08309 Lab 5 Animation, Touches and Gestures

Goal

There are lots of ways to use animations to create responsive user interfaces on the Android platform. In this lab you work with two projects. In the first project you will create the start of a simple game where you drag a pirate ship around the sea, avoiding passing rocks. You will learn how to create a simple animation by extending a view, updating it on a background thread and overriding the view's onDraw method. You will also learn how to create a smoother animation by extending the SurfaceView class. Then you will learn how to intercept a touch event to move an object. In the second project you will make a game called plunder. In plunder you either see a picture that is treasure, or not treasure. Fling right for treasure and left for not treasure. You will create a GestureDetector to detect a fling gesture.

Check out the folder **<your repository>**\Week5\Lab_5_1_PirateRocks to C:\Temp and open this application in Android Studio. When you've finished remember to commit your changes and delete your work from the local drive. Run the application so you can see what is going on.

Extending the ImageView class to make a countdown

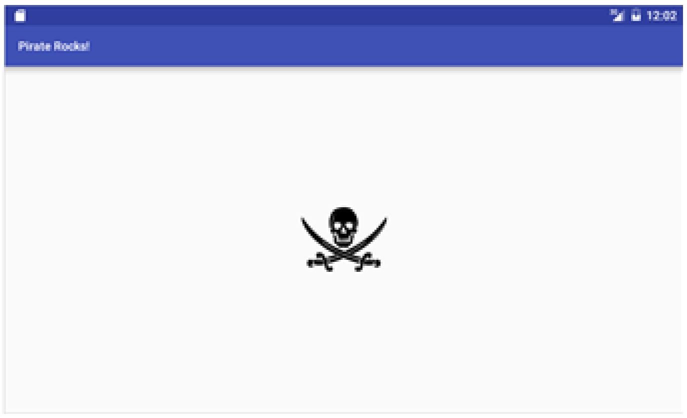
For the first part of the lab we want to create a timer that display a different image each section and then, once the time is finished, launches a new activity. We could use something like an imageView, but we need to change the way the onDraw method works.

Inside the MainActivity class create a class called CountdownView that extends ImageView. It should have a private Bitmap member variable called m_Bitmap, and the constructor should take a Context object called pContext and a Bitmap object called pBitmap. In the constructor you should call the parent constructor passing the Context object. You should also assign pBitmap to m_Bitmap, and call the setImageBitmap method to set the bitmap that the ImageView uses.

Once you've done all that, in the onCreate method of the MainActivity class create a final CountdownView (final prevents anything else from being assigned to this variable). Add the new custom view instance to the layout and set some parameters to centre it.

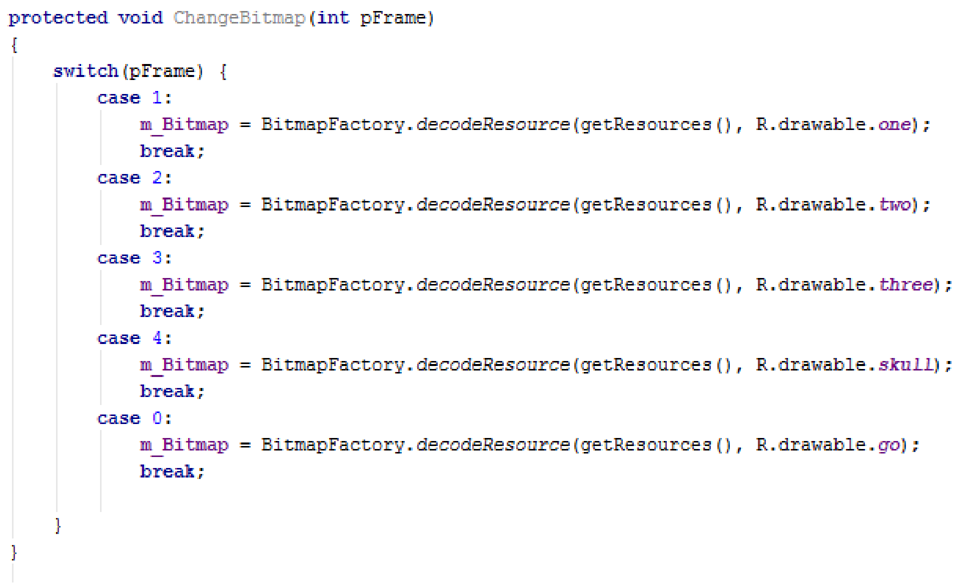
```
final CountdownView countdownView = new CountdownView(getApplicationContext(), bitmap);
layout.addView(countdownView);
RelativeLayout.LayoutParams layoutParams =
        (RelativeLayout.LayoutParams)countdownView.getLayoutParams();
layoutParams.addRule(RelativeLayout.CENTER_IN_PARENT, RelativeLayout.TRUE);
```

You should end up with something like this:



Once you've done that commit your code to SVN with an appropriate message like *"Completed lab 5.1 task 1 created a custom view by extending imageView"*.

