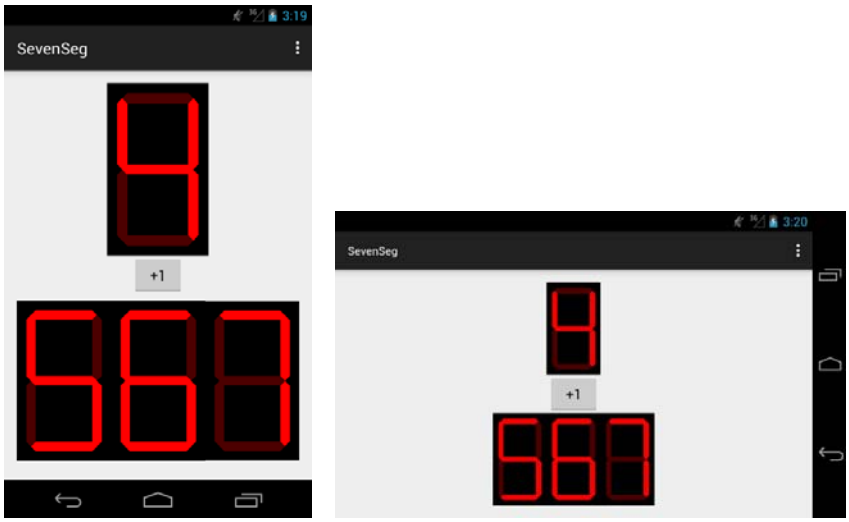
# CSCD 372: Android Development
# Lab 4: Custom View

## Lab Overview

This lab covers the creation of a custom, scalable view, with a constrained aspect ratio. It seems to me that making custom views that automatically scale to the size available, but with a fixed aspect ratio, would be a common desire for Android, but it was not trivial figuring out how to accomplish this. For this lab you will create a custom view implementing a seven-segment display. It will size to whatever space is allowed by its container, within the constraints of an aspect ratio that it enforces.

Below I show a test with 4 instances. The top instance is in a vertical linear layout and the three bottom instances are in a horizontal linear layout inside the outer vertical linear layout. The top and bottom row are weighted to share the available height equitably, and the bottom 3 share the available width. See the accompanying file with a sketch of the layout hierarchy. In the portrait orientation the height of the top instance is constrained by the available (shared) height, and the width by the aspect ratio. The bottom instances are being constrained by the available (shared) width, and the height by the aspect ratio. The bottom instances are not quite as tall as the top instance because the aspect ratio constraint will not allow them to be. As a consequence they do not occupy all of the height that is available to their container.

In the landscape orientation, the height of all instances is constrained by the available height, and the widths by the aspect ratio. In this case the heights of the bottom instances are the same as the top instance.




As usual, following are suggestions and hints for how to proceed.

## Task 1: Create a project shell and an initial SevenSegment class that draws only one segment

1. Start with the Blank Activity template. Please use the standard naming convention for Lab projects.
2. Add a java class called SevenSegment, and have it extend View. You're probably going to want at least two member variables: one for the current value being displayed (an int), and another array that stores the on/off state of each of the seven segments. Yes, these duplicate information that could be recreated from each other, but it would be inconvenient to do that on the fly.

3. You will want to implement the 3 constructor versions for custom views discussed in module 7. Each should call the superclass version and perform any other initialization that you find yourself needing. Don't duplicate the initialization code: collect it into a common private method called by all 3.

4. Sit down and sketch out the 7 polygons of a seven segment display on a piece of graph paper. The points and the angled sides of the points should abut with adjacent segments. The polygons should all be the same size. You can pick any scale for your coordinate system that you like, because the drawing will be rescaled at render time. However, if the dimensions you design to are small, the segments may come out blurry when you scale up. If that happens, turn off hardware acceleration as discussed in step 6b below. Put a reasonable margin around your drawing. The containing box tells you the aspect ratio that you will be enforcing. My results were a 20x34 box including a 2 unit margin all around, but I would recommend making your coordinate system an order of magnitude larger, like 200 x 340.

5. Pick an origin and record the vertices of the polygon for one of your segments. I put these into a final array of floats. I ended up re-coding these into a single array of alternating x,y values after I discovered how points are passed to one of the drawing methods. You can do otherwise, but this is the ONLY set of polygon vertices that I hardcoded. I draw all my segments by drawing this segment 7 times, with different coordinate system translations and rotations.

6. I suggest that at this point you override onDraw() and see if you can draw that single segment onto the Canvas without scaling. Here are some hints for that:

   a. The first thing you'll need to draw a filled polygon is a Path object. The path is defined by a moveTo() your start point, followed by a succession of lineTo() calls around the polygon, and finishing with a close(), which will connect the last point back to the first. I have a method called makePath() that does this from an array of points. Although I don't recommend this, you can avoid the need for the Paint object required by drawPath() by instead using the Path to define a clipping region (clipPath, pay attention to the second argument) and then calling drawColor(), which fills the Canvas (or the currently clipped region), with a color. I used bright red, Color.rgb(255,0,0) for the on state, and a dark red, Color.rgb(76,0,0), for the off state, so that off segments are still somewhat visible, as in a real device. Those are also final variables in my class.

   b. For most phones, using clipPath to draw the segment requires that you turn off hardware acceleration in order for the segment to come out properly, because hardware accelerators will simplify a clipping region to the surrounding rectangle. You can turn off hardware acceleration by calling: setLayerType(this.LAYER_TYPE_SOFTWARE, null). If you do this, you should probably not call this from onDraw(), but from your constructors instead (ask me why if you're curious).

   c. On your main layout, drop a Custom View widget, and just give it a fixed width and height for now. As soon as you select a Custom View, it will ask you to pick from those it finds, and your class should be one of them. If your design dimensions are as small as mine are, this initial rendering is going to come out tiny. But it's a start.

