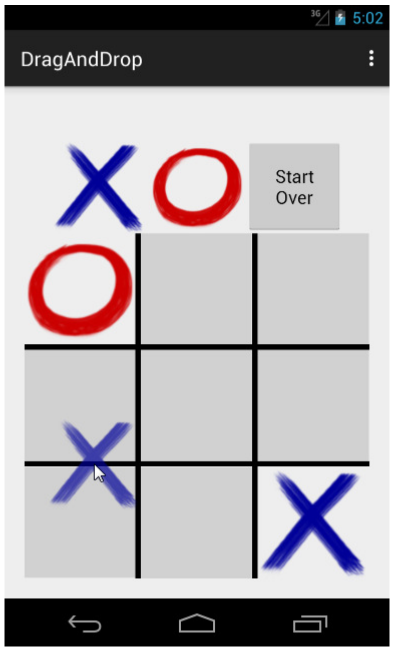
