4. Account for ambient mode

We'll make sure to have long lasting fun by making the battery last longer.

React to ambient mode

Aside from the interactive mode, Android Wear also has an ambient mode. This mode helps the device to conserve power when it's not actively used. We typically recommend developers use black, white, and grays in this mode. Developers may also use limited color but the design should clearly signal that the watch is in ambient mode.

Aside from color differences, another way that ambient mode is different is that the watch face will only be updated once a minute. As a result, screen elements which update more often such as animation or seconds hand should be removed in this mode.

React to ambient mode

Within the `MyWatchFace.Engine` class, there is a method called `onAmbientModeChanged`. This method will be called when the watch is going into or out of ambient mode. This gives you a chance to change the design.

For our watch face, we will do three things to the watch face when it does go into ambient mode:

1. Change the background to be grayscale
2. Remove anti-aliasing of the watch hand paints
3. Remove the seconds hand

Change the background to be grayscale

This can be done by creating a copy of the bitmap and applying a grayscale filter to it:

Create a new Bitmap object called `mGrayBackgroundBitmap` in `MyWatchFaceService.Engine`.

Make a copy of the background bitmap and apply a grayscale filter to it. We recommend doing this in a separate method as good practice in clean code. We'll invoke this new method `initGrayBackgroundBitmap()` at the end of the method `onSurfaceChanged`. The method

should do the following:

```
private void initGrayBackgroundBitmap() {
    mGrayBackgroundBitmap = Bitmap.createBitmap(mBackgroundBitmap.getWidth(),
        mBackgroundBitmap.getHeight(), Bitmap.Config.ARGB_8888);
    Canvas canvas = new Canvas(mGrayBackgroundBitmap);
    Paint grayPaint = new Paint();
    ColorMatrix colorMatrix = new ColorMatrix();
    colorMatrix.setSaturation(0);
    ColorMatrixColorFilter filter = new
    ColorMatrixColorFilter(colorMatrix);
    grayPaint.setColorFilter(filter);
    canvas.drawBitmap(mBackgroundBitmap, 0, 0, grayPaint);
}
```

In `onDraw`, replace the background image draw code with the following which takes into account of ambient modes:

```
if (mAmbient) {
    canvas.drawBitmap(mGrayBackgroundBitmap, 0, 0, mBackgroundPaint);
} else {
    canvas.drawBitmap(mBackgroundBitmap, 0, 0, mBackgroundPaint);
}
```

Remove anti-aliasing of the watch hand paints

To do this, we add the following code into `onAmbientModeChanged` between `mAmbient = inAmbientMode;` and `invalidate();` to switch anti-aliasing on and off.

```
if (inAmbientMode) {
    mHandPaint.setAntiAlias(false);
} else {
    mHandPaint.setAntiAlias(true);
}
```

Remove the seconds hand

Here we update the code in `onDraw` to put a condition to check if `mAmbient` is `false` around the code that rotates and draw the seconds hand. This ensures that the seconds hand is only drawn in interactive mode. This is because the watch face is usually only updated once a minute in ambient mode.

Run the watch face

- If you have an **emulator**, press F7 on your keyboard to toggle ambient mode.
- If you have a **physical device**, cover the display with your hand.

Account for the peek card in ambient mode

In the image above, you can see that there is a peek card saying "0 steps to...". In this case, the interference with the watch face has not been too bad but in other cases, it may lead to misreading those messages. To avoid this, we recommend that developers at least put a black rectangle behind the peek card like this:

Create a Rect variable in `MyWatchFaceService.Engine` called `mCardBounds`. Remember to initialise it with `new Rect()`. Otherwise, when the code first refers to it, it will be `null` and the watch face will crash.