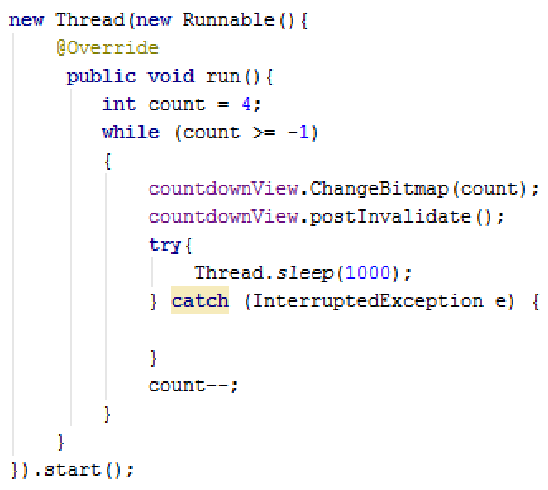
So far so good, but we're meant to be animating this imageView and replacing the image every second. Start by adding a method to CountdownView that changes the bitmap that we store in the m_Bitmap member variable according an integer parameter that represents the current frame of the animation.



```
protected void ChangeBitmap(int pFrame)
{
    switch(pFrame) {
        case 1:
            m_Bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.one);
            break;
        case 2:
            m_Bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.two);
            break;
        case 3:
            m_Bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.three);
            break;
        case 4:
            m_Bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.skull);
            break;
        case 0:
            m_Bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.go);
            break;

    }
}
```

Next in the onCreate method of the main activity we need to start a thread that will call this method. Take a minute to check over the code in the run method. It sets a count value to 4, and then calls our new ChangeBitmap method and calls postInvalidate on our CountdownView. Then the Thread sleeps for a second before decrementing the count variable and repeating the process.



```
new Thread(new Runnable(){
    @Override
     public void run(){
        int count = 4;
        while (count >= -1)
        {
            countdownView.ChangeBitmap(count);
            countdownView.postInvalidate();
            try{
                Thread.sleep(1000);
            } catch (InterruptedException e) {

            }
            count--;
        }
    }
}).start();
```

When postInvalidate is called on our CountdownView that notifies the system that this view may need redrawing. Remember, we can't change views that were created in the UI thread in this thread, so instead of calling redraw directly we are asking the system to redraw. In the CountdownView overload the onDraw method. In that method you should set the bitmap image to be the member variable we have already stored, and we should call the onDraw method in the parent class.

```
@Override
protected void onDraw(Canvas pCanvas)
{
    setImageBitmap(m_Bitmap);
    super.onDraw(pCanvas);
}
```

Run the code. You should see the bitmap change to countdown from three to one, and then display a Go! Message. Commit your code to SVN with an appropriate message like *"Completed lab 5.1 task 2 animated the bitmaps in a custom view using a separate thread"*.

For the next part of the application we want to launch the SeaActivity. You should know how to do this already. In the ChangeBitmap method add the code to launch the SeaActivity in a new case for the value -1.

Once you've done that run your code and ensure that the SeaActivity starts after the animation has finished. Then commit your code with an appropriate message like *"Completed lab 5.1 task 3 launched SeaActivity after the animation is finished"*.

Take some time to look at the code that is already in SeaActivity. You should notice that there is a class called SeaView that extends SurfaceView and implements the SurfaceHolder.Callback interface. That means that there are overridden methods called surfaceChanged, surfaceCreated and surfaceDestroyed. At the moment they don't do anything, but we'll fix that soon. This part of the program will feature three rocks that move from the left hand side of the screen to the right hand side. A Bitmap that is used to draw the rocks is already included, as is some float values used to position the rocks and some functions that help set the rocks positions to be random values.