## Task 3: Scale the drawing and finish the other segments

1. Probably next you should try scaling that single segment up. Your initial effort at this may only work when you run on the emulator or an actual device. It may not come out proper in the layout design preview window, but we'll fix that next.

2. In onDraw(), call getWidth() and getHeight() on the Canvas and divide those by your design dimensions in order to get a scale factor for each dimension. You'll want to cast the individual ints to

floats for that. Then call canvas.scale() to make subsequent drawing activities scale up. I wouldn't try to force any aspect ratio constraint at this point. We'll do that later in a way that ensures the container is aware of the size resulting from the constraint (constraining the scale in onDraw won't accomplish that).

3. If you're going to, now is a good time to start using the translate() and and rotate() calls to draw the other segments. Try adding only a second segment at first, as it makes a difference whether you rotate() first and then translate() or visa versa. Note also that these coordinate transformations are concatenated together (the resulting matrices are all saved, and multiplied one after the other). In other words, two successive calls to rotate(90) will result in a 180 degree rotation. If that becomes too confusing, you can restore a transformation matrix to an earlier state using the save() and restore() calls. I did a save() after the scaling call, then a restore() and save() before each successive segment is drawn, so that I know I am starting from just the initial scaling. Keep at it until you've got all 7 segments rendered. If you must do it the inelegant way, you can always encode 7 different path arrays.
4. While debugging this, you may have to test on the emulator or an actual device. If it looks too obnoxious in your preview window, which is likely, use the isInEditMode() trick shown in class to force it to a dummy rendering of a gray box for the preview.
5. Once you have all segments drawing correctly, try a couple of different sizes on your layout to make sure it is scaling properly.

## Task 4: Fix the preview problem

6. Your onDraw method is called by the design preview, but calls you make to getWidth() and getHeight() on the canvas do not return valid results at this point (at least not at the time this was written). There is another way to get those that will work in preview mode as well as live mode. Override the onSizeChanged() method, call the superclass version, and record the incoming width and height (the new values, not the old values) to some member variables. Then use those values in your onDraw() scaling instead of calling getWidth() and getHeight(). It should work in preview mode now.

## Task 3: Finish the SevenSegment state and interface

1. You'll need set() and get() methods for the value currently being displayed. The set() should constrain the value to the range 0 to 10. Use the value 10 to indicate the all off state, which should be the initial state of the display. The set() method will also need to set the on/off state of the segments, which you should be holding in an array of booleans. I use a lookup table implemented as a final 2D array to help me convert from the passed int value to the array of segment states. The first (zeroth) row of that 2D array contains 7 entries that tell me which segments are on and off for the value 0, row 1 gives me 7 on/off states for the value 1, etc. Row 10 contains all segments off.
2. Drop a button onto your layout and use it to circularly increment the value from 0 through 10 to make sure all values display properly. Be smart and do this using the modulo operator, rather than a big case statement.

## Task 5: Enforce the aspect ratio constraint

1. Override the onMeasure() method. Your container calls this to allow the View to determine the final measured width and height. The contract for onMeasure() requires that you call setMeasuredDimension() in response to the call. The ints that are passed in are not actually dimensions. They are encoded View.MeasureSpec's, from which you can extract the Mode being enforced by the container (UNSPECIFIED, EXACTLY, or AT_MOST) along with the container's

allocated dimensions. We don't need to look at the mode here. Use the static methods MeasureSpec.getSize() to obtain the width and height being allocated by the container.

2. What you want to do here is to use as much of either the allocated height or width as possible, without violating your aspect ratio constraint. There are variations on how to do this, but here is what I do. I calculate a potential width by multiplying the allocated height by my aspect ratio, and calculate a potential height by dividing the allocated width by my aspect ratio. Then, if the potential height is larger than the allocated height, I set the final height to the allocated height, and the final width to the potential width. Visa-versa otherwise. Call setMeasuredDimension() with the final width and height.
3. Test the result by reproducing my test case shown in the intro. You can put 4 instances in the bottom row to make the differing height from the top row more obvious. Have your increment button rotate through the range circularly, but with a +1 offset between instances, as shown above. Some hints on the layouts:
    a. Change the generated RelativeLayout to a Linear Layout in the vertical dimension with match_parent width and height.
    b. Set the width on the top instance to match_parent, and the height to 0dp, with a layout_weight of 1. Likewise for the horizontal linear layout at the bottom. The button should get wrap_content in both dimensions, and no weight. This way the top instance and bottom row share all available height not taken up by the button. The bottom linear layout should also set a gravity (not layout_gravity) of "center".
4. Test your solution. Increment the displays and then rotate. If the displays reset, then you need to fix that ...

## Task 5: Ensure that the custom views retain state through configuration changes

1. Override the View.onSaveInstanceState() and onRestoreInstanceState() to save the state. You should only need to save the member variables that encode the value displayed. I suggest you do this in a Bundle stuffed into the Parcelable as discussed in class. Note that there are putIntArray() and getIntArray() methods available. Note also that on restoration, Android will construct your view, then call onRestoreInstanceState() with the Parcelable, and then redraw your View, so you needn't worry if you're initializing your state to 10 on construction.
2. Retest rotations to make sure it works. Submit your apk file along with the java source for your custom view and the xml file for your main layout. Don't forget to include an About box and make sure it is accessible.