but the asymptotic complexity leaves out any constants). In a good, non-brute-force broad phase, we try to figure out which pairs of objects are actually in danger of colliding. Other pairs (e.g., two objects that are too far apart for a collision to happen) will not be checked. We can reduce the computational load this way, as narrow-phase testing is usually pretty expensive.

Narrow phase: Once we know which pairs of objects can potentially collide, we test whether they really collide or not by doing an overlap test of their bounding shapes.

Let's focus on the narrow phase first and leave the broad phase for later.

Narrow Phase

Once we are done with the broad phase, we have to check whether the bounding shapes of the potentially colliding objects overlap. I mentioned earlier that we have a couple of options for bounding shapes. Triangle meshes are the most computationally expensive and cumbersome to create. It turns out that we can get away with bounding rectangles and bounding circles in most 2D games, so that's what we'll concentrate on here.

Circle Collision

Bounding circles are the cheapest way to check whether two objects collide. Let's define a simple `Circle` class. Listing 8–4 shows the code.

Listing 8–4. *Circle.java, a Simple Circle Class*

```
package com.badlogic.androidgames.framework.math;

public class Circle {
    public final Vector2 center = new Vector2();
    public float radius;

    public Circle(float x, float y, float radius) {
        this.center.set(x,y);
        this.radius = radius;
    }
}
```

We just store the center as a `Vector2` and the radius as a simple float. How can we check whether two circles overlap? Look at Figure 8–13.

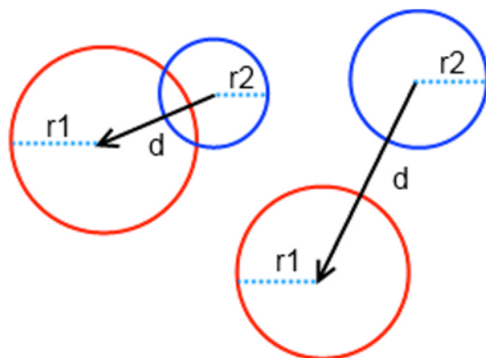

Figure 8–13. *Two circles overlapping (left), and two circles not overlapping (right)*

It's really simple and computationally efficient. All we need to do is figure out the distance between the two centers. If the distance is greater then the sum of the two radii, then we know the two circles do not overlap. In code this will look as follows:

```
public boolean overlapCircles(Circle c1, Circle c2) {
    float distance = c1.center.dist(c2.center);
    return distance <= c1.radius + c2.radius;
}
```

We first measure the distance between the two centers and then check if the distance is smaller or equal to the sum of the radii.

We have to take a square root in the `Vector2.dist()` method. That's unfortunate, as taking the square root is a costly operation. Can we make this faster? Yes we can—all we need to do is reformulate our condition:

```
sqrt(dist.x × dist.x + dist.y × dist.y) <= radius1 + radius2
```

We can get rid of the square root by exponentiating both sides of the inequality, as follows:

```
dist.x × dist.x + dist.y × dist.y <= (radius1 + radius2) × (radius1 + radius2)
```

We trade the square root for an additional addition and multiplication on the right side. That's a lot better. Let's create a `Vector2.distSquared()` function that will return the squared distance between two vectors:

```
public float distSquared(Vector2 other) {
    float distX = this.x - other.x;
    float distY = this.y - other.y;
    return distX*distX + distY*distY;
}
```

The `overlapCircles()` method then becomes the following:

```
public boolean overlapCircles(Circle c1, Circle c2) {
    float distance = c1.center.distSquared(c2.center);
    float radiusSum = c1.radius + c2.radius;
    return distance <= radiusSum * radiusSum;
}
```

Rectangle Collision

Let's move on to rectangles. First we need a class that can represent a rectangle. We previously said we want a rectangle to be defined by its lower-left corner position plus its width and height. We do just that in Listing 8–5.

Listing 8–5. *Rectangle.java, a Rectangle Class*

```
package com.badlogic.androidgames.framework.math;

public class Rectangle {
    public final Vector2 lowerLeft;
    public float width, height;

    public Rectangle(float x, float y, float width, float height) {
        this.lowerLeft = new Vector2(x,y);
        this.width = width;
        this.height = height;
    }
}
```

We store the lower-left corner's position in a `Vector2` and the width and height in two floats. How can we check whether two rectangles overlap? Figure 8–14 should give you a hint.

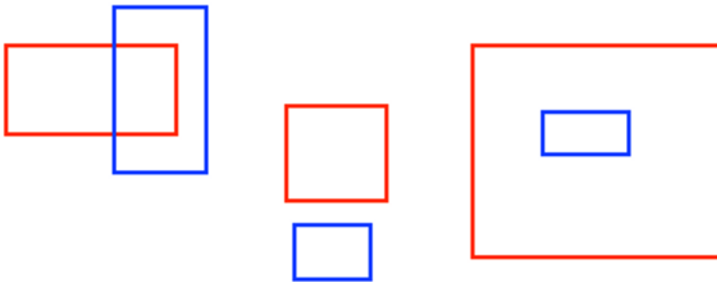

Figure 8–14. *Lots of overlapping and nonoverlapping rectangles*

The first two cases of partial overlap and nonoverlap are easy. The last one is a surprise. A rectangle can of course be completely contained in another rectangle. That can happen in the case of circles as well. However, our circle overlap test will return the correct result if one circle is contained in the other circle.

Checking for overlap in the rectangle case looks complex at first. However, we can create a very simple test if we invoke a little logic. Here's the simplest method to check for overlap between two rectangles:

```
public boolean overlapRectangles(Rectangle r1, Rectangle r2) {
    if(r1.lowerLeft.x < r2.lowerLeft.x + r2.width &&
       r1.lowerLeft.x + r1.width > r2.lowerLeft.x &&
       r1.lowerLeft.y < r2.lowerLeft.y + r2.height &&
       r1.lowerLeft.y + r1.height > r2.lowerLeft.y)
        return true;
```

```
    else
        return false;
}
```

This looks a little bit confusing at first sight, so let's go over each condition. The first condition states that the left edge of the first rectangle must be to the left of the right edge of the second rectangle. The next condition states that the right edge of the first rectangle must be to the right of the left edge of the second rectangle. The other two conditions state the same for the top and bottom edges of the rectangles. If all these conditions are met, then the two rectangles overlap. Double-check this with Figure 8–14. It also covers the containment case.

Circle/Rectangle Collision

Can we check for overlap between a circle and a rectangle? Yes we can. However, it is a little more involved. Take a look at Figure 8–15.

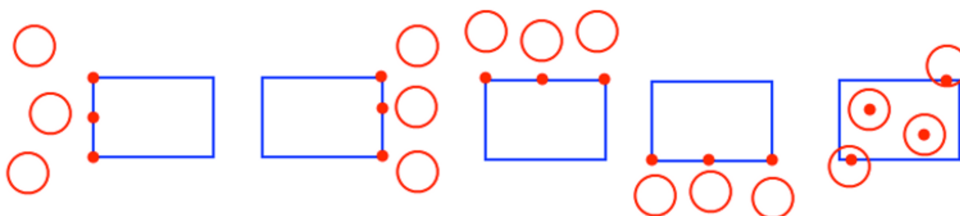

Figure 8–15. *Overlap-testing a circle and a rectangle by finding the closest point on/in the rectangle to the circle*

The overall strategy for testing for overlap between a circle and a rectangle goes like this:

- Find the closest x-coordinate on or in the rectangle to the circle's center. This coordinate can either be a point on the left or right edge of the rectangle, unless the circle center is contained in the rectangle, in which case the closest x-coordinate is the circle center's x-coordinate.
- Find the closest y-coordinate on or in the rectangle to the circle's center. This coordinate can either be a point on the top or bottom edge of the rectangle, unless the circle center is contained in the rectangle, in which case the closest y-coordinate is the circle center's y-coordinate.
- If the point composed of the closest x- and y-coordinates is within the circle, the circle and rectangle overlap.

While not depicted in Figure 8–15, this method also works for circles that completely contain the rectangle. Let's code it up:

```
public boolean overlapCircleRectangle(Circle c, Rectangle r) {
    float closestX = c.center.x;
    float closestY = c.center.y;
```

```
    if(c.center.x < r.lowerLeft.x) {
        closestX = r.lowerLeft.x;
    }
    else if(c.center.x > r.lowerLeft.x + r.width) {
        closestX = r.lowerLeft.x + r.width;
    }

    if(c.center.y < r.lowerLeft.y) {
        closestY = r.lowerLeft.y;
    }
    else if(c.center.y > r.lowerLeft.y + r.height) {
        closestY = r.lowerLeft.y + r.height;
    }

    return c.center.distSquared(closestX, closestY) < c.radius * c.radius;
}
```

The description looked a lot scarier than the implementation. We determine the closest point on the rectangle to the circle, and then simply check whether the point lies inside the circle. If that's the case, there is an overlap between the circle and the rectangle.

Note that I added an overloaded `distSquared()` method to `Vector2` that takes two float arguments instead of another `Vector2`. I did the same for the `dist()` function.

Putting It All Together

Checking whether a point lies inside a circle or rectangle can be useful too. Let's code up two more methods and put them into a class called `OverlapTester`, together with the other three methods we just defined. Listing 8–6 shows the code.

Listing 8–6. *OverlapTester.java; Testing Overlap Between Circles, Rectangles, and Points*

```
package com.badlogic.androidgames.framework.math;

public class OverlapTester {
    public static boolean overlapCircles(Circle c1, Circle c2) {
        float distance = c1.center.distSquared(c2.center);
        float radiusSum = c1.radius + c2.radius;
        return distance <= radiusSum * radiusSum;
    }

    public static boolean overlapRectangles(Rectangle r1, Rectangle r2) {
        if(r1.lowerLeft.x < r2.lowerLeft.x + r2.width &&
           r1.lowerLeft.x + r1.width > r2.lowerLeft.x &&
           r1.lowerLeft.y < r2.lowerLeft.y + r2.height &&
           r1.lowerLeft.y + r1.height > r2.lowerLeft.y)
            return true;
        else
            return false;
    }

    public static boolean overlapCircleRectangle(Circle c, Rectangle r) {
        float closestX = c.center.x;
        float closestY = c.center.y;
```

```
        if(c.center.x < r.lowerLeft.x) {
            closestX = r.lowerLeft.x;
        }
        else if(c.center.x > r.lowerLeft.x + r.width) {
            closestX = r.lowerLeft.x + r.width;
        }

        if(c.center.y < r.lowerLeft.y) {
            closestY = r.lowerLeft.y;
        }
        else if(c.center.y > r.lowerLeft.y + r.height) {
            closestY = r.lowerLeft.y + r.height;
        }

        return c.center.distSquared(closestX, closestY) < c.radius * c.radius;
    }

    public static boolean pointInCircle(Circle c, Vector2 p) {
        return c.center.distSquared(p) < c.radius * c.radius;
    }

    public static boolean pointInCircle(Circle c, float x, float y) {
        return c.center.distSquared(x, y) < c.radius * c.radius;
    }

    public static boolean pointInRectangle(Rectangle r, Vector2 p) {
        return r.lowerLeft.x <= p.x && r.lowerLeft.x + r.width >= p.x &&
               r.lowerLeft.y <= p.y && r.lowerLeft.y + r.height >= p.y;
    }

    public static boolean pointInRectangle(Rectangle r, float x, float y) {
        return r.lowerLeft.x <= x && r.lowerLeft.x + r.width >= x &&
               r.lowerLeft.y <= y && r.lowerLeft.y + r.height >= y;
    }
}
```

Sweet, now we have a fully functional 2D math library we can use for all our little physics models and collision detection. Let's talk about the broad phase in a little more detail now.

Broad Phase

So how can we achieve the magic that the broad phase promises us? Look at Figure 8–16, which shows a typical Super Mario Brothers scene.

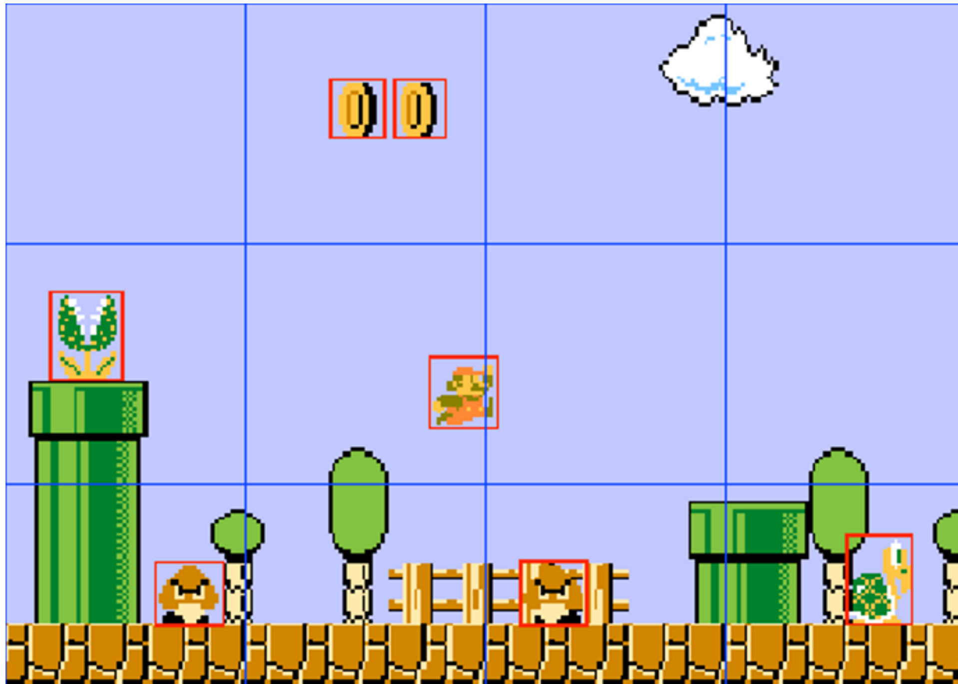

Figure 8–16. *Super Mario and his enemies. Boxes around objects are their bounding rectangles; the big boxes make up a grid imposed on the world.*

Can you already guess what we could do to eliminate some checks? The blue grid in Figure 8–16 represents cells we partition our world with. Each cell has the exact same size, and the whole world is covered in cells. Mario is currently in two of those cells, and the other objects Mario could potentially collide with are in different cells. We thus don't need to check for any collisions, as Mario is not in the same cells as any of the other objects in the scene. All we need to do is the following:

- Update all objects in the world based on our physics and controller step.
- Update the position of each bounding shape of each object according to the object's position. We can of course also include the orientation and scale as well here.
- Figure out which cell or cells each object is contained in based on its bounding shape, and add it to the list of objects contained in those cells.
- Check for collisions, but only between object pairs that can collide (e.g., Goombas don't collide with other Goombas) and are in the same cell.