We still need to do a bit of setting up though. We need to create a member variable of type Paint called m_Paint to help us draw things, and a SurfaceHolder called m_SurfaceHolder. Create those member variables and then set them up at the end of the SeaView constructor.

```
m_Paint = new Paint();
m_SurfaceHolder = getHolder();
m_SurfaceHolder.addCallback(this);
```

Note the line that adds this class as the SurfaceHolder.Callback. This will register the appropriate methods concerning our SurfaceView to be called. To make sure this is working add the following code to the surfaceCreated method.

```
Canvas canvas = null;
canvas = m_SurfaceHolder.lockCanvas();
canvas.drawColor(Color.parseColor("#75a2d9"));
m_SurfaceHolder.unlockCanvasAndPost(canvas);
```

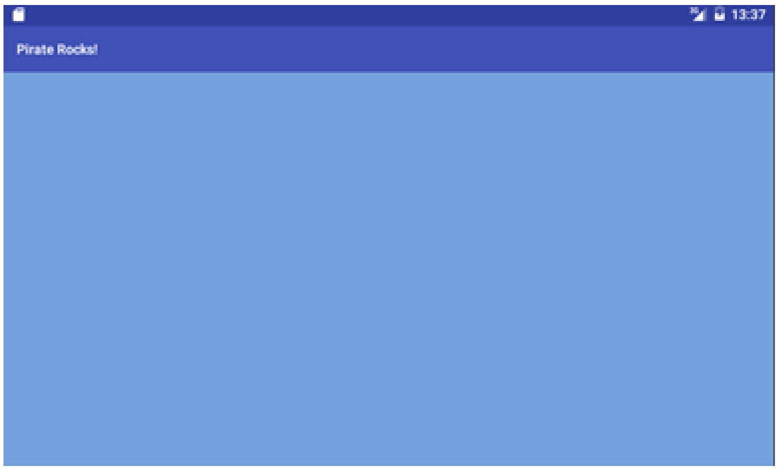
In contrast to previous examples this code gets a lock on the canvas, enabling another thread to draw to it. In this case the colour is cleared to a nice sea blue colour and the lock is returned.

So, we've extended SurfaceView, implements SurfaceHolder.Callback and registered the appropriate methods to be called at the appropriate times. We still need to create an instance of our new

SeaView class in the SeaActivity, and add that to the existing layout. Create a member variable of type SeaView and call it m_SeaView, then use the following code to create a new instance of SeaView, get hold of the layout and add the new view to it.

```
RelativeLayout relativeLayout = (RelativeLayout) findViewById(R.id.seaLayout);
m_SeaView = new SeaView(this);
relativeLayout.addView(m_SeaView);
```

Run your code. After the countdown is complete you should see a blue screen like this.



Commit your code with an appropriate message like *"Completed lab 5.1 task 4 Created a SeaView class, registered its callbacks and turned the canvas blue!"*

Next we want to animate some rocks moving from left to right. Add a method to SeaView called MoveRocks:

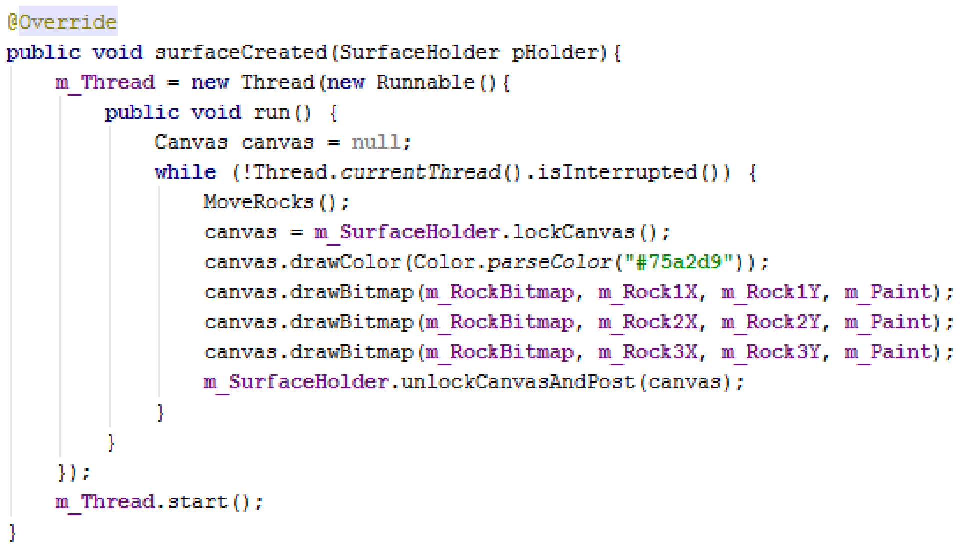
```
private void MoveRocks() {
    DisplayMetrics displayMetrics = new DisplayMetrics();
    SeaActivity.this.getWindowManager().getDefaultDisplay().getMetrics(displayMetrics);
    float screenWidth = displayMetrics.widthPixels;
    m_Rock1X += 10;
    if(m_Rock1X > screenWidth){
        m_Rock1X = getRandomRockX();
        m_Rock1Y = getRandomRockY();
    }

    m_Rock2X += 10;
    if(m_Rock2X > screenWidth){
        m_Rock2X = getRandomRockX();
        m_Rock2Y = getRandomRockY();
    }

    m_Rock3X += 10;
    if(m_Rock3X > screenWidth){
        m_Rock3X = getRandomRockX();
        m_Rock3Y = getRandomRockY();
    }
}
```

