Mobile Application Lab 14

Instructors: Legand Burge, Asad Ullah Naweed
Date: 10/04/2018
Topics: Services and Notifications

Be sure to click on the links in this doc to view more information about certain topics. These may or may not help you in your lab, but can always provide some extra context and information.

Initial Setup

You'll be creating a new project for this lab. This new project should have the package name `com.techexchange.mobileapps.lab14`. In this lab, we'll be exploring three different concepts: Services, Handlers and Notifications.

Custom Views

Right-click on the main package in your project, and create a new class called `BouncingBallView`. Make this a subclass of the `View` class, as follows:

```
public class BouncingBallView extends View {
  private static final String TAG = "BouncingBallView";

  // Having this constructor is necessary.
  public BouncingBallView(Context context) {
    super(context);
  }

  @Override
  public void onDraw(Canvas canvas) {
    super.onDraw(canvas);

    // This is where we'll be drawing things...

    try {
      Thread.sleep(30);
    } catch (InterruptedException ex) {
      Log.e(TAG, "Sleep interrupted!", ex);
    }
```

```
    invalidate(); // Force a redraw.
  }
}
```

Now, inside your `MainActivity` class, modify the `onCreate` method to be as follows:

```
public class MainActivity extends AppCompatActivity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(new BouncingBallView(this));
  }
}
```

Note that we are not using the layout XML file to specify our layout. We're going to use our custom view as the main content view for our activity. This is a bit more complex, but it allows us to use the view as a canvas to draw whatever we want. We can also set the content view of a fragment to be this view, in case we want to have multiple fragments, each with its own custom view.

The remainder of this lab will show you how you can use a custom view to implement complex physics simulations. We're going to be looking at different particle systems, and seeing how they behave under different situations.

Bouncing Balls Part 1

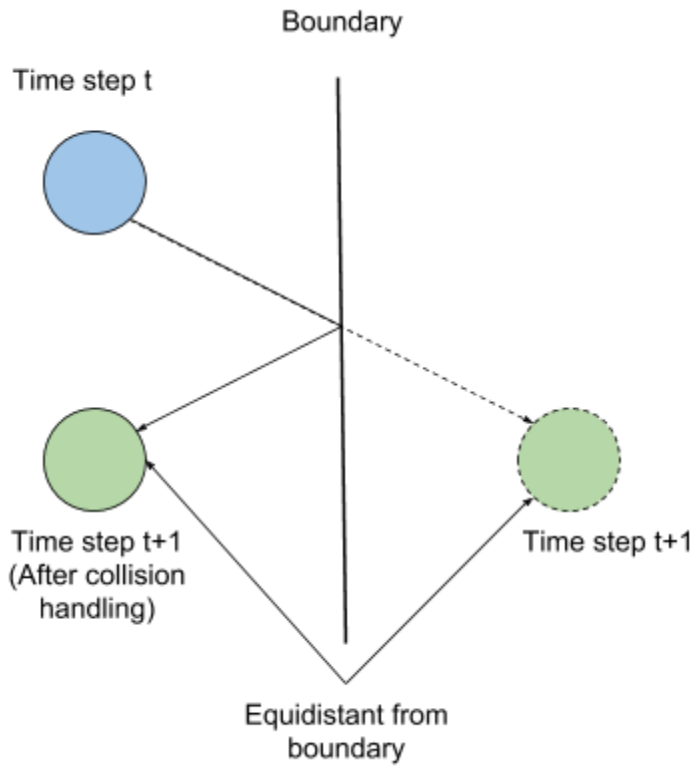
For this first exercise, you're going to implement a simulation of a single particle bouncing around the screen with a constant speed. Assume that the particle is moving in 2D space, and can be modelled as a perfect circle. It should should collide with the walls of the screen, and then it should bounce back, while maintaining its constant speed.

Because your activity is sleeping for 30ms between consecutive frame redraws, your time step interval should be 0.003 seconds. Specify this as a constant floating point number, and use it when writing your simulation code.

```
public class BouncingBallView extends View {
  private static final String TAG = "BouncingBallView";

  // Having this constructor is necessary.
```

```
  public BouncingBallView(Context context) {
    super(context);
  }

  @Override
  public void onDraw(Canvas canvas) {
    super.onDraw(canvas);

    // This is where we'll be drawing things...

    try {
      Thread.sleep(30);
    } catch (InterruptedException ex) {
      Log.e(TAG, "Sleep interrupted!", ex);
    }

    invalidate(); // Force a redraw.
  }
}
```

At each time step after updating the position of the particle, you should be checking for collisions with the boundaries, and then handling them appropriately. You should correct both the position as well as the velocity of the particle when this happens.