This is called a *spatial hash grid* broad phase, and it is very easy to implement. The first thing we have to define is the size of each cell. This is highly dependent on the scale and units we use for our game's world.

An Elaborate Example

Let's develop the spatial hash grid broad phase based on our last cannonball example. We will completely rework it to incorporate everything covered in this section so far. In addition to the cannon and the ball, we also want to have targets to fire at. We'll make our lives easy and just use squares of size 0.5×0.5 meters as targets. These squares don't move; they're static. Our cannon is static as well. The only thing that moves is the cannonball itself. We can generally categorize objects in our game world as static objects or dynamic objects. So let's devise a class that can represent such objects.

GameObject, DynamicGameObject, and Cannon

Let's start with the static case, or base case, in Listing 8–7.

Listing 8–7. *GameObject.java, a Static Game Object with a Position and Bounds*

```
package com.badlogic.androidgames.gamedev2d;

import com.badlogic.androidgames.framework.math.Rectangle;
import com.badlogic.androidgames.framework.math.Vector2;

public class GameObject {
    public final Vector2 position;
    public final Rectangle bounds;

    public GameObject(float x, float y, float width, float height) {
        this.position = new Vector2(x,y);
        this.bounds = new Rectangle(x-width/2, y-height/2, width, height);
    }
}
```

Every object in our game has a position that coincides with its center. Additionally we let each object have a single bounding shape—a rectangle in this case. In our constructor we set the position and bounding rectangle (which is centered around the center of the object) according to the parameters.

For dynamic objects, that is, objects which move, we also need to keep track of their velocity and acceleration (if they're actually accelerated by themselves—e.g., via an engine or thruster). Listing 8–8 shows the code for the `DynamicGameObject` class.

Listing 8–8. *DynamicGameObject.java: Extending the GameObject with a Velocity and Acceleration Vector*

```
package com.badlogic.androidgames.gamedev2d;

import com.badlogic.androidgames.framework.math.Vector2;

public class DynamicGameObject extends GameObject {
    public final Vector2 velocity;
    public final Vector2 accel;

    public DynamicGameObject(float x, float y, float width, float height) {
        super(x, y, width, height);
        velocity = new Vector2();
```

```
        accel = new Vector2();
    }
}
```

We just extend the GameObject class to inherit the position and bounds members. Additionally we create vectors for the velocity and acceleration. A new dynamic game object will have zero velocity and acceleration after it has been initialized.

In our cannonball example we have the cannon, the cannonball, and the targets. The cannonball is a DynamicGameObject, as it moves according to our simple physics model. The targets are static and can be implemented by using the standard GameObject. The cannon itself can also be implemented via the GameObject class. We will derrive a Cannon class from the GameObject class and add a field storing the cannon's current angle. Listing 8–9 shows the code.

Listing 8–9. *Cannon.java: Extending the GameObject with an Angle*

```
package com.badlogic.androidgames.gamedev2d;

public class Cannon extends GameObject {
    public float angle;

    public Cannon(float x, float y, float width, float height) {
        super(x, y, width, height);
        angle = 0;
    }
}
```

That nicely encapsulates all the data needed to represent an object in our cannon world. Every time we need a special kind of object, like the cannon, we can simply derive from GameObject if it is a static object, or DynamicGameObject if it has a velocity and acceleration.

> **NOTE:** The overuse of inheritance can lead to severe headaches and very ugly code architecture. Do not use it just for the sake of using it. The simple class hierarchy just used is OK, but we shouldn't let it go a lot deeper (e.g., by extending Cannon). There are alternative representations of game objects that do away with all inheritance by composition. For our purposes, simple inheritance is more than enough, though. If you are interested in other representations, search for "composites" or "mixins" on the Web.

The Spatial Hash Grid

Our cannon will be bounded by a rectangle of 1×1 meters, the cannonball will have a bounding rectangle of 0.2×0.2 meters, and the targets will each have a bounding rectangle of 0.5×0.5 meters. The bounding rectangles are centered around each object's positions to make our lives a little easier.

When our cannon example starts up, we'll simply place a number of targets at random positions. Here's how we could set up the objects in our world:

```
Cannon cannon = new Cannon(0, 0, 1, 1);
DynamicGameObject ball = new DynamicGameObject(0, 0, 0.2f, 0.2f);
GameObject[] targets = new GameObject[NUM_TARGETS];
for(int i = 0; i < NUM_TARGETS; i++) {
    targets[i] = new GameObject((float)Math.random() * WORLD_WIDTH,
                                (float)Math.random() * WORLD_HEIGHT,
                                0.5f, 0.5f);
}
```

The constants `WORLD_WIDTH` and `WORLD_HEIGHT` define the size of our game world. Everything should happen inside the rectangle bounded by (0,0) and (`WORLD_WIDTH`,`WORLD_HEIGHT`). Figure 8–17 shows a little mock-up of our game world so far.

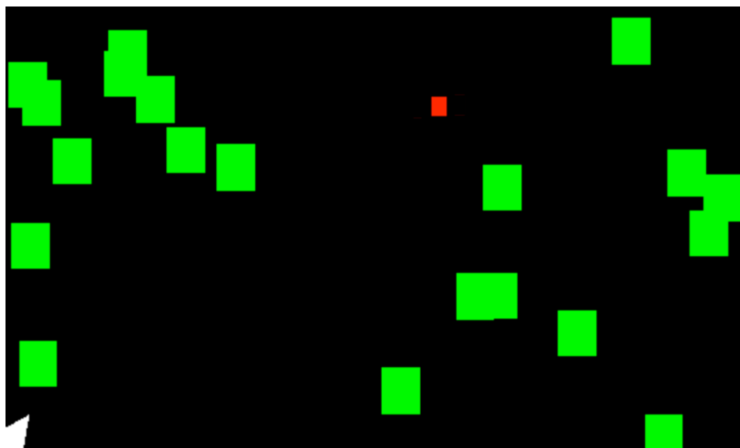

Figure 8–17. *A mock-up of our game world*

Our world will look like this later on, but for now, let's overlay a spatial hash grid. How big should the cells of the hash grid be? There's no silver bullet, but I tend to choose it to be five times bigger than the biggest object in the scene. In our example, the biggest object is the cannon, but we do not collide anything with the cannon. So let's base the grid size on the next biggest objects in our scene, the targets. These are 0.5×0.5 meters in size. A grid cell should thus have a size of 2.5×2.5 meters. Figure 8–18 shows the grid overlaid onto our world.

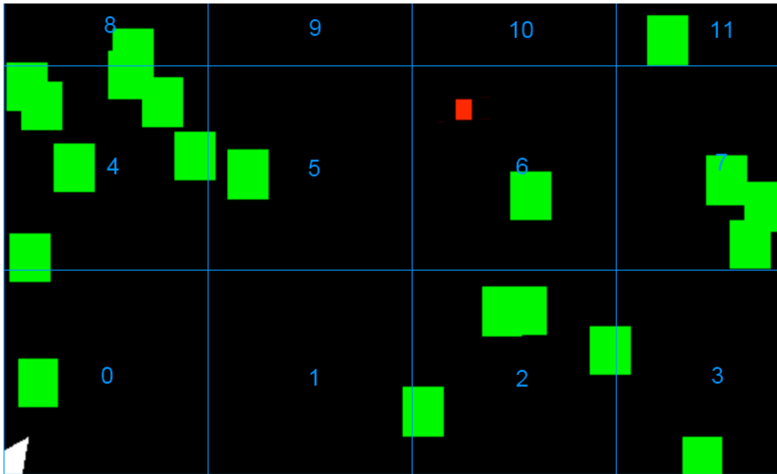

Figure 8–18. *Our cannon world overlaid with a spatial hash grid consisting of 12 cells*

We have a fixed number of cells—in the case of the cannon world, 12 cells to be exact. We give each cell a unique number, starting at the bottom-left cell, which gets the ID 0. Note that the top cells actually extend outside of our world. This is not a problem; we just need to make sure all our objects stay inside the boundaries of our world.

What we want to do is figure out which cell(s) an object belongs to. Ideally we want to calculate the IDs of the cells the object is contained in. This allows us to use the following simple data structure to store our cells:

```
List<GameObject>[] cells;
```

That's right, we represent each cell as a list of `GameObjects`. The spatial hash grid itself is then just composed of an array of lists of `GameObjects`.

Let's think about how we can figure out the IDs of the cells an object is contained in. Figure 8–18 shows a couple of targets that span two cells. In fact, a small object can span up to four cells, and an object bigger than a grid cell can span even more than four cells. We can make sure this never happens by choosing our grid cell size to be a multiple of the size of the biggest object in our game. That leaves us with the possibility of one object being contained in at most four cells.

To calculate the cell IDs for an object, we can simply take the four corner points of its bounding rectangle and check which cell each corner point is in. Determining the cell that a point is in is easy—we just need to divide its coordinates by the cell width first. Say we have a point at (3,4) and a cell size of 2.5×2.5 meters. The point would be in the cell with ID 5 in Figure 8–18.

We can divide each the point's coordinates by the cell size to get 2D integer coordinates , as follows:

```
cellX = floor(point.x / cellSize) = floor(3 / 2.5) = 1
cellY = floor(point.y / cellSize) = floor(4 / 2.5) = 1
```

And from these cell coordinates, we can easily get the cell ID:

```
cellId = cellX + cellY × cellsPerRow = 1 + 1 × 4 = 5
```

The constant `cellsPerRow` is simply the number of cells we need to cover our world with cells on the x-axis:

```
cellsPerRow = ceil(worldWidth / cellSize) = ceil(9.6 / 2.5) = 4
```

We can calculate the number of cells needed per column like this:

```
cellsPerColumn = ceil(worldHeight / cellSize) = ceil(6.4 / 2.5) = 3
```

Based on this we can implement the spatial hash grid rather easily. We set up it up by giving it the world's size and the desired cell size. We assume that all the action is happening in the positive quadrant of the world. This means that all x- and y-coordinates of points in the world will be positive. That's a constraint we can accept.

From the parameters, the spatial hash grid can figure out how many cells it needs (`cellsPerRow` × `cellsPerColumn`). We can also add a simple method to insert an object into the grid that will use the object's boundaries to determine the cells it is contained in. The object will then be added to each cell's list of objects that it contains. In case one of the corner points of the bounding shape of the object is outside of the grid, we'll just ignore that corner point.

We will reinsert every object into the spatial hash grid each frame after we update its position. However, there are objects in our cannon world that don't move, so inserting them anew each frame is very wasteful. We'll thus make a distinction between dynamic objects and static objects by storing two lists per cell. One will be updated each frame and only hold moving objects, and the other will be static and only modified when a new static object is inserted.

Finally we need a method that returns a list of objects in the cells of an object we'd like to collide with other objects. All this method will do is check which cells the object in question is in, retrieve the list of dynamic and static objects in those cells, and return them to the caller. We'll of course have to make sure that we don't return any duplicates, which can happen if an object is in multiple cells.

Listing 8–10 shows the code (well, most of it). We'll discuss the `SpatialHashGrid.getCellIds()` method in a minute, as it is a little involved.

Listing 8–10. *Excerpt from SpatialHashGrid.java: A Spatial Hash Grid Implementation*

```java
package com.badlogic.androidgames.framework.gl;

import java.util.ArrayList;
import java.util.List;

import com.badlogic.androidgames.gamedev2d.GameObject;

import android.util.FloatMath;

public class SpatialHashGrid {
    List<GameObject>[] dynamicCells;
    List<GameObject>[] staticCells;
    int cellsPerRow;
    int cellsPerCol;
```

```
float cellSize;
int[] cellIds = new int[4];
List<GameObject> foundObjects;
```

As discussed we store two cell lists, one for dynamic and one for static objects. We also store the cells per row and column so we can later decide whether a point we check is inside or outside of the world. The cell size needs to be stored as well. The `cellIds` array is a working array that we'll use to temporarily store the four cell IDs a `GameObject` is contained in. If it is only contained in one cell, then only the first element of the array will be set to the cell ID of the cell that contains the object entirely. If the object is contained in two cells, then the first two elements of that array will hold the cell ID, and so on. To indicate the number of cell IDs we set all "empty" elements of the array to –1. The `foundObjects` list is also a working list, which we'll return upon a call to `getPotentialColliders()`. Why do keep those two members instead of instantiating a new array and list each time one is needed? Remember the garbage collector monster.

```
@SuppressWarnings("unchecked")
public SpatialHashGrid(float worldWidth, float worldHeight, float cellSize) {
    this.cellSize = cellSize;
    this.cellsPerRow = (int)FloatMath.ceil(worldWidth/cellSize);
    this.cellsPerCol = (int)FloatMath.ceil(worldHeight/cellSize);
    int numCells = cellsPerRow * cellsPerCol;
    dynamicCells = new List[numCells];
    staticCells = new List[numCells];
    for(int i = 0; i < numCells; i++) {
        dynamicCells[i] = new ArrayList<GameObject>(10);
        staticCells[i] = new ArrayList<GameObject>(10);
    }
    foundObjects = new ArrayList<GameObject>(10);
}
```

The constructor of that class takes the world's size and the desired cell size. From those arguments we calculate how many cells are needed, and instantiate the cell arrays and the lists holding the objects contained in each cell. We also initialize the `foundObjects` list here. All the `ArrayLists` we instantiate will have an initial capacity of ten `GameObjects`. We do this to avoid memory allocations. The assumption is that it is unlikely that one single cell will contain more than ten `GameObjects`. As long as that is true, the arrays don't need to be resized.

```
public void insertStaticObject(GameObject obj) {
    int[] cellIds = getCellIds(obj);
    int i = 0;
    int cellId = -1;
    while(i <= 3 && (cellId = cellIds[i++]) != -1) {
        staticCells[cellId].add(obj);
    }
}

public void insertDynamicObject(GameObject obj) {
    int[] cellIds = getCellIds(obj);
    int i = 0;
    int cellId = -1;
    while(i <= 3 && (cellId = cellIds[i++]) != -1) {
```

```
            dynamicCells[cellId].add(obj);
        }
    }
```

Next up are the methods `insertStaticObject()` and `insertDynamicObject()`. They calculate the IDs of the cells that the object is contained in via a call to `getCellIds()`, and insert the object into the appropriate lists accordingly. The `getCellIds()` method will actually fill the `cellIds` member array.

```
    public void removeObject(GameObject obj) {
        int[] cellIds = getCellIds(obj);
        int i = 0;
        int cellId = -1;
        while(i <= 3 && (cellId = cellIds[i++]) != -1) {
            dynamicCells[cellId].remove(obj);
            staticCells[cellId].remove(obj);
        }
    }
```