Take a minute to look at this code. It's certainly not pretty, but it does a job. The first section gets some information about the size of the screen. Next the X value of the first rock in increased by ten. Then we test to see if the rock has gone off of the right side of the screen, and if it has we call some methods to assign new random values for the rock, back on the left hand side of the screen. This is repeated for each rock.

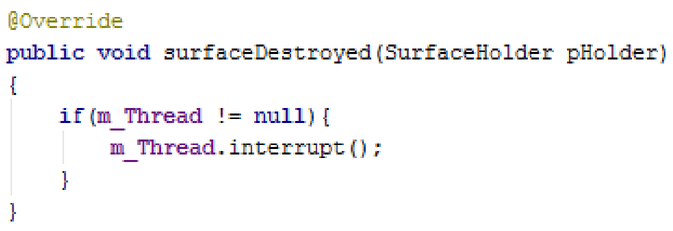
As in the previous example we need to update the display this on a new thread. The difference is that with a surface view we are also able to take control of the UI and render from the new thread too. In SeaView create a Thread member variable called m_Thread. We want to setup and start that thread just after the SurfaceView is created, so the perfect place is the surfaceCreated method. Replace the current code with the following code



```
@Override
public void surfaceCreated(SurfaceHolder pHolder){
    m_Thread = new Thread(new Runnable(){
        public void run() {
            Canvas canvas = null;
            while (!Thread.currentThread().isInterrupted()) {
                MoveRocks();
                canvas = m_SurfaceHolder.lockCanvas();
                canvas.drawColor(Color.parseColor("#75a2d9"));
                canvas.drawBitmap(m_RockBitmap, m_Rock1X, m_Rock1Y, m_Paint);
                canvas.drawBitmap(m_RockBitmap, m_Rock2X, m_Rock2Y, m_Paint);
                canvas.drawBitmap(m_RockBitmap, m_Rock3X, m_Rock3Y, m_Paint);
                m_SurfaceHolder.unlockCanvasAndPost(canvas);
            }
        }
    });
    m_Thread.start();
}
```

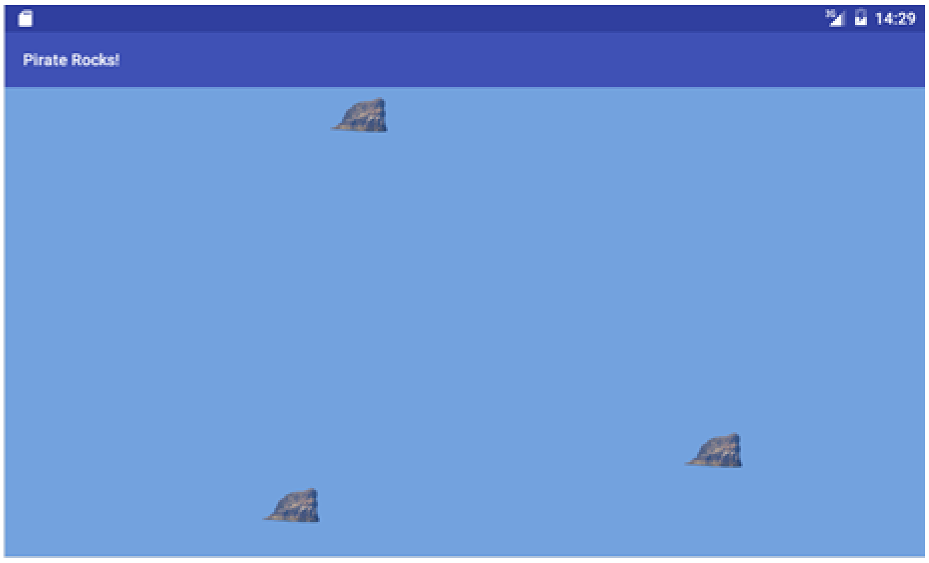
This code creates a new Thread instance and assigns it to m_Thread. In the new Thread's run method we check the thread has not been interrupted, then we move the rocks, the get hold of the canvas, draw the three rocks on a blue background and the release the lock and post.