The way to do this is to set up an update function, call it on every draw, and then force a redraw after a delay, like so:

```
public class BouncingBallView extends View {
  private static final String TAG = "BouncingBallView";

  // Having this constructor is necessary.
  public BouncingBallView(Context context) {
    super(context);
  }

  @Override
  public void onDraw(Canvas canvas) {
    super.onDraw(canvas);

    // This is where we'll be drawing things...

    update();

    try {
      Thread.sleep(30);
    } catch (InterruptedException ex) {
      Log.e(TAG, "Sleep interrupted!", ex);
    }

    invalidate(); // Force a redraw.
  }

  private void update() {
    // Perform physics simulations here...
  }
}
```

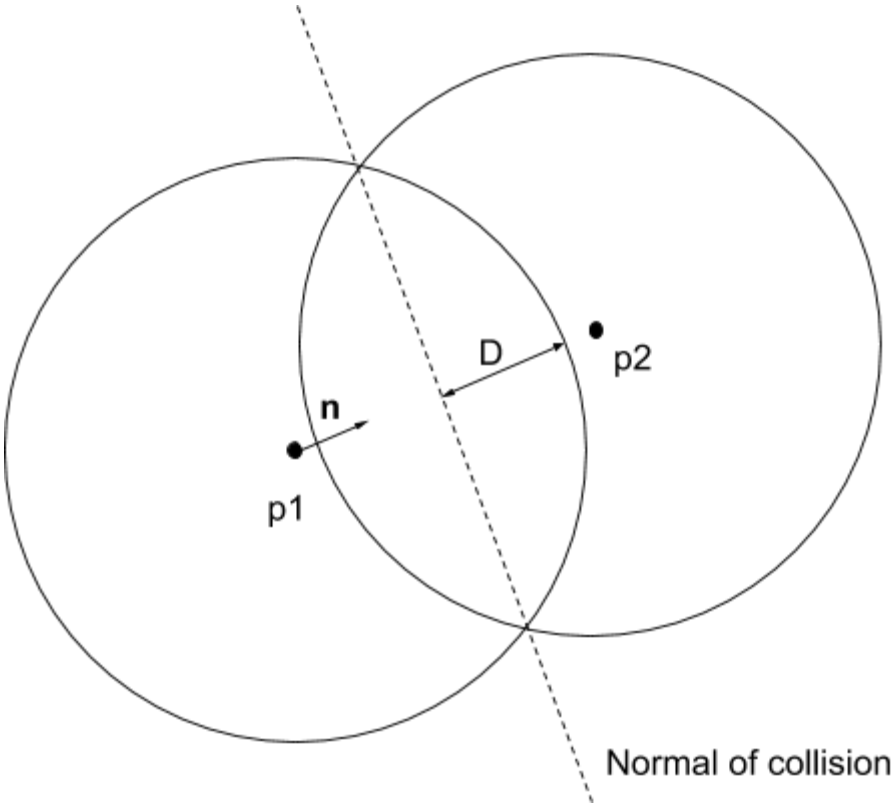
To draw things on your custom view, you'll need to invoke methods on the `Canvas` argument passed to the `onDraw` method. Check out the `Canvas` documentation to see what methods are available. Pay close attention to the `drawOval` method.

Bouncing Balls Part 2

Create a new class called `MultiBouncingBallView`, similar to how you created your `BouncingBallView` class. Copy-paste the simulation code from Part 1 in this new class, but extend it to simulate multiple particles instead of just one.

First, set up the code to create multiple particles, each with a fixed random color. Create a private static int variable called `NUM_BALLS` and set it to be 20, and then write code to simulate `NUM_BALLS` number of particles. Don't worry about the particles colliding with each other for the moment.

Once you have multiple particles moving around and colliding with the walls, it's time to simulate inter-particle collision. Doing this right involves a bit of vector math, which is outlined below:



In this case, the position and velocities of the two particles need to be flipped along the normal of collision, instead of a fixed axis-aligned boundary. This makes the math slightly more complicated, but not unsolvable. Try and figure it out by yourself on pen and paper first before coding it up. If you have trouble, please ask the instructor.

Bouncing Balls Part 3

For this last part, we're going to go ahead and add gravity to the particle simulation. Make a new custom view class called `GravityBouncingBallView` and copy-paste the code from Part 1. We're going to add gravity to the single-particle simulation first.

In your `update()` function, you were previously only updating the position of the ball. You will now also need to update the velocity of the ball. For this simulation, assume that the motion of the ball is linear over each small time step, even though this is obviously an over-simplification.

Once you have the particle bouncing under gravity, and colliding with the walls, create another custom view and implement a multi-particle simulation with the particles colliding with each other and acting under gravity.