We also have a `removeObject()` method, which we'll use to figure out what cells the object is in and then delete it from the dynamic and static lists accordingly. This will be needed when a game object dies, for example.

```
    public void clearDynamicCells(GameObject obj) {
        int len = dynamicCells.length;
        for(int i = 0; i < len; i++) {
            dynamicCells[i].clear();
        }
    }
```

The `clearDynamicCells()` method will be used to clear all dynamic cell lists. We need to call this each frame before we reinsert the dynamic objects, as discussed earlier.

```
    public List<GameObject> getPotentialColliders(GameObject obj) {
        foundObjects.clear();
        int[] cellIds = getCellIds(obj);
        int i = 0;
        int cellId = -1;
        while(i <= 3 && (cellId = cellIds[i++]) != -1) {
            int len = dynamicCells[cellId].size();
            for(int j = 0; j < len; j++) {
                GameObject collider = dynamicCells[cellId].get(j);
                if(!foundObjects.contains(collider))
                    foundObjects.add(collider);
            }

            len = staticCells[cellId].size();
            for(int j = 0; j < len; j++) {
                GameObject collider = staticCells[cellId].get(j);
                if(!foundObjects.contains(collider))
                    foundObjects.add(collider);
            }
        }
        return foundObjects;
    }
```

Finally there's the `getPotentialColliders()` method. It takes an object and returns a list of neighboring objects that are contained in the same cells as that object. We use the working list `foundObjects` to store the list of found objects. Again, we do not want to instantiate a new list each time this method is called. All we do is figure out which cells the object passed to the method is in. We then simply add all dynamic and static objects found in those cells to the `foundObjects` list and make sure that there are no duplicates. Using `foundObjects.contains()` to check for duplicates is of course a little suboptimal. But given that the number of found objects will never be large, it is OK to use it in this case. If you run into performance problems, then this is your number one candidate to optimize. Sadly, that's not trivial, however. We could use a `Set`, of course, but that allocates new objects internally each time we add an object to it. For now, we'll just leave it as it is, knowing that we can come back to it should anything go wrong performance-wise.

The method I left out is `SpatialHashGrid.getCellIds()`. Listing 8–11 shows its code. Don't be afraid, it just looks menacing.

Listing 8–11. *The Rest of SpatialHashGrid.java: Implementing getCellIds()*

```
    public int[] getCellIds(GameObject obj) {
        int x1 = (int)FloatMath.floor(obj.bounds.lowerLeft.x / cellSize);
        int y1 = (int)FloatMath.floor(obj.bounds.lowerLeft.y / cellSize);
        int x2 = (int)FloatMath.floor((obj.bounds.lowerLeft.x + obj.bounds.width) /
cellSize);
        int y2 = (int)FloatMath.floor((obj.bounds.lowerLeft.y + obj.bounds.height) /
cellSize);

        if(x1 == x2 && y1 == y2) {
            if(x1 >= 0 && x1 < cellsPerRow && y1 >= 0 && y1 < cellsPerCol)
                cellIds[0] = x1 + y1 * cellsPerRow;
            else
                cellIds[0] = -1;
            cellIds[1] = -1;
            cellIds[2] = -1;
            cellIds[3] = -1;
        }
        else if(x1 == x2) {
            int i = 0;
            if(x1 >= 0 && x1 < cellsPerRow) {
                if(y1 >= 0 && y1 < cellsPerCol)
                    cellIds[i++] = x1 + y1 * cellsPerRow;
                if(y2 >= 0 && y2 < cellsPerCol)
                    cellIds[i++] = x1 + y2 * cellsPerRow;
            }
            while(i <= 3) cellIds[i++] = -1;
        }
        else if(y1 == y2) {
            int i = 0;
            if(y1 >= 0 && y1 < cellsPerCol) {
                if(x1 >= 0 && x1 < cellsPerRow)
                    cellIds[i++] = x1 + y1 * cellsPerRow;
                if(x2 >= 0 && x2 < cellsPerRow)
                    cellIds[i++] = x2 + y1 * cellsPerRow;
            }
```

```
            while(i <= 3) cellIds[i++] = -1;
        }
        else {
            int i = 0;
            int y1CellsPerRow = y1 * cellsPerRow;
            int y2CellsPerRow = y2 * cellsPerRow;
            if(x1 >= 0 && x1 < cellsPerRow && y1 >= 0 && y1 < cellsPerCol)
                cellIds[i++] = x1 + y1CellsPerRow;
            if(x2 >= 0 && x2 < cellsPerRow && y1 >= 0 && y1 < cellsPerCol)
                cellIds[i++] = x2 + y1CellsPerRow;
            if(x2 >= 0 && x2 < cellsPerRow && y2 >= 0 && y2 < cellsPerCol)
                cellIds[i++] = x2 + y2CellsPerRow;
            if(x1 >= 0 && x1 < cellsPerRow && y2 >= 0 && y2 < cellsPerCol)
                cellIds[i++] = x1 + y2CellsPerRow;
            while(i <= 3) cellIds[i++] = -1;
        }
        return cellIds;
    }

}
```

The first four lines of this method just calculate the cell coordinates of the bottom-left and top-right corners of the object's bounding rectangle. We already discussed this calculation earlier. To understand the rest of this method, we have to think about how an object can overlap grid cells. There are four possibilities:

- The object is contained in a single cell. The bottom-left and top-right corners of the bounding rectangle thus both have the same cell coordinates.
- The object overlaps two cells horizontally. The bottom-left corner is in one cell and the top-right corner is in the cell to the right.
- The object overlaps two cells vertically. The bottom-left corner is in one cell and the top-right corner is in the cell above.
- The object overlaps four cells. The bottom-left corner is in one cell, the bottom-right corner is in the cell to the right, the top-right corner is in the cell above that, and the top-left corner is in the cell above the first cell.

All this method does is make a special case for each of these possibilities. The first `if` statement checks for the single-cell case, the second `if` statement checks for the horizontal double-cell case, the third `if` statement checks for the vertical double-cell case, and the `else` block handles the case of the object overlapping four grid cells. In each of the four blocks we make sure that we only set the cell ID if the corresponding cell coordinates are within the world. And that's all there is to this method.

Now, the method looks like it would take a lot of computational power. And indeed it does, but less than its size would suggest. The most common case will be the first one, and processing that is pretty cheap. Can you see opportunities to optimize this method further?

Putting It All Together

Let's put all the knowledge we gathered in this section together to form a nice little example. We'll extend the cannon example of the last section as discussed a few pages back. We'll use a `Cannon` object for the cannon, a `DynamicGameObject` for the cannonball, and a number of `GameObjects` for the targets. Each target will have a size of 0.5×0.5 meters and be placed randomly in the world.

We want to be able to shoot those targets. For this we need collision detection. We could just loop over all targets and check them against the cannonball, but that would be boring. We'll use our fancy new `SpatialHashGrid` class to speed up finding the potentially colliding targets for the current ball position. We won't insert the ball or the cannon into the grid, though, as that wouldn't really gain us anything.

Since this example is already pretty big, we'll split it up into multiple listings. We'll call the test `CollisionTest` and the corresponding screen `CollisionScreen`. As always, we'll only look at the screen. Let's start with the members and the constructor, in Listing 8–12.

Listing 8–12. *Excerpt from CollisionTest.java: Members and Constructor*

```
class CollisionScreen extends Screen {
    final int NUM_TARGETS = 20;
    final float WORLD_WIDTH = 9.6f;
    final float WORLD_HEIGHT = 4.8f;
    GLGraphics glGraphics;
    Cannon cannon;
    DynamicGameObject ball;
    List<GameObject> targets;
    SpatialHashGrid grid;

    Vertices cannonVertices;
    Vertices ballVertices;
    Vertices targetVertices;

    Vector2 touchPos = new Vector2();
    Vector2 gravity = new Vector2(0,-10);

    public CollisionScreen(Game game) {
        super(game);
        glGraphics = ((GLGame)game).getGLGraphics();

        cannon = new Cannon(0, 0, 1, 1);
        ball = new DynamicGameObject(0, 0, 0.2f, 0.2f);
        targets = new ArrayList<GameObject>(NUM_TARGETS);
        grid = new SpatialHashGrid(WORLD_WIDTH, WORLD_HEIGHT, 2.5f);
        for(int i = 0; i < NUM_TARGETS; i++) {
            GameObject target = new GameObject((float)Math.random() * WORLD_WIDTH,
                                               (float)Math.random() * WORLD_HEIGHT,
                                               0.5f, 0.5f);
            grid.insertStaticObject(target);
            targets.add(target);
        }
```

```
        cannonVertices = new Vertices(glGraphics, 3, 0, false, false);
        cannonVertices.setVertices(new float[] { -0.5f, -0.5f,
                                                  0.5f, 0.0f,
                                                 -0.5f, 0.5f }, 0, 6);

        ballVertices = new Vertices(glGraphics, 4, 6, false, false);
        ballVertices.setVertices(new float[] { -0.1f, -0.1f,
                                                0.1f, -0.1f,
                                                0.1f,  0.1f,
                                               -0.1f,  0.1f }, 0, 8);
        ballVertices.setIndices(new short[] {0, 1, 2, 2, 3, 0}, 0, 6);

        targetVertices = new Vertices(glGraphics, 4, 6, false, false);
        targetVertices.setVertices(new float[] { -0.25f, -0.25f,
                                                  0.25f, -0.25f,
                                                  0.25f,  0.25f,
                                                 -0.25f,  0.25f }, 0, 8);
        targetVertices.setIndices(new short[] {0, 1, 2, 2, 3, 0}, 0, 6);
    }
```

We brought over a lot from the CannonGravityScreen. We start off with a couple of constant definitions, governing the number of targets and our world's size. Next we have the GLGraphics instance, as well as the objects for the cannon, the ball, and the targets, which we store in a list. We also have a SpatialHashGrid, of course. For rendering our world we need a few meshes: one for the cannon, one for the ball, and one we'll use to render each target. Remember that we only had a single rectangle in BobTest to render the 100 Bobs to the screen. We'll reuse that principle here as well, instead of having a single Vertices instance holding the triangles (rectangles) of our targets. The last two members are the same as in the CannonGravityTest. We use them to shoot the ball and apply gravity when the user touches the screen.

The constructor just does all the things we discussed already. We instantiate our world objects and meshes. The only interesting thing is that we also add the targets as static objects to the spatial hash grid.

Let's check out the next method of the CollisionTest class, in Listing 8–13.

Listing 8–13. *Excerpt from CollisionTest.java: The update() Method*

```
@Override
public void update(float deltaTime) {
    List<TouchEvent> touchEvents = game.getInput().getTouchEvents();
    game.getInput().getKeyEvents();

    int len = touchEvents.size();
    for (int i = 0; i < len; i++) {
        TouchEvent event = touchEvents.get(i);

        touchPos.x = (event.x / (float) glGraphics.getWidth())* WORLD_WIDTH;
        touchPos.y = (1 - event.y / (float) glGraphics.getHeight()) * WORLD_HEIGHT;

        cannon.angle = touchPos.sub(cannon.position).angle();

        if(event.type == TouchEvent.TOUCH_UP) {
            float radians = cannon.angle * Vector2.TO_RADIANS;
```

```
                float ballSpeed = touchPos.len() * 2;
                ball.position.set(cannon.position);
                ball.velocity.x = FloatMath.cos(radians) * ballSpeed;
                ball.velocity.y = FloatMath.sin(radians) * ballSpeed;
                ball.bounds.lowerLeft.set(ball.position.x - 0.1f, ball.position.y - 0.1f);
            }
        }

        ball.velocity.add(gravity.x * deltaTime, gravity.y * deltaTime);
        ball.position.add(ball.velocity.x * deltaTime, ball.velocity.y * deltaTime);
        ball.bounds.lowerLeft.add(ball.velocity.x * deltaTime, ball.velocity.y * deltaTime);

        List<GameObject> colliders = grid.getPotentialColliders(ball);
        len = colliders.size();
        for(int i = 0; i < len; i++) {
            GameObject collider = colliders.get(i);
            if(OverlapTester.overlapRectangles(ball.bounds, collider.bounds)) {
                grid.removeObject(collider);
                targets.remove(collider);
            }
        }
    }
```

As always, we first fetch the touch and key events, and only iterate over the touch events. The handling of touch events is nearly the same as in the `CannonGravityTest`. The only difference is that we use the `Cannon` object instead of the vectors we had in the old example, and we also reset the ball's bounding rectangle when the cannon is made ready to shoot on a touch-up event.

The next change is how we update the ball. Instead of straight vectors, we use the members of the `DynamicGameObject` that we instantiated for the ball. We neglect the `DynamicGameObject.acceleration` member and instead add our gravity to the ball's velocity. We also multiply the ball's speed by 2 so that the cannonball flies a little faster. The interesting thing is that we update not only the ball's position, but also the bounding rectangle's lower-left corner's position. This is crucial, as otherwise our ball would move but its bounding rectangle wouldn't. Why don't we just use the ball's bounding rectangle to store the ball's position? We might want to have multiple bounding shapes attached to an object. Which bounding shape would then hold the actual position of the object? Separating these two things is thus beneficial, and introduces only a little computational overhead. We could of course optimize this a little by only multiplying the velocity with the delta time once. The overhead would then boil down to two more additions—a small price to pay for the flexibility we gain.

The final portion of this method is our collision detection code. All we do is find the targets in the spatial hash grid that are in the same cells as our cannonball. We use the `SpatialHashGrid.getPotentialColliders()` method for this. Since the cells the ball is contained in are evaluated in that method directly, we do not need to insert the ball into the grid. Next we loop through all the potential colliders and check if there really is an overlap between the ball's bounding rectangle and a potential collider's bounding rectangle. If there is, we simply remove the target from the target list. Remember, we only added targets as static objects to the grid.

And those are our complete game mechanics. The last piece of the puzzle is the actual rendering, which shouldn't really surprise you. See the code in Listing 8–14.