The last piece of this puzzle is to modify the surfaceDestroyed method. When the SurfaceView is created we create a new thread. When the SurfaceView is destroyed we should stop the thread. To do that use this code:



```
@Override
public void surfaceDestroyed(SurfaceHolder pHolder)
{
    if(m_Thread != null){
        m_Thread.interrupt();
    }
}
```

Run your code. You should see rocks moving from left to right on a blue background.



Commit your code with an appropriate message like *"Completed lab 5.1 task 5 Animated rocks on a SurfaceView!"*

Next we want to add a ship that we can control using touch. First of all in SeaView create a private Bitmap member variable called m_ShipBitmap. Add two float member variables called m_ShipX and m_ShipY to keep track of the ships position. In the SeaView constructor get some content for the m_ShipBitmap member variable using the R.drawable.ship resource.

Once you've done that little bit of housekeeping we can look more carefully at how to implement touch. First of all in the onCreate method of the SeaActivity we need to set a touch listener and use it to pass the touch event to the instance of SeaView. Add the following code to the onCreate method:

```
relativeLayout.setOnTouchListener(new View.OnTouchListener() {
    @Override
    public boolean onTouch(View v, MotionEvent event) {
        return m_SeaView.onTouch(v, event);
    }
});
```

At the moment the onTouch method call is red and angry. That's because at the moment SeaView does not have an onTouch method to call. In the SeaView class add that method like this:

```
public boolean onTouch(View pView, MotionEvent pEvent){
    if (pEvent.getAction() == MotionEvent.ACTION_DOWN || pEvent.getAction() == MotionEvent.ACTION_MOVE){
        m_ShipX = pEvent.getX();
        m_ShipY = pEvent.getY();
        return true;
    }
    return false;
}
```

This method checks to see if the event it has received is a finger touching the screen or a move event. If it is then the member variables that manage the ships position are updated with the position of the ***first*** pointer down. If that happens the event is consumed and the method returns true. Otherwise the method returns false and the event can be resolved elsewhere. The final step is to draw the ship in the surfaceCreated method. Add a method to do that using the rocks as a template. Run the code to see if it works. When it does commit your code to SVN with an appropriate log message like *“Completed lab 5.1 task 6 Can drag a ship around the sea!”*.

You’ve probably notices that the ship doesn’t follow your input exactly, but rather the top left point of the bitmap representing your ship does. Can you figure out how to centre the ship on your touch? When you do commit your code to SVN with an appropriate log message like *“Completed lab 5.1 task 7 Centred the ship to the touch pointer!”*

At this point you could spend some time finishing off the game, adding a consequence if the ship gets too close to the rocks. All that would be good practice, but you might not learn very much more about andoird. Feel free to finish the game as you see fit, and move on when you are ready. Remember to check in when you are done.