Override `onPeekCardPositionUpdate` method in `MyWatchFaceService.Engine`.

```
@Override
public void onPeekCardPositionUpdate(Rect rect) {
    super.onPeekCardPositionUpdate(rect);
    mCardBounds.set(rect);
}
```

Draw a black rectangle below the peek card by adding the following code at the end of the `onDraw` method:

```
if (mAmbient) {
    canvas.drawRect(mCardBounds, mBackgroundPaint);
}
```

If you run the watch face now, it should look like this:

Pro-tip

The `onPeekCardPositionUpdate` actually works in both ambient mode and interactive mode. In interactive mode, the system software takes care of drawing a rectangle behind the card so no work is required. You might ask what is the use of `onPeekCardPositionUpdate` in interactive mode? With certain designs, by knowing the card boundary you can actually shrink the watch face to the top visible portion of the screen and eliminate all interference.

Account for special screens

Some Android Wear watches can have special ambient modes. In order to make the most out of these features we need to know about them and account for them in our code.

Support low bit mode

In this mode pixels can only be switched on or off. Since each pixel consists of red, green and blue, it is possible to create some color in this mode but it will be very basic and may not look its best. We recommend developers think about black and white images which may work, or switch off the background altogether.

Take care of burn in protection

Some screens such as OLED suffer burn in if the same image is displayed constantly. As a result, we recommend that no more than 5% of the screen should be lit in ambient mode for these watches. Any solid filled areas should be hollowed out. Lastly, for analog watch faces, the center of the watch arms should be hollow so as not to burn in the center pixels.

To keep things simple, we will switch off the background if either of these modes are detected:

We will create two `boolean` member variables in `MyWatchFaceService.Engine` to keep track of 1) the low-bit screen mode: `mLowBitAmbient` and 2) the burn-in protection screen mode: `mBurnInProtection`.

Add and override the method `onPropertiesChanged` in `MyWatchFaceService.Engine` like this:

```
@Override
public void onPropertiesChanged(Bundle properties) {
    super.onPropertiesChanged(properties);
    mLowBitAmbient = properties.getBoolean(PROPERTY_LOW_BIT_AMBIENT, false);
    mBurnInProtection = properties.getBoolean(PROPERTY_BURN_IN_PROTECTION, false);
}
```

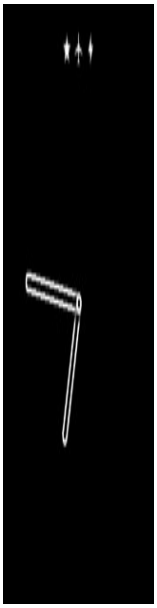
Replace the background `drawBitmap` code in `onDraw` with the following conditional code which only draws the background if these two modes are not detected in ambient mode:

```
if (mAmbient && (mLowBitAmbient || mBurnInProtection)){
    canvas.drawColor(Color.BLACK);
} else if (mAmbient) {
    canvas.drawBitmap(mGrayBackgroundBitmap, 0, 0, mBackgroundPaint);
} else {
    canvas.drawBitmap(mBackgroundBitmap, 0, 0, mBackgroundPaint);
}
```

Also as a refinement to the code above, we can put a condition to only create the grayscale bitmap if the watch is not in low bit or burn in protection mode:

```
if (!mBurnInProtection || !mLowBitAmbient) {
    initGrayBackgroundBitmap();
}
```

If you run this code on a supported device (e.g. *Sony Smartwatch 3* for low-bit ambient mode and *LG G Watch R* for burn-in protection mode), it should look like the following picture. For the purpose of this code lab, it is okay not to test this. For production, we recommend that you test against devices with these different screens.



Summary

In this step you've learned about:

- Taking ambient mode into account
- Making a peek card visible in ambient mode
- Building watch faces for special screens.

Next up

An optional activity to learn about how we can add automatic color selection into our code, so that the watch hand color will automatically work with the background.

If you still have time but don't fancy having a go at the palette API, we encourage you to alter the different parameters of the screen elements, for example, stroke size, the radius of the watch arms, color of the various screen element, etc.

Let's see what you get!