Listing 8–14. *Excerpt from CollisionTest.java: The present() Method*

```
@Override
public void present(float deltaTime) {
    GL10 gl = glGraphics.getGL();
    gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
    gl.glMatrixMode(GL10.GL_PROJECTION);
    gl.glLoadIdentity();
    gl.glOrthof(0, WORLD_WIDTH, 0, WORLD_HEIGHT, 1, -1);
    gl.glMatrixMode(GL10.GL_MODELVIEW);

    gl.glColor4f(0, 1, 0, 1);
    targetVertices.bind();
    int len = targets.size();
    for(int i = 0; i < len; i++) {
        GameObject target = targets.get(i);
        gl.glLoadIdentity();
        gl.glTranslatef(target.position.x, target.position.y, 0);
        targetVertices.draw(GL10.GL_TRIANGLES, 0, 6);
    }
    targetVertices.unbind();

    gl.glLoadIdentity();
    gl.glTranslatef(ball.position.x, ball.position.y, 0);
    gl.glColor4f(1,0,0,1);
    ballVertices.bind();
    ballVertices.draw(GL10.GL_TRIANGLES, 0, 6);
    ballVertices.unbind();

    gl.glLoadIdentity();
    gl.glTranslatef(cannon.position.x, cannon.position.y, 0);
    gl.glRotatef(cannon.angle, 0, 0, 1);
    gl.glColor4f(1,1,1,1);
    cannonVertices.bind();
    cannonVertices.draw(GL10.GL_TRIANGLES, 0, 3);
    cannonVertices.unbind();
}
```

Nothing new here. As always, we set the projection matrix and viewport, and clear the screen first. Next we render all targets, reusing the rectangular model stored in `targetVertices`. This is essentially the same thing we did in `BobTest`, but this time we render targets instead. Next we render the ball and the cannon, as we did in the `CollisionGravityTest`.

The only thing to note here is that I changed the drawing order so that the ball will always be above the targets and the cannon will always be above the ball. I also colored the targets green with a call to `glColor4f()`.

The output of this little test is exactly the same as in Figure 8–17, so I'll spare you the repetition. When you fire the cannonball, it will plow through the field of targets. Any target that gets hit by the ball will be removed from the world.

This example could actually be a nice game if we polish it up a little and add some motivating game mechanics. Can you think of additions? I suggest you play around with the example a little to get a feeling for all the new tools we have developed over the course of the last couple of pages.

There are a few more things I'd like to discuss in this chapter: cameras, texture atlases, and sprites. These use graphics-related tricks that are independent of our model of the game world. Let's get going!

A Camera in 2D

Up until now, we haven't had the concept of a camera in our code; we've only had the definition of our view frustum via glOrthof(), like this:

```
gl.glMatrixMode(GL10.GL_PROJECTION);
gl.glLoadIdentity();
gl.glOrthof(0, FRUSTUM_WIDTH, 0, FRUSTUM_HEIGHT, 1, -1);
```

From Chapter 6 we know that the first two parameters define the x-coordinates of the left and right edges of our frustum in the world, the next two parameters define the y-coordinates of the bottom and top edges of the frustum, and the last two parameters define the near and far clipping planes. Figure 8–19 shows that frustum again.

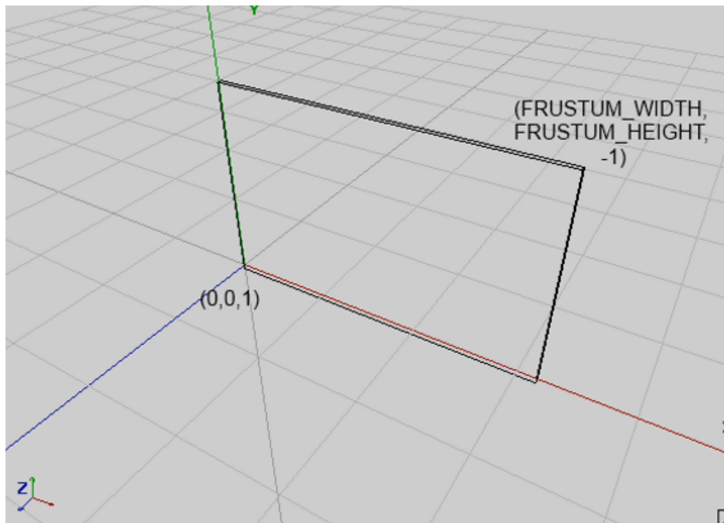

Figure 8–19. *The view frustum for our 2D world, again*

So we only see the region (0,0,1) to (FRUSTUM_WIDTH, FRUSTUM_HEIGHT,–1) of our world. Wouldn't it be nice if we could move the frustum? Say, to the left? Of course that would be nice, and it is dead simple as well:

```
gl.glOrthof(x, x + FRUSTUM_WIDTH, 0, FRUSTUM_HEIGHT, 1, -1);
```

In this case, x is just some offset we can define. We can of course also move on the x- and y-axes:

```
gl.glOrthof(x, x + FRUSTUM_WIDTH, y, y + FRUSTUM_HEIGHT, 1, -1);
```

Figure 8–20 shows what that means.

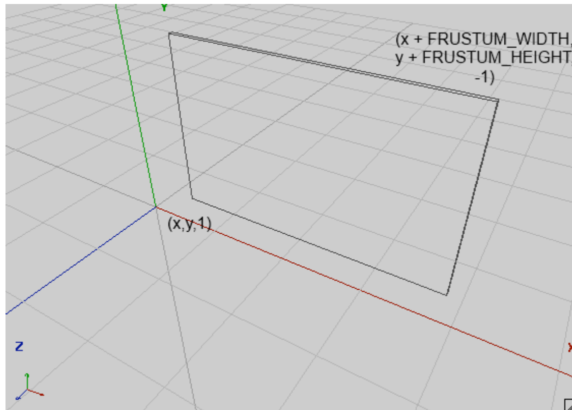

Figure 8–20. *Moving the frustum around*

By this we simply specify the bottom-left corner of our view frustum in the world space. This is already sufficient to implement a freely movable 2D camera. But we can do better. What about not specifying the bottom-left corner of the view frustum with x and y, but instead specifying the center of the view frustum? That way we could easily center our view frustum on an object at a specific location—say, the cannonball from our preceding example:

```
gl.glOrthof(x - FRUSTUM_WIDTH / 2, x + FRUSTUM_WIDTH / 2, y - FRUSTUM_HEIGHT / 2, y +
FRUSTUM_HEIGHT / 2, 1, -1);
```

Figure 8–21 shows what this looks like.

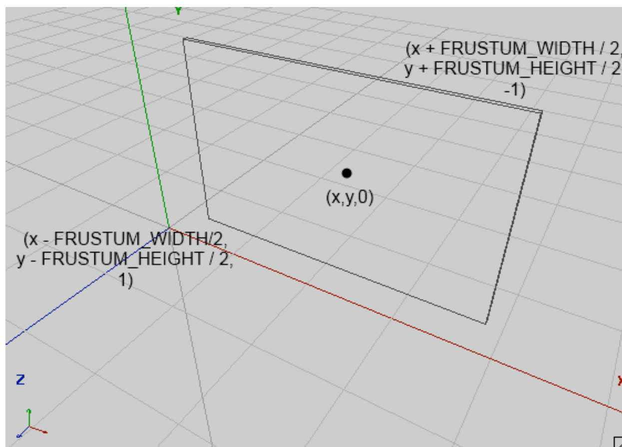

Figure 8–21. *Specifying the view frustum in terms of its center*

That's still not all we can do with `glOrthof()`. What about zooming? Let's think about this for a little while. We know that via `glViewportf()` we can tell OpenGL ES what

portion of our screen to render the contents of our view frustum to. OpenGL ES will automatically stretch and scale the output to align with the viewport. Now, if we make the width and height of our view frustum smaller, we will simply show a smaller region of our world on the screen. That's zooming in. If we make the frustum bigger, we'll show more of our world—that's zooming out. We can therefore introduce a zoom factor and multiply it by our frustum's width and height to zoom in an out. A factor of 1 will show us the world as in Figure 8–21, using the normal frustum width and height. A factor smaller than 1 will zoom in on the center of our view frustum. And a factor bigger than 1 will zoom out, showing us more of our world (e.g., setting the zoom factor to 2 will show us twice as much of our world). Here's how we can use `glOrthof()` to do that for us:

```
gl.glOrthof(x - FRUSTUM_WIDTH / 2 * zoom, x + FRUSTUM_WIDTH / 2 * zoom, y -
FRUSTUM_HEIGHT / 2 * zoom, y + FRUSTUM_HEIGHT / 2 * zoom, 1, -1);
```

Dead simple! We can now create a camera class that has a position it is looking at (the center of the view frustum), a standard frustum width and height, and a zoom factor that makes the frustum smaller or bigger, thereby showing us either less of our world (zooming in) or more of our world (zooming out). Figure 8–22 shows a view frustum with a zoom factor of 0.5 (the inner gray box), and one with a zoom factor of 1 (the outer, transparent box).

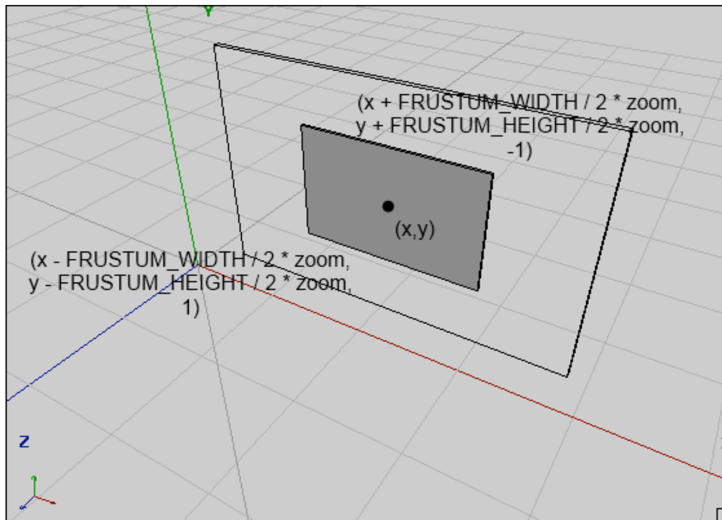

Figure 8–22. *Zooming by manipulating the frustum size*

To make our lives complete we should add one more thing. Imagine that we touch the screen and want to figure out what point in our 2D world we touched. We already did this a couple of times in our iteratively improving cannon examples. With a view frustum configuration that does not factor in the camera's position and zoom, as in Figure 8–19, we had the following equations (see the `update()` method of our cannon examples):

```
worldX = (touchX / Graphics.getWidth()) × FRUSTUM_WIDTH;
worldY = (1 - touchY / Graphics.getHeight()) × FRUSTUM_HEIGHT;
```

We first normalize the touch x- and y-coordinates to the range 0 to 1 by dividing by the screen's width and height, and then we scale them so that they are expressed in terms of our world space by multiplying them with the frustum's width and height. All we need to do is factor in the position of the view frustum as well as the zoom factor. Here's how we do that:

```
worldX = (touchX / Graphics.getWidth()) × FRUSTUM_WIDTH + x – FRUSTUM_WIDTH / 2;
worldY = (1 - touchY / Graphics.getHeight()) × FRUSTUM_HEIGHT + y – FRUSTUM_HEIGHT / 2;
```

Here, x and y are our camera's position in world space.

The Camera2D Class

Let's put all this together into a single class. We want it to store the camera's position, the standard frustum width and height, and the zoom factor. We also want a convenience method that sets the viewport (always use the whole screen) and projection matrix correctly. Additionally we want a method that can translate touch coordinates to world coordinates. Listing 8–15 shows our new `Camera2D` class.

Listing 8–15. ***Camera2D.java, Our Shiny New Camera Class for 2D Rendering***

```
package com.badlogic.androidgames.framework.gl;

import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.framework.impl.GLGraphics;
import com.badlogic.androidgames.framework.math.Vector2;

public class Camera2D {
    public final Vector2 position;
    public float zoom;
    public final float frustumWidth;
    public final float frustumHeight;
    final GLGraphics glGraphics;
```

As discussed, we store the camera's position, frustum width and height, and zoom factor as members. The position and zoom factor are public, so we can easily manipulate them. We also need a reference to `GLGraphics` so we can get the up-to-date width and height of the screen in pixels for transforming touch coordinates to world coordinates.

```
    public Camera2D(GLGraphics glGraphics, float frustumWidth, float frustumHeight) {
        this.glGraphics = glGraphics;
        this.frustumWidth = frustumWidth;
        this.frustumHeight = frustumHeight;
        this.position = new Vector2(frustumWidth / 2, frustumHeight / 2);
        this.zoom = 1.0f;
    }
```

In the constructor we take a `GLGraphics` instance and the frustum's width and height at the zoom factor 1 as parameters. We store them and initialize the position of the camera to look at the center of the box bounded by (0,0,1) and (`frustumWidth`, `frustumHeight`,–1), as in Figure 8–19. The initial zoom factor is set to 1.

```
    public void setViewportAndMatrices() {
        GL10 gl = glGraphics.getGL();
        gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();
        gl.glOrthof(position.x - frustumWidth * zoom / 2,
                    position.x + frustumWidth * zoom/ 2,
                    position.y - frustumHeight * zoom / 2,
                    position.y + frustumHeight * zoom/ 2,
                    1, -1);
        gl.glMatrixMode(GL10.GL_MODELVIEW);
        gl.glLoadIdentity();
    }
```

The `setViewportAndMatrices()` method sets the viewport to span the whole screen, and sets the projection matrix in accordance with our camera's parameters, as discussed previously. At the end of the method we tell OpenGL ES that all further matrix operations are targeting the model view matrix and load an identity matrix. We will call this method each frame so we can start from a clean slate. No more direct OpenGL ES calls to set up our viewport and projection matrix.

```
    public void touchToWorld(Vector2 touch) {
        touch.x = (touch.x / (float) glGraphics.getWidth()) * frustumWidth * zoom;
        touch.y = (1 - touch.y / (float) glGraphics.getHeight()) * frustumHeight * zoom;
        touch.add(position).sub(frustumWidth * zoom / 2, frustumHeight * zoom / 2);
    }
}
```

The `touchToWorld()` method takes a `Vector2` instance containing touch coordinates and transforms the vector to world space. This is the same thing we just discussed; the only difference is that we use our fancy `Vector2` class.

An Example

Let's use the `Camera2D` class in our cannon example. I copied the `CollisionTest` file and renamed it `Camera2DTest`. I also renamed the `GLGame` class inside the file `Camera2DTest`, and renamed the `CollisionScreen` class `Camera2DScreen`. We'll just discuss the little changes we have to make to use our new `Camera2D` class.