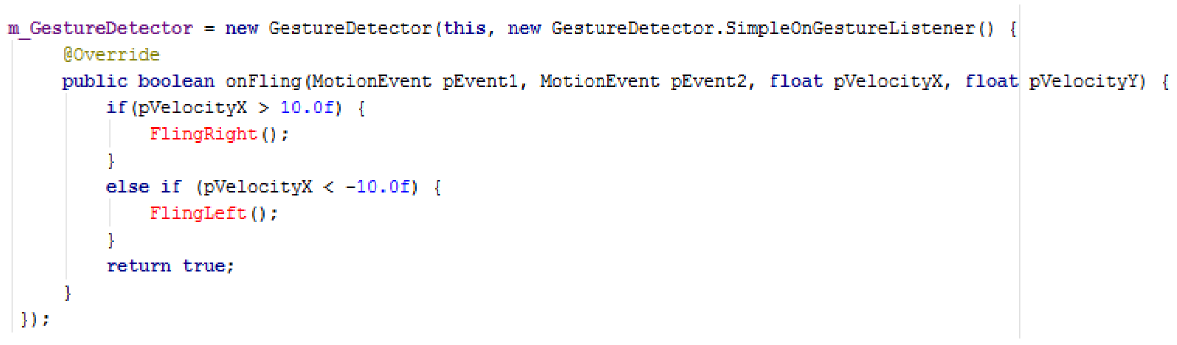
In the next section we’re going to add gesture control to an application called plunder. Pirates are shown pictures that either treasure or trash. If there are treasure swipe right to increase your score. If there are trash swipe left.

Check out the folder **<your repository>**\Week5\Lab_5_2_Plunder to C:\Temp and open this application in Android Studio. When you’ve finished remember to commit your changes and delete your work from the local drive. Run the application so you can see what is going on.

Take a look at the code. Most of the work has been done, but just the gesture control remains. Once you get past the title screen a ViewAnimator called ViewFlipper is being used to switch between two images. You can see this in the activity_plundering.xml file. Towards the bottom of the class are four methods that return Animation objects and are used to control how the ViewFlipper transitions between the two ImageViews that are contain within it.

To detect a gesture we need to create an instance of the GestureDetector class. Create this as a member variable of the Plundering Activity class. Call it m_GestureDetector.

Next, we need to set the GestureDetector with a SimpleOnGestureListener. In the listener we want to pick up onFling gestures. We want to check to see if the fling gesture was fast enough, and then call either FlingRight or FlingLeft. Here is how to do that in the onCreate method:



```java
m_GestureDetector = new GestureDetector(this, new GestureDetector.SimpleOnGestureListener() {
    @Override
    public boolean onFling(MotionEvent pEvent1, MotionEvent pEvent2, float pVelocityX, float pVelocityY) {
        if(pVelocityX > 10.0f) {
            FlingRight();
        }
        else if (pVelocityX < -10.0f) {
            FlingLeft();
        }
        return true;
    }
});
```

FlingRight and FlingLeft are highlighted because they don’t exist yet, so the next step is to create them.

```java
public void FlingRight() {
    m_ViewFlipper.setInAnimation(inFromLeftAnimation());
    m_ViewFlipper.setOutAnimation(outToRightAnimation());
    addCurrentScore(m_CurrentImageValue);
    if(m_CurrentImageView == 1) {
        m_CurrentImageView = 2;
        m_CurrentImageValue = SetRandomImage(m_ImageView2);
    }
    else
    {
        m_CurrentImageView = 1;
        m_CurrentImageValue = SetRandomImage(m_ImageView1);
    }
    m_ViewFlipper.showPrevious();
}

public void FlingLeft() {
    m_ViewFlipper.setInAnimation(inFromRightAnimation());
    m_ViewFlipper.setOutAnimation(outToLeftAnimation());
    if(m_CurrentImageView == 1) {
        m_CurrentImageView = 2;
        m_CurrentImageValue = SetRandomImage(m_ImageView2);
    }
    else
    {
        m_CurrentImageView = 1;
        m_CurrentImageValue = SetRandomImage(m_ImageView1);
    }
    m_ViewFlipper.showPrevious();
}
```

Last but by no means least we need to override the onTouchEvent for the PlunderingActivity. In this method we simply delegate to our GestureDetector object like this:

```java
@Override
public boolean onTouchEvent(MotionEvent pEvent) {
    return m_GestureDetector.onTouchEvent(pEvent);
}
```

Run the code. Now you've detected gestures the application should be fully functional. If the code works commit your code with an appropriate log message like *"Completed lab 5.2 task 1 Created a gesture detector to detect a simple fling gesture"*. When testing remember the a fling ends with your finger off of the screen, so you should do the same with the mouse when testing.

Summary

In this lab you have learnt about animations, touch and gestures. You've created a simple animation by updating a custom view from a background thread and then requesting a redrawing using postInvalidate. You've created a smoother and more flexible animation by extending the SurfaceView class, and drawing to the canvas safely from a thread other than the UI thread. You've learnt how to listen for and respond to touch events, and finally you've used a gesture detector to listen for and respond to a simple fling gesture.