The first thing we do is add a new member to the `Camera2DScreen` class:

```
Camera2D camera;
```

We initialize this member in the constructor as follows:

```
camera = new Camera2D(glGraphics, WORLD_WIDTH, WORLD_HEIGHT);
```

We just pass in our `GLGraphics` instance and the world's width and height, which we previously used as the frustum's width and height in our call to `glOrthof()`. All we need to do now is replace our direct OpenGL ES calls in the `present()` method, which looked like this:

```
gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
gl.glMatrixMode(GL10.GL_PROJECTION);
```

```
gl.glLoadIdentity();
gl.glOrthof(0, WORLD_WIDTH, 0, WORLD_HEIGHT, 1, -1);
gl.glMatrixMode(GL10.GL_MODELVIEW);
```

We replace them with this:

```
gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
camera.setViewportAndMatrices();
```

We still have to clear the framebuffer, of course, but all the other direct OpenGL ES calls are nicely hidden inside the `Camera2D.setViewportAndMatrices()` method. If you run that code, you'll see that nothing has changed. Everything works like before—all we did was make things a little nicer and more flexible.

We can also simplify the `update()` method of the test a little. Since we added the `Camera2D.touchToWorld()` method to the camera class, we might as well use it. We can replace this snippet from the update method:

```
touchPos.x = (event.x / (float) glGraphics.getWidth())* WORLD_WIDTH;
touchPos.y = (1 - event.y / (float) glGraphics.getHeight()) * WORLD_HEIGHT;
```

with this:

```
camera.touchToWorld(touchPos.set(event.x, event.y));
```

Neat, everything is nicely encapsulated now. But it would be very boring if we didn't use the features of our camera class to their full extent. Here's the plan: we want to have the camera look at the world in the "normal" way as long as the cannonball does not fly. That's easy; we're already doing that. We can determine whether the cannonball flies or not by checking whether the y-coordinate of its position is less than or equal to zero. Since we always apply gravity to the cannonball, it will of course fall even if we don't shoot it, so that's a cheap way to check matters.

Our new addition will come into effect when the cannonball is flying (when the y-coordinate is greater than zero). We want the camera to follow the cannonball. We can achieve this by simply setting the camera's position to the cannonball's position. That will always keep the cannonball in the center of the screen. We also want to try out our zooming functionality. Therefore well increase the zoom factor depending on the y-coordinate of the cannonball. The further away from zero, the higher the zoom factor. This will make the camera zoom out if the cannonball has a higher y-coordinate. Here's what we need to add at the end of the `update()` method in our test's screen:

```
if(ball.position.y > 0) {
    camera.position.set(ball.position);
    camera.zoom = 1 + ball.position.y / WORLD_HEIGHT;
} else {
    camera.position.set(WORLD_WIDTH / 2, WORLD_HEIGHT / 2);
    camera.zoom = 1;
}
```

As long as the y-coordinate of our ball is greater then zero, the camera will follow it and zoom out. We just add a value to the standard zoom factor of 1. That value is just the relation between the ball's y-position and the world's height. If the ball's y-coordinate is at `WORLD_HEIGHT`, the zoom factor will be 2, so we we'll see more of our world. The way I did this is really arbitrary; you could come up with any formula that you want here—

there's nothing magical about it. In case the ball's position is less than or equal to zero, we show the world normally, as we did in the previous examples.

Texture Atlas: Because Sharing Is Caring

Up until this point we have only ever used a single texture in our programs. What if we not only want to render Bob, but other superheroes or enemies or explosions or coins as well? We could have multiple textures, each holding the image of one object type. But OpenGL ES wouldn't like that much, since we'd need to switch textures for every object type we render (e.g., bind Bob's texture, render Bobs, bind the coin texture, render coins, etc.). We can do better by putting multiple images into a single texture. And that's a texture atlas: a single texture containing multiple images. We only need to bind that texture once, and we can then render any entity types for which there is an image in the atlas. That saves some state change overhead and increases our performance. Figure 8–23 shows such a texture atlas.

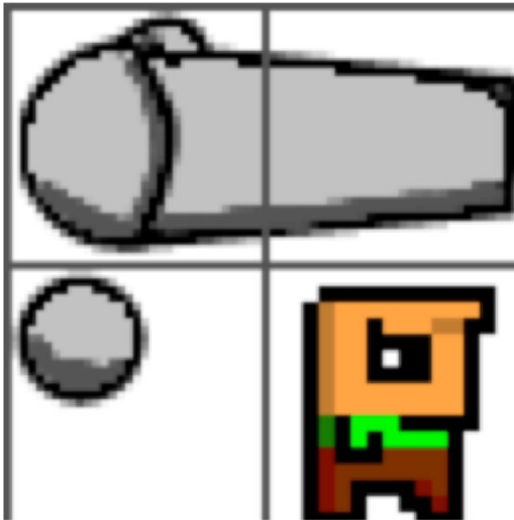

Figure 8–23. *A texture atlas*

There are three objects in Figure 8–23: a cannon, a cannonball, and Bob. The grid is not part of the texture; it's only there to illustrate how I usually create my texture atlases.

The texture atlas is 64×64 pixels in size, and each grid is 32×32 pixels. The cannon takes up two cells, the cannonball a little less than one-quarter of a cell, and Bob a single cell. Now, if you look back at how we defined the bounds (and graphical rectangles) of the cannon, cannonball, and targets, you will notice that the relation of their sizes to each other is very similar to what we have in the grid here. The target is 0.5×0.5 meters in our world, and the cannon is 0.2×0.2 meters. In our texture atlas, Bob takes up 32×32 pixels and the cannonball a little under 16×16 pixels. The relationship between the texture atlas and the object sizes in our world should be clear: 32 pixels in the atlas equals 0.5 meters in our world. Now, the cannon was 1×1 meters in our original

example, but we can of course change that. According to our texture atlas, in which the cannon takes up 64×32 pixels, we should let our cannon have a size of 1×0.5 meters in our world. Wow, that is exceptionally easy isn't it?

So why did I choose 32 pixels to match 1 meter in our world? Remember that textures must have power-of-two widths and heights. Using a power-of-two pixel unit like 32 to map to 0.5 meters in our world is a convenient way for the artist to cope with the restriction on texture sizes. It also makes it easier to get the size relations of different objects in our world right in the pixel art as well.

Note that there's nothing keeping you from using more pixels per world unit. You could choose 64 pixels or 50 pixels to match 0.5 meters in our world just fine. So what's a good pixel-to-meters size, then? That again depends on the screen resolution our game will run at. Let's do some calculations.

Our cannon world is bounded by (0,0) in the bottom-left corner and (9.6,4.8) in the top-left corner. This is mapped to our screen. Let's figure out how many pixels per world unit we have on the screen of a Hero (480×320 pixels in landscape mode):

```
pixelsPerUnitX = screenWidth / worldWidth = 480 / 9.6 =  50 pixels / meter
pixelsPerUnitY = screenHeight / worldHeight = 320 / 6.4 = 50 pixels / meter
```

Our cannon, which will now take up 1×0.5 meters in the world, will thus use 50x25 pixels on the screen. We'd use a 64×32-pixel region from our texture, so we'd actually downscale the texture image a little when rendering the cannon. That's totally fine—OpenGL ES will do this automatically for us. Depending on the minification filter we set for the texture, the result will either be crisp and pixelated (`GL_NEAREST`) or a little smoothed out (`GL_LINEAR`). If we wanted a pixel-perfect rendering on the Hero, we'd need to scale our texture images a little. We could use a grid size of 25×25 pixels instead of 32×32. However, if we just resized the atlas image (or rather redraw everything by hand), we'd have a 50×50-pixel image—a no-go with OpenGL ES. We'd have to add padding to the left and bottom to obtain a 64×64 image (since OpenGL ES requires power-of-two widths and heights). I'd say we are totally fine with OpenGL ES scaling our texture image down on the Hero.

How's the situation on higher-resolution devices like the Nexus One (800×480 in landscape mode)? Let's perform the calculations for this screen configuration via the following equations:

```
pixelsPerUnitX = screenWidth / worldWidth = 800 / 9.6 =  83 pixels / meter
pixelsPerUnitY = screenHeight / worldHeight = 480 / 6.4 = 75 pixels / meter
```

We have different pixels per unit on the x- and y-axes because the aspect ratio of our view frustum (9.6 / 6.4 = 1.5) is different from the screen's aspect ratio (800 / 480 = 1.66). We already talked about this in Chapter 4 when we outlined a couple of solutions. Back then we targeted a fixed pixel size and aspect ratio; now we'll adopt that scheme and target a fixed frustum width and height for our example. In the case of the Nexus One, the cannon, the cannonball, and Bob would get scaled up a little and stretched, due to the higher resolution and different aspect ratio. We accept this fact since we want all players to see the same region of our world. Otherwise, players with higher aspect ratios could have the advantage of being able to see more of the world.

So, how do we use such a texture atlas? We just remap our rectangles. Instead of using all of the texture, we just use portions of it. To figure out the texture coordinates of the corners of the images contained in the texture atlas, we can reuse the equations from one of the last examples. Here's a quick refresher:

```
u = x / imageWidth
v = y / imageHeight
```

Here, u and v are the texture coordinates and x and y are the pixel coordinates. Bob's top-left corner in pixel coordinates is at (32,32). If we plug that into the preceding equation, we get (0.5,0.5) as texture coordinates. We can do the same for any other corners we need, and based on this set the correct texture coordinates for the vertices of our rectangles.

An Example

Let's add this texture atlas to our previous example to make it look more beautiful. Bob will be out target.

We just copy the `Camera2DTest` and modify it a little. I placed the copy in a file called `TextureAtlasTest.java` and renamed the two classes contained in it accordingly (`TextureAtlasTest` and `TextureAtlasScreen`).

The first thing we do is add a new member to the `TextureAtlasScreen`:

```
Texture texture;
```

Instead of creating a `Texture` in the constructor, we create it in the `resume()` method. Remember that textures will get lost when our application comes back from a paused state, so we have to re-create them in the `resume()` method:

```
@Override
public void resume() {
    texture = new Texture(((GLGame)game), "atlas.png");
}
```

I just put the image in Figure 8–23 in the `assets/` folder of our project and named it `atlas.png`. (It of course doesn't contain the gridlines shown in the figure.)

Next we need to change the definitions of the vertices. We have one `Vertices` instance for each entity type (cannon, cannonball, and Bob) holding a single rectangle of four vertices and six indices, making up three triangles. All we need to do is add texture coordinates to each of the vertices in accordance with the texture atlas. We also change the cannon from being represented as a triangle to being represented by a rectangle of size 1×0.5 meters. Here's what we replace the old vertex creation code in the constructor with:

```
cannonVertices = new Vertices(glGraphics, 4, 6, false, true);
cannonVertices.setVertices(new float[] { -0.5f, -0.25f, 0.0f, 0.5f,
                                          0.5f, -0.25f, 1.0f, 0.5f,
                                          0.5f,  0.25f, 1.0f, 0.0f,
                                         -0.5f,  0.25f, 0.0f, 0.0f },
                                         0, 16);
```

```
cannonVertices.setIndices(new short[] {0, 1, 2, 2, 3, 0}, 0, 6);

ballVertices = new Vertices(glGraphics, 4, 6, false, true);
ballVertices.setVertices(new float[] { -0.1f, -0.1f, 0.0f, 0.75f,
                                        0.1f, -0.1f, 0.25f, 0.75f,
                                        0.1f,  0.1f, 0.25f, 0.5f,
                                       -0.1f,  0.1f, 0.0f, 0.5f },
                                        0, 16);
ballVertices.setIndices(new short[] {0, 1, 2, 2, 3, 0}, 0, 6);

targetVertices = new Vertices(glGraphics, 4, 6, false, true);
targetVertices.setVertices(new float[] { -0.25f, -0.25f, 0.5f, 1.0f,
                                          0.25f, -0.25f, 1.0f, 1.0f,
                                          0.25f,  0.25f, 1.0f, 0.5f,
                                         -0.25f,  0.25f, 0.5f, 0.5f },
                                         0, 16);
targetVertices.setIndices(new short[] {0, 1, 2, 2, 3, 0}, 0, 6);
```

Each of our meshes is now composed of four vertices, each having a 2D position and texture coordinates. We added six indices to the mesh, specifying the two triangles we want to render. We also made the cannon a little smaller on the y-axis. It now has size of 1×0.5 meters instead of 1×1 meters. This is also reflected in the construction of the Cannon object earlier in the constructor:

```
cannon = new Cannon(0, 0, 1, 0.5f);
```

Since we don't do any collision detection with the cannon itself, it doesn't really matter what size we set in that constructor, though. We just do it for consistency.

The last thing we need to change is our render method. Here it is in its full glory:

```
@Override
public void present(float deltaTime) {
    GL10 gl = glGraphics.getGL();
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
    camera.setViewportAndMatrices();

    gl.glEnable(GL10.GL_BLEND);
    gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
    gl.glEnable(GL10.GL_TEXTURE_2D);
    texture.bind();

    targetVertices.bind();
    int len = targets.size();
    for(int i = 0; i < len; i++) {
        GameObject target = targets.get(i);
        gl.glLoadIdentity();
        gl.glTranslatef(target.position.x, target.position.y, 0);
        targetVertices.draw(GL10.GL_TRIANGLES, 0, 6);
    }
    targetVertices.unbind();

    gl.glLoadIdentity();
    gl.glTranslatef(ball.position.x, ball.position.y, 0);
    ballVertices.bind();
    ballVertices.draw(GL10.GL_TRIANGLES, 0, 6);
    ballVertices.unbind();
```

```
    gl.glLoadIdentity();
    gl.glTranslatef(cannon.position.x, cannon.position.y, 0);
    gl.glRotatef(cannon.angle, 0, 0, 1);
    cannonVertices.bind();
    cannonVertices.draw(GL10.GL_TRIANGLES, 0, 6);
    cannonVertices.unbind();
}
```

Here, we enable blending and set a proper blending function, and enable texturing and bind our atlas texture. We also slightly adapt the `cannonVertices.draw()` call, which now renders two triangles instead of one. That's all there is to it. Figure 8–24 of our face-lifting operation.

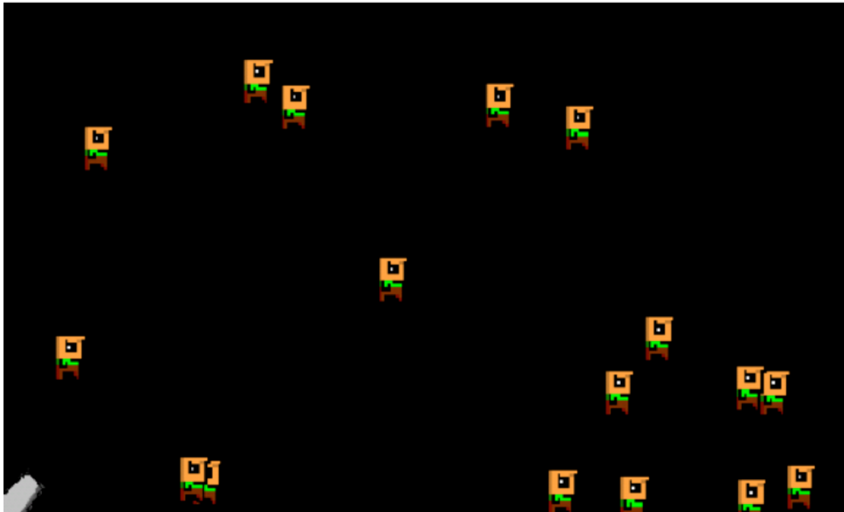

Figure 8–24. *Beautifying the cannon example with a texture atlas*

There are a few more things we need to know about texture atlases:

- When we use `GL_LINEAR` as the minification and/or magnification filter, there might be artifacts when two images within the atlas are touching each other. This is due to the texture mapper actually fetching the four nearest texels from a texture for a pixel on the screen. When it does that for the border of an image, it will also fetch texels from the neighboring image in the atlas. We can eliminate this problem by introducing an empty border of 2 pixels between our images. Even better, we can duplicate the border pixel of each image. The first solution is of course easier—just make sure your texture stays a power of two.

- There's no need to lay out all the images in the atlas in a fixed grid. We could put arbitrarily sized images in the atlas as tightly as possible. All we need to know is where one image starts and ends in the atlas so we can calculate proper texture coordinates for it. Packing arbitrarily sized images is a nontrivial problem, however. There are a couple of tools on the Web that can help you with creating a texture atlas; just do a search and you'll be hit by a plethora of options.
- Often we cannot group all images of our game into a single texture. Remember that there's a maximum texture size that varies from device to device. We can safely assume that all devices support a texture size of 512×512 pixels (or even 1024×1024). So, we just have multiple texture atlases. You should try to group objects that will be seen on the screen together in one atlas, though—say, all the objects of level 1 in one atlas, all the objects of level 2 in another, all the UI elements in another, and so on. Think about the logical grouping before finalizing your art assets.
- Remember how we drew numbers dynamically in Mr. Nom? We used a texture atlas for that. In fact, we can perform all dynamic text rendering via a texture atlas. Just put all the characters you need for your game into an atlas and render them on demand via multiple rectangles mapping to the appropriate characters in the atlas. There are tools you can find on the Web that will generate such a so-called *bitmap font* for you. For our purposes in the coming chapters, we will stick to the approach we used in Mr. Nom, though: static text gets prerendered as a whole, and only dynamic text (e.g., numbers in high scores) will get rendered via an atlas.

You might have noticed that Bobs disappear a little before they are actually hit by the cannonball graphically. That's because our bounding shapes are a little too big. We have some whitespace around Bob and the cannonball in the border. What's the solution? We just make the bounding shapes a little smaller. I want you to get a feel for this, so manipulate the source until the collision feels right. You will often find such fine-tuning "opportunities" while developing a game. Fine tuning is probably one of the most crucial parts apart from good level design. Getting things to feel right can be hard, but is highly satisfactory once you achieved the level of perfection of Super Mario Brothers. Sadly, this is nothing I can teach you, as it is dependent on the look and feel of your game. Consider it the magic sauce that sets good and bad games apart.

> **NOTE:** To handle the disappearance issue just mentioned, make the bounding rectangles a little smaller than their graphical representations to allow for some overlap before a collision is triggered.

Texture Regions, Sprites, and Batches: Hiding OpenGL ES

Our code so far for the cannon example is made up of a lot of boilerplate, some of which can be reduced. One such area is the definition of the `Vertices` instances. It's tedious to always have seven lines of code just to define a single textured rectangle. Another area we could improve is the manual calculation of texture coordinates for images in a texture atlas. Finally, there's a lot of code involved when we want to render our 2D rectangles that's highly repetitive. I also hinted at a better way of rendering many objects than having one draw call per object. We can solve all these issues by introducing a few new concepts:

- *Texture regions*: We worked with texture regions in the last example. A texture region is a rectangular area within a single texture (e.g., the area that contains the cannon in our atlas). We want a nice class that can encapsulate all the nasty calculations for translating pixel coordinates to texture coordinates.
- *Sprites*: A sprite is a lot like one of our game objects. It has a position (and possibly orientation and scale), as well as a graphical extent. We render a sprite via a rectangle, just as we render Bob or the cannon. In fact, the graphical representations of Bob and the other objects can and should be considered sprites. A sprite also maps to a region in a texture. That's where texture regions come into. While it is tempting to combine sprites with game directly, we keep them separated, following the Model-View-Controller pattern. This clean seperation between graphics and mode code makes for a better design.
- *Sprite batchers*: A sprite batcher is responsible for rendering multiple sprites in one go. To do this, the sprite batcher needs to know each sprite's position, size, and texture region. The sprite batcher will be our magic ingredient to get rid of multiple draw calls and matrix operations per object.

These concepts are highly interconnected; we'll discuss them next.

The TextureRegion Class

Since we've worked with texture regions already, it should be straightforward to figure out what we need. We know how to convert from pixel coordinates to texture coordinates. We want to have a class where we can specify pixel coordinates of an image in a texture atlas that then stores the corresponding texture coordinates for the atlas region for further processing (e.g., when we want to render a sprite). Without further ado, Listing 8–16 shows our `TextureRegion` class.

Listing 8–16. *TextureRegion.java: Converting Pixel Coordinates to Texture Coordinates*

```
package com.badlogic.androidgames.framework.gl;

public class TextureRegion {
    public final float u1, v1;
    public final float u2, v2;
    public final Texture texture;

    public TextureRegion(Texture texture, float x, float y, float width, float height) {
        this.u1 = x / texture.width;
        this.v1 = y / texture.height;
        this.u2 = this.u1 + width / texture.width;
        this.v2 = this.v1 + height / texture.height;
        this.texture = texture;
    }
}
```

The `TextureRegion` stores the texture coordinates of the top-left corner (`u1`,`v1`) and bottom-right corner (`u2`,`v2`) of the region in texture coordinates. The constructor takes a `Texture` and the top-left corner, as well as the width and height of the region, in pixel coordinates. To construct a texture region for the `Cannon`, we could do this:

```
TextureRegion cannonRegion = new TextureRegion(texture, 0, 0, 64, 32);
```

Similarly we could construct a region for Bob:

```
TextureRegion bobRegion = new TextureRegion(texture, 32, 32, 32, 32);
```

And so on and so forth. We could use this in the example code that we've already created, and use the `TextureRegion.u1`, `v1`, `u2`, and `v2` members for specifying the texture coordinates of the vertices of our rectangles. But we won't do that, since we want to get rid of these tedious definitions altogether. That's what we'll use the sprite batcher for.

The SpriteBatcher Class

As already discussed, a sprite can be easily defined by its position, size, and texture region (and optionally, its rotation and scale). It is simply a graphical rectangle in our world space. To make things easier we'll stick to the conventions of the position being in the center of the sprite and the rectangle constructed around that center. Now, we could have a `Sprite` class and use it like this:

```
Sprite bobSprite = new Sprite(20, 20, 0.5f, 0.5f, bobRegion);
```

That would construct a new sprite with its center at (20,20) in the world, extending 0.25 meters to each side, and using the `bobRegion` `TextureRegion`. But we could do this instead:

```
spriteBatcher.drawSprite(bob.x, bob.y, BOB_WIDTH, BOB_HEIGHT, bobRegion);
```

Now that looks a lot better. We don't need to construct yet another object to represent the graphical side of our object. Instead we draw an instance of Bob on demand. We could also have an overloaded method:

```
spriteBatcher.drawSprite(cannon.x, cannon.y, CANNON_WIDTH, CANNON_HEIGHT, cannon.angle,
cannonRegion);
```

That would draw the cannon, rotated by its angle. So how can we implement the sprite batcher? Where are the `Vertices` instances? Let's think about how the batcher could work.

What is batching anyway? In the graphics community, batching is defined as collapsing multiple draw calls into a single draw call. This makes the GPU happy, as discussed in the previous chapter. A sprite batcher offers one way to make this happen. Here's how:

- The batcher has a buffer that is empty initially (or becomes empty after we signal it to be cleared). That buffer will hold vertices. It is a simple float array in our case.
- Each time we call the `SpriteBatcher.drawSprite()` method we add four vertices to the buffer, based on the position, size, orientation, and texture region that were specified as arguments. This also means that we have to manually rotate and translate the vertex positions without the help of OpenGL ES. Fear not, though, the code of our `Vector2` class will come in handy here. This is the key to eliminating all the draw calls.
- Once we have specified all the sprites we want to render, we tell the sprite batcher to actually submit the vertices for all the rectangles of the sprites to the GPU in one go, and then call the actual OpenGL ES drawing method to render all the rectangles. For this, we'll transfer the contents of the float array to a `Vertices` instance and use it to render the rectangles.

> **NOTE:** We can only batch sprites that use the same texture. However, it's not a huge problem since we'll use texture atlases anyway.

The usual usage pattern of a sprite batcher looks like this:

```
batcher.beginBatch(texture);
// call batcher.drawSprite() as often as needed, referencing regions in the texture
batcher.endBatch();
```

The call to `SpriteBatcher.beginBatch()` will tell the batcher two things: it should clear its buffer and use the texture we pass in. We will bind the texture within this method for convenience.

Next we render as many sprites that reference regions within this texture as we need to. This will fill the buffer, adding four vertices per sprite.

The call to `SpriteBatcher.endBatch()` signals to the sprite batcher that we are done rendering the batch of sprites and that it should now upload the vertices to the GPU for actual rendering. We are going to use indexed rendering with a `Vertices` instance, so we'll also need to specify indices, in addition to the vertices in the float array buffer. However, since we are always rendering rectangles, we can generate the indices

beforehand once in the constructor of the SpriteBatcher. For this we need to know how many sprites the batcher should be able to draw maximally per batch. By putting a hard limited on the number of sprites that can be rendered per batch, we don't need to grow any arrays of other buffers; we can just allocate these arrays and buffers once in the constructor.

The general mechanics are rather simple. The SpriteBatcher.drawSprite() method may seem like a mystery, but it's not a big problem (if we leave out rotation and scaling for a moment). All we need to do is calculate the vertex positions and texture coordinates as defined by the parameters. We have done this manually already in previous examples—for instance, when we defined the rectangles for the cannon, the cannonball, and Bob. We'll do more or less the same in the SpriteBatcher.drawSprite() method, only automatically based on the parameters of the method. So let's check out the SpriteBatcher. Listing 8–17 shows the code.

Listing 8–17. *Excerpt from SpriteBatcher.java, Without Rotation and Scaling*

```
package com.badlogic.androidgames.framework.gl;

import javax.microedition.khronos.opengles.GL10;

import android.util.FloatMath;

import com.badlogic.androidgames.framework.impl.GLGraphics;
import com.badlogic.androidgames.framework.math.Vector2;

public class SpriteBatcher {
    final float[] verticesBuffer;
    int bufferIndex;
    final Vertices vertices;
    int numSprites;
```

Let's look at the members first. The member verticesBuffer is the temporary float array we store the vertices of the sprites of the current batch in. The member bufferIndex indicates where in the float array we should start to write the next vertices. The member vertices is the Vertices instance is used to render the batch. It also stores the indices we'll define in a minute. The member numSprites holds the number drawn so far in the current batch.

```
    public SpriteBatcher(GLGraphics glGraphics, int maxSprites) {
        this.verticesBuffer = new float[maxSprites*4*4];
        this.vertices = new Vertices(glGraphics, maxSprites*4, maxSprites*6, false,
true);
        this.bufferIndex = 0;
        this.numSprites = 0;

        short[] indices = new short[maxSprites*6];
        int len = indices.length;
        short j = 0;
        for (int i = 0; i < len; i += 6, j += 4) {
                indices[i + 0] = (short)(j + 0);
                indices[i + 1] = (short)(j + 1);
                indices[i + 2] = (short)(j + 2);
```

```
            indices[i + 3] = (short)(j + 2);
            indices[i + 4] = (short)(j + 3);
            indices[i + 5] = (short)(j + 0);
        }
        vertices.setIndices(indices, 0, indices.length);
    }
```

Moving to the constructor, we see that we have two arguments: the `GLGraphics` instance we need for creating the `Vertices` instance, and the maximum number of sprites the batcher should be able to render in one batch. The first thing we do in the constructor is create the float array. We have four vertices per sprite, and each vertex takes up four floats (two for the x- and y-coordinates and another two for the texture coordinates). We can have `maxSprites` sprites maximally, so that's 4 × 4 × `maxSprites` floats that we need for the buffer. Next we create the `Vertices` instance. We need it to store `maxSprites` × 4 vertices and `maxSprites` × 6 indices at most. We also tell the `Vertices` instance that we have not only positional attributes, but also texture coordinates for each vertex. We then initialize the `bufferIndex` and `numSprites` members to zero. Then we create the indices for our `Vertices` instance. We need to do this only once, as the indices will never change. The first sprite in a batch will always have the indices 0, 1, 2, 2, 3, 0; the next sprite will have 4, 5, 6, 6, 7, 4; and so on. We can precompute those and store them in the `Vertices` instance. This way we only need to set them once, instead of once for each sprite.

```
    public void beginBatch(Texture texture) {
        texture.bind();
        numSprites = 0;
        bufferIndex = 0;
    }
```

Next up is the `beginBatch()` method. It binds the texture and resets the `numSprites` and `bufferIndex` members so the first sprite's vertices will get inserted at the front of the `verticesBuffer` float array.

```
    public void endBatch() {
        vertices.setVertices(verticesBuffer, 0, bufferIndex);
        vertices.bind();
        vertices.draw(GL10.GL_TRIANGLES, 0, numSprites * 6);
        vertices.unbind();
    }
```

The next method is `endBatch()`; we'll call it to finalize and draw the current batch. It first transfers the vertices defined for this batch from the float array to the `Vertices` instance. All that's left is binding the `Vertices` instance, drawing `numSprites` × 2 triangles, and unbinding the `Vertices` instance again. Since we use indexed rendering, we specify the number of indices to use—which is six indices per sprite times `numSprites`. That's all there is to rendering.

```
    public void drawSprite(float x, float y, float width, float height, TextureRegion
region) {
        float halfWidth = width / 2;
        float halfHeight = height / 2;
        float x1 = x - halfWidth;
        float y1 = y - halfHeight;
```

```
        float x2 = x + halfWidth;
        float y2 = y + halfHeight;

        verticesBuffer[bufferIndex++] = x1;
        verticesBuffer[bufferIndex++] = y1;
        verticesBuffer[bufferIndex++] = region.u1;
        verticesBuffer[bufferIndex++] = region.v2;

        verticesBuffer[bufferIndex++] = x2;
        verticesBuffer[bufferIndex++] = y1;
        verticesBuffer[bufferIndex++] = region.u2;
        verticesBuffer[bufferIndex++] = region.v2;

        verticesBuffer[bufferIndex++] = x2;
        verticesBuffer[bufferIndex++] = y2;
        verticesBuffer[bufferIndex++] = region.u2;
        verticesBuffer[bufferIndex++] = region.v1;

        verticesBuffer[bufferIndex++] = x1;
        verticesBuffer[bufferIndex++] = y2;
        verticesBuffer[bufferIndex++] = region.u1;
        verticesBuffer[bufferIndex++] = region.v1;

        numSprites++;
    }
```

The next method is the workhorse of the `SpriteBatcher`. It takes the x- and y-coordinates of the center of the sprite, its width and height, and the `TextureRegion` it maps to. The method's responsibility is to add four vertices to the float array starting at the current `bufferIndex`. These four vertices form a texture-mapped rectangle . We calculate the position of the bottom-left corner (`x1`,`y1`) and the top-right corner (`x2`,`y2`), and use these four variables to construct the vertices, together with the texture coordinates from the `TextureRegion`. The vertices are added in counterclockwise order, starting at the bottom-left vertex. Once they are added to the float array, we increment the `numSprites` counter and wait for either another sprite to be added or for the batch to be finalized.

And that is all there is to do. We just eliminated a lot of drawing methods by simply buffering pretransformed vertices in a float array and rendering them in one go. That will increase our 2D sprite-rendering performance considerably compared to the method we were using before. Fewer OpenGL ES state changes and fewer drawing calls make the GPU happy.

There's one more thing we need to implement: a `SpriteBatcher.drawSprite()` method that can draw a rotated sprite All we need to do is construct the four corner vertices without adding the position, rotate them around the origin, add the position of the sprite so that the vertices are placed in the world space, and then proceed as in the previous drawing method. We could use `Vector2.rotate()` for this, but that would mean some functional overhead. We therefore reproduce the code in `Vector2.rotate()` and optimize where possible. The final method of the `SpriteBatcher` looks like Listing 8–18.

Listing 8–18. *The Rest of SpriteBatcher.java: A Method to Draw Rotated Sprites*

```
    public void drawSprite(float x, float y, float width, float height, float angle,
TextureRegion region) {
        float halfWidth = width / 2;
        float halfHeight = height / 2;

        float rad = angle * Vector2.TO_RADIANS;
        float cos = FloatMath.cos(rad);
        float sin = FloatMath.sin(rad);

        float x1 = -halfWidth * cos - (-halfHeight) * sin;
        float y1 = -halfWidth * sin + (-halfHeight) * cos;
        float x2 = halfWidth * cos - (-halfHeight) * sin;
        float y2 = halfWidth * sin + (-halfHeight) * cos;
        float x3 = halfWidth * cos - halfHeight * sin;
        float y3 = halfWidth * sin + halfHeight * cos;
        float x4 = -halfWidth * cos - halfHeight * sin;
        float y4 = -halfWidth * sin + halfHeight * cos;

        x1 += x;
        y1 += y;
        x2 += x;
        y2 += y;
        x3 += x;
        y3 += y;
        x4 += x;
        y4 += y;

        verticesBuffer[bufferIndex++] = x1;
        verticesBuffer[bufferIndex++] = y1;
        verticesBuffer[bufferIndex++] = region.u1;
        verticesBuffer[bufferIndex++] = region.v2;

        verticesBuffer[bufferIndex++] = x2;
        verticesBuffer[bufferIndex++] = y2;
        verticesBuffer[bufferIndex++] = region.u2;
        verticesBuffer[bufferIndex++] = region.v2;

        verticesBuffer[bufferIndex++] = x3;
        verticesBuffer[bufferIndex++] = y3;
        verticesBuffer[bufferIndex++] = region.u2;
        verticesBuffer[bufferIndex++] = region.v1;

        verticesBuffer[bufferIndex++] = x4;
        verticesBuffer[bufferIndex++] = y4;
        verticesBuffer[bufferIndex++] = region.u1;
        verticesBuffer[bufferIndex++] = region.v1;

        numSprites++;
    }
}
```

We do the same as in the simpler drawing method, except that we construct all four corner points instead of only the two opposite ones. This is needed for the rotation. The rest is the same as before.

What about scaling? We do not explicitly need another method, since scaling a sprite only requires scaling its width and height. We can do that outside the two drawing methods, so there's no need to have another bunch of methods for scaled drawing of sprites.

And that's the big secret behind lighting-fast sprite rendering with OpenGL ES.

Using the SpriteBatcher Class

Let's incorporate the `TextureRegion` and `SpriteBatcher` classes in our cannon example. I copied the `TextureAtlas` example and renamed it `SpriteBatcherTest`. The classes contained in it are called `SpriteBatcherTest` and `SpriteBatcherScreen`.

The first thing I did was get rid of the `Vertices` members in the screen class. We don't need them anymore, since the `SpriteBatcher` will do all the dirty work for us. Instead I added the following members:

```
TextureRegion cannonRegion;
TextureRegion ballRegion;
TextureRegion bobRegion;
SpriteBatcher batcher;
```

We now have a `TextureRegion` for each of the three objects in our atlas, as well as a `SpriteBatcher`.

Next I modified the constructor of the screen. I got rid of all the `Vertices` instantiation and initialization code, and replaced it with a single line of code:

```
batcher = new SpriteBatcher(glGraphics, 100);
```

That will set out `batcher` member to a fresh `SpriteBatcher` instance that can render 100 sprites in one batch.

The `TextureRegions` get initialized in the `resume()` method, as they depend on the `Texture`:

```
@Override
public void resume() {
    texture = new Texture(((GLGame)game), "atlas.png");
    cannonRegion = new TextureRegion(texture, 0, 0, 64, 32);
    ballRegion = new TextureRegion(texture, 0, 32, 16, 16);
    bobRegion = new TextureRegion(texture, 32, 32, 32, 32);
}
```

No surprises there. The last thing we need to change is the `present()` method. You'll be surprised how clean it's looking now. Here it is:

```
@Override
public void present(float deltaTime) {
    GL10 gl = glGraphics.getGL();
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
    camera.setViewportAndMatrices();

    gl.glEnable(GL10.GL_BLEND);
    gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
```

```
    gl.glEnable(GL10.GL_TEXTURE_2D);

    batcher.beginBatch(texture);

    int len = targets.size();
    for(int i = 0; i < len; i++) {
        GameObject target = targets.get(i);
        batcher.drawSprite(target.position.x, target.position.y, 0.5f, 0.5f, bobRegion);
    }

    batcher.drawSprite(ball.position.x, ball.position.y, 0.2f, 0.2f, ballRegion);
    batcher.drawSprite(cannon.position.x, cannon.position.y, 1, 0.5f, cannon.angle,
cannonRegion);
    batcher.endBatch();
}
```

That is super sweet. The only OpenGL ES calls we issue now are for clearing the screen, enabling blending and texturing, and setting the blend function. The rest is pure SpriteBatcher and Camera2D goodness. Since all our objects share the same texture atlas, we can render them in a single batch. We call batcher.beginBatch() with the atlas texture, render all the Bob targets using the simple drawing method, render the ball (again with the simple drawing method), and finally render the cannon using the drawing method that can rotate a sprite. We end the method by calling batcher.endBatch(), which will actually transfer the geometry of our sprites to the GPU and render everything.

Measuring Performance

So how much faster is the SpriteBatcher method than the method we used in BobTest? I added an FPSCounter to the code and timed it on a Hero, a Droid, and a Nexus One, as we did in the case of BobTest. I also increased the number of targets to 100 and set the maximum number of sprites the SpriteBatcher can render to 102, since we render 100 targets, 1 ball, and 1 cannon. Here are the results:

```
Hero (1.5):
12-27 23:51:09.400: DEBUG/FPSCounter(2169): fps: 31
12-27 23:51:10.440: DEBUG/FPSCounter(2169): fps: 31
12-27 23:51:11.470: DEBUG/FPSCounter(2169): fps: 32
12-27 23:51:12.500: DEBUG/FPSCounter(2169): fps: 32

Droid (2.1.1):
12-27 23:50:23.416: DEBUG/FPSCounter(8145): fps: 56
12-27 23:50:24.448: DEBUG/FPSCounter(8145): fps: 56
12-27 23:50:25.456: DEBUG/FPSCounter(8145): fps: 56
12-27 23:50:26.456: DEBUG/FPSCounter(8145): fps: 55

Nexus One (2.2.1):
12-27 23:46:57.162: DEBUG/FPSCounter(754): fps: 61
12-27 23:46:58.171: DEBUG/FPSCounter(754): fps: 61
12-27 23:46:59.181: DEBUG/FPSCounter(754): fps: 61
12-27 23:47:00.181: DEBUG/FPSCounter(754): fps: 60
```

Before we come to any conclusions, let's test the old method as well. Since our example is not equivalent to the old BobTest, I also modified the TextureAtlasTest, which is the

same as our current example—the only difference being that it uses the old BobTest method for rendering. Here are the results:

```
Hero (1.5):
12-27 23:53:45.950: DEBUG/FPSCounter(2303): fps: 46
12-27 23:53:46.720: DEBUG/dalvikvm(2303): GC freed 21811 objects / 524280 bytes in 135ms
12-27 23:53:46.970: DEBUG/FPSCounter(2303): fps: 40
12-27 23:53:47.980: DEBUG/FPSCounter(2303): fps: 46
12-27 23:53:48.990: DEBUG/FPSCounter(2303): fps: 46

Droid (2.1.1):
12-28 00:03:13.004: DEBUG/FPSCounter(8277): fps: 52
12-28 00:03:14.004: DEBUG/FPSCounter(8277): fps: 52
12-28 00:03:15.027: DEBUG/FPSCounter(8277): fps: 53
12-28 00:03:16.027: DEBUG/FPSCounter(8277): fps: 53

Nexus One (2.2.1):
12-27 23:56:09.591: DEBUG/FPSCounter(873): fps: 61
12-27 23:56:10.591: DEBUG/FPSCounter(873): fps: 60
12-27 23:56:11.601: DEBUG/FPSCounter(873): fps: 61
12-27 23:56:12.601: DEBUG/FPSCounter(873): fps: 60
```

The Hero performs a lot worse with our new SpriteBatcher method as compared to the old way of using glTranslate() and similar methods. The Droid actually benefits from the new SpriteBatcher method, and the Nexus One doesn't really care what we use. If we'd increased the number of targets by another 100, you'd see that the SpriteBatcher method would also be faster on the Nexus One.

So what's up with the Hero? The problem in BobTest was that we called too many OpenGL ES methods, so why is it performing worse now that we're fewer OpenGL ES method calls?

Working Around a Bug in FloatBuffer

The reason for this isn't obvious at all. Our SpriteBatcher puts a float array into a direct ByteBuffer each frame when we call Vertices.setVertices(). The method boils down to calling FloatBuffer.put(float[]), and that's the culprit of our performance hit here. While desktop Java implements that FloatBuffer method via a real bulk memory move, the Harmony version calls FloatBuffer.put(float) for each element in the array. And that's extremely unfortunate, as that method is a JNI method, which has a lot of overhead (much like the OpenGL ES methods, which are also JNI methods).

There are a couple of solutions. IntBuffer.put(int[]) does not suffer from this problem, for example. We could replace the FloatBuffer in our Vertices class with an IntBuffer and modify Vertices.setVertices() so that it first transfers the floats from the float array to a temporary int array and then copies the contents of that int array to the IntBuffer. This solution was proposed by Ryan McNally, a fellow game developer, who also reported the bug on the Android bug tracker. It produces a five-times performance increase on the Hero, and a little less on other Android devices.

I modified the Vertices class to include this fix. For this I changed the vertices member to be an IntBuffer. I also added a new member called tmpBuffer, which is an int[] array. The tmpBuffer array is initialized in the constructor of Vertices as follows:

```
this.tmpBuffer = new int[maxVertices * vertexSize / 4];
```

We also get an IntBuffer view from the ByteBuffer in the constructor instead of a FloatBuffer:

```
vertices = buffer.asIntBuffer();
```

And the Vertices.setVertices() method looks like this now:

```
public void setVertices(float[] vertices, int offset, int length) {
    this.vertices.clear();
    int len = offset + length;
    for(int i=offset, j=0; i < len; i++, j++)
        tmpBuffer[j] = Float.floatToRawIntBits(vertices[i]);
    this.vertices.put(tmpBuffer, 0, length);
    this.vertices.flip();
}
```

So, all we do is first transfer the contents of the vertices parameter to the tmpBuffer. The static method Float.floatToRawIntBits() reinterprets the bit pattern of a float as an int. We then just need to copy the contents of the int array to the IntBuffer, formerly known as a FloatBuffer. Does it improve performance? Running the SpriteBatcherTest produces the following output now on the Hero, Droid, and Nexus One:

```
Hero (1.5):
12-28 00:24:54.770: DEBUG/FPSCounter(2538): fps: 61
12-28 00:24:54.770: DEBUG/FPSCounter(2538): fps: 61
12-28 00:24:55.790: DEBUG/FPSCounter(2538): fps: 62
12-28 00:24:55.790: DEBUG/FPSCounter(2538): fps: 62

Droid (2.1.1):
12-28 00:35:48.242: DEBUG/FPSCounter(1681): fps: 61
12-28 00:35:49.258: DEBUG/FPSCounter(1681): fps: 62
12-28 00:35:50.258: DEBUG/FPSCounter(1681): fps: 60
12-28 00:35:51.266: DEBUG/FPSCounter(1681): fps: 59

Nexus One (2.2.1):
12-28 00:27:39.642: DEBUG/FPSCounter(1006): fps: 61
12-28 00:27:40.652: DEBUG/FPSCounter(1006): fps: 61
12-28 00:27:41.662: DEBUG/FPSCounter(1006): fps: 61
12-28 00:27:42.662: DEBUG/FPSCounter(1006): fps: 61
```

Yes, I double-checked; this is not a typo. The Hero really achieves 60 FPS now. A workaround consisting of five lines of code increases our performance by 50 percent. The Droid also benefited from this fix a little.

The problem is fixed in the latest release of Android version 2.3. However, it will be quite some time before most phones run this version, so we should keep this workaround for the time being.

NOTE: There's another, even faster workaround. It involves a custom JNI method that does the memory move in native code. You can find it if you search for the "Android Game Development Wiki" on the Net. I use this most of the time instead of the pure Java workaround. However, including JNI methods is a bit more complex, which is why I described the pure-Java workaround here.

Sprite Animation

If you've ever played a 2D video game, you know that we are still missing one vital component: sprite animation. The animation consists of so-called *keyframes*, which produce the illusion of movement. Figure 8–25 shows a nice animated sprite by Ari Feldmann (part of his royalty-free SpriteLib).

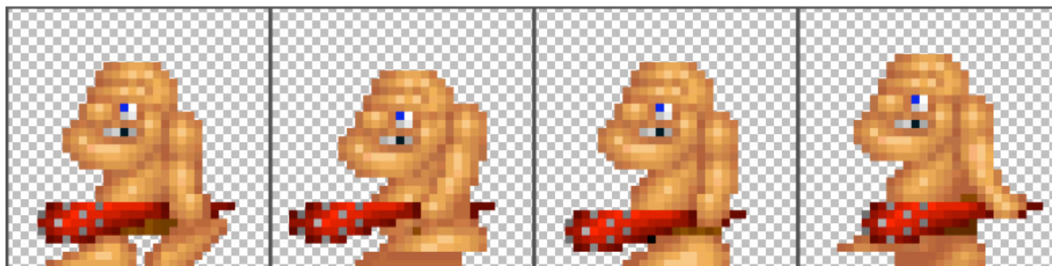

Figure 8–25. *A walking caveman, by Ari Feldmann (grid not in original)*

The image is 256×64 pixels in size, and each keyframe is 64×64 pixels. To produce animation, we just draw a sprite using the first keyframe for some amount of time—say, 0.25 seconds—then we switch to the next keyframe, and so on. When we reach the last frame we have two options: we can stay at the last keyframe or start at the beginning again (and perform what is called a *looping animation*).

We can easily do this with our `TextureRegion` and `SpriteBatcher` classes. Usually we'd not only have a single animation like in Figure 8–25, but many more in a single atlas. Besides the walk animation, we could have a jump animation, an attack animation, and so on. For each animation we need to know the frame duration, which tells us how long we keep using a single keyframe of the animation before we switch to the next frame.

The Animation Class

From this we can define the requirements for an `Animation` class, which stores the data for a single animation, such as the walk animation in Figure 8–25:

- An `Animation` holds a number of `TextureRegions`, which store where in the texture atlas each keyframe is located. The order of the `TextureRegions` is the same as that used for playing back the animation.
- The `Animation` also stores the frame duration that has to pass before we switch to the next frame.
- The `Animation` should provide us with a method to which we pass the time we've been in the state that the `Animation` represents (e.g., walking left), and that will return the appropriate `TextureRegion`. The method should take into consideration whether we want the `Animation` to loop or to stay at the last frame when the end is reached.

This last bullet point is important because it allows us to store a single `Animation` instance to be used by multiple objects in our world. An object just keeps track of its current state (e.g., whether it is walking, shooting, or jumping, and how long it has been in that state). When we render this object, we use the state to select the animation we want to play back, and the state time to get the correct `TextureRegion` from the `Animation`. Listing 8–19 shows the code of our new `Animation` class.

Listing 8–19. *Animation.java, a Simple Animation Class*

```
package com.badlogic.androidgames.framework.gl;

public class Animation {
    public static final int ANIMATION_LOOPING = 0;
    public static final int ANIMATION_NONLOOPING = 1;

    final TextureRegion[] keyFrames;
    final float frameDuration;

    public Animation(float frameDuration, TextureRegion ... keyFrames) {
        this.frameDuration = frameDuration;
        this.keyFrames = keyFrames;
    }

    public TextureRegion getKeyFrame(float stateTime, int mode) {
        int frameNumber = (int)(stateTime / frameDuration);

        if(mode == ANIMATION_NONLOOPING) {
            frameNumber = Math.min(keyFrames.length-1, frameNumber);
        } else {
            frameNumber = frameNumber % keyFrames.length;
        }
        return keyFrames[frameNumber];
    }
}
```

We first define two constants to be used with the getKeyFrame() method. The first one says the animation should be looping, and the other one says that is should stop at the last frame.

Next we define two members: an array holding the TextureRegions and a float storing the frame duration.

We pass the frame duration and the TextureRegions that hold the keyframes to the constructor, which simply stores them. We could make a defensive copy of the keyFrames array, but that would allocate a new object, which would make the garbage collector a little mad.

The interesting piece is the getKeyFrame() method. We pass in the time that the object has been in the state that the animation represents, as well as the mode, either Animation.ANIMATION_LOOPING or Animation.NON_LOOPING. We first calculate how many frames have already been played for the given state based on the stateTime. In case the animation shouldn't be looping, we simply clamp the frameNumber to the last element in the TextureRegion array. Otherwise, we take the modulus, which will automatically create the looping effect we desire (e.g., 4 % 3 = 1). All that's left is returning the proper TextureRegion.

An Example

Let's create an example called AnimationTest with a corresponding screen called AnimationScreen. As always we'll only discuss the screen itself.

We want to render a number of cavemen, all walking to the left. Our world will be the same size as our view frustum, which has the size 4.8×3.2 meters (this is really arbitrary; we could use any size). A caveman is a DynamicGameObject with a size of 1×1 meters. We will derive from DynamicGameObject and create a new class called Caveman, which will store an additional member that keeps track of how long the caveman has been walking already. Each caveman will move 0.5 m/s either to the left or to the right. We'll also add an update() method to the Caveman class to update the caveman's position based on the delta time and his velocity. If a caveman reaches the left or right edge of our world, we set him to the other side of the world. We'll use the image in Figure 8–25 and create TextureRegions and an Animation instance accordingly. For rendering we'll use a Camera2D instance and a SpriteBatcher because they are fancy. Listing 8–20 shows the code of the Caveman class.

Listing 8–20. *Excerpt from AnimationTest, Showing the Inner Caveman Class.*

```
static final float WORLD_WIDTH = 4.8f;
static final float WORLD_HEIGHT = 3.2f;

static class Caveman extends DynamicGameObject {
    public float walkingTime = 0;

    public Caveman(float x, float y, float width, float height) {
        super(x, y, width, height);
        this.position.set((float)Math.random() * WORLD_WIDTH,
```

```
                        (float)Math.random() * WORLD_HEIGHT);
            this.velocity.set(Math.random() > 0.5f?-0.5f:0.5f, 0);
            this.walkingTime = (float)Math.random() * 10;
        }

        public void update(float deltaTime) {
            position.add(velocity.x * deltaTime, velocity.y * deltaTime);
            if(position.x < 0) position.x = WORLD_WIDTH;
            if(position.x > WORLD_WIDTH) position.x = 0;
            walkingTime += deltaTime;
        }
    }
```

The two constants WORLD_WIDTH and WORLD_HEIGHT are part of the enclosing AnimationTest class and are used by the inner classes. Our world is 4.8×3.2 meters in size.

Next up is the inner Caveman class, which extends DynamicGameObject, since we will move cavemen based on velocity. We define an additional member that keeps track of how long the caveman is walking already. In the constructor we place the caveman at a random position and let him either walk left or right. We also initialize the walkingTime member to a number between 0 and 10; this way our cavemen won't walk in sync.

The update() method advances the caveman based on his velocity and the delta time. In case he leaves the world, we reset him to either the left or right edge. We also add the delta time to the walkingTime to keep track of how long he's been walking.

Listing 8–21 shows the AnimationScreen class.

Listing 8–21. *Excerpt from AnimationTest.java: The AnimationScreen Class*

```
class AnimationScreen extends Screen {
    static final int NUM_CAVEMEN = 10;
    GLGraphics glGraphics;
    Caveman[] cavemen;
    SpriteBatcher batcher;
    Camera2D camera;
    Texture texture;
    Animation walkAnim;
```

Our screen class has the usual suspects as members. We have a GLGraphics instance, a Caveman array, a SpriteBatcher, a Camera2D, the Texture containing the walking keyframes, and an Animation instance.

```
    public AnimationScreen(Game game) {
        super(game);
        glGraphics = ((GLGame)game).getGLGraphics();
        cavemen = new Caveman[NUM_CAVEMEN];
        for(int i = 0; i < NUM_CAVEMEN; i++) {
            cavemen[i] = new Caveman((float)Math.random(), (float)Math.random(), 1, 1);
        }
        batcher = new SpriteBatcher(glGraphics, NUM_CAVEMEN);
        camera = new Camera2D(glGraphics, WORLD_WIDTH, WORLD_HEIGHT);
    }
```

In the constructor we create the Caveman instances, as well as the SpriteBatcher and Camera2D.

```
    @Override
    public void resume() {
        texture = new Texture(((GLGame)game), "walkanim.png");
        walkAnim = new Animation( 0.2f,
                                  new TextureRegion(texture, 0, 0, 64, 64),
                                  new TextureRegion(texture, 64, 0, 64, 64),
                                  new TextureRegion(texture, 128, 0, 64, 64),
                                  new TextureRegion(texture, 192, 0, 64, 64));
    }
```

In the resume() method we load the texture atlas containing the animation keyframes from the asset file walkanim.png, which is the same as in Figure 8–25. Afterward, we create the Animation instance, setting the frame duration to 0.2 seconds and passing in a TextureRegion for each of the keyframes in the texture atlas.

```
    @Override
    public void update(float deltaTime) {
        int len = cavemen.length;
        for(int i = 0; i < len; i++) {
            cavemen[i].update(deltaTime);
        }
    }
```

The update() method just loops over all Caveman instances and calls their Caveman.update() method with the current delta time. This will make the cavemen move and update their walking times.

```
    @Override
    public void present(float deltaTime) {
        GL10 gl = glGraphics.getGL();
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        camera.setViewportAndMatrices();

        gl.glEnable(GL10.GL_BLEND);
        gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
        gl.glEnable(GL10.GL_TEXTURE_2D);

        batcher.beginBatch(texture);
        int len = cavemen.length;
        for(int i = 0; i < len; i++) {
            Caveman caveman = cavemen[i];
            TextureRegion keyFrame = walkAnim.getKeyFrame(caveman.walkingTime,
Animation.ANIMATION_LOOPING);
            batcher.drawSprite(caveman.position.x, caveman.position.y,
caveman.velocity.x < 0?1:-1, 1, keyFrame);
        }
        batcher.endBatch();
    }

    @Override
    public void pause() {
    }
```

```
    @Override
    public void dispose() {
    }
}
```

Finally we have the present() method. We start off by clearing the screen and setting the viewport and projection matrix via our camera. Next we enable blending and texture mapping, and set the blend function. We start rendering by telling the sprite batcher that we want to start a new batch using the animation texture atlas. Next we loop through all the cavemen and render them. For each caveman we first fetch the correct keyframe from the Animation instance based on the caveman's walking time. We specify that the animation should be looping. Then we draw the caveman with the correct texture region at his position.

But what do we do with the width parameter here? Remember that our animation texture only contains keyframes for the "walk left" animation. We want to flip the texture horizontally in case the caveman is walking to the right, which we can do by simply specifying a negative width. If you don't trust me, go back to the SpriteBatcher code and check whether this works. We essentially flip the rectangle of the sprite by specifying a negative width. We could do the same vertically as well by specifying a negative height.

Figure 8–26 shows our walking cavemen.

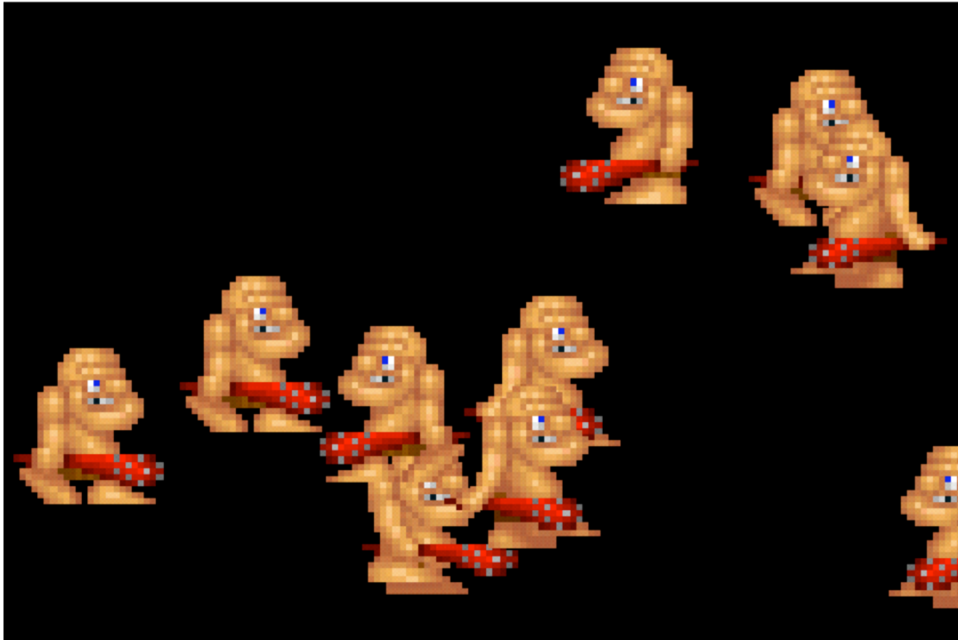

Figure 8–26. *Cavemen walking*

And that is all there is to know to produce a nice 2D game with OpenGL ES. Note how we still separate the game logic and the presentation from each other. A caveman does not need to know that he is actually being rendered. He therefore doesn't keep any

rendering-related members, such as an `Animation` instance or a `Texture`. All we need to do is keep track of the state of the caveman and how long he's been in that state. Together with his position and size, we can then render him easily by using our little helper classes.

Summary

You should now be well equipped to create almost any 2D game you want. We discussed vectors and how to work with them, resulting in a nice, reusable `Vector2` class. We also looked into basic physics for creating things like ballistic cannonballs. Collision detection is also a vital part of most games, and you should now know how to do it correctly and efficiently via a `SpatialHashGrid`. We explored a way to keep our game logic and objects separated from the rendering by creating `GameObject` and `DynamicGameObject` classes that keep track of the state and shape of objects. We covered how easy it is to implement the concept of a 2D camera via OpenGL ES, all based on a single method called `glOrthof()`. We discussed texture atlases, why we need them, and how we can use them. We expanded on the concept by introducing texture regions, sprites, and how we can render them efficiently via a `SpriteBatcher`. Finally we looked into sprite animation, which turns out to be extremely simple to implement.

In the next chapter, we'll create a new game with all the new tools we have. You'll be surprised how easy that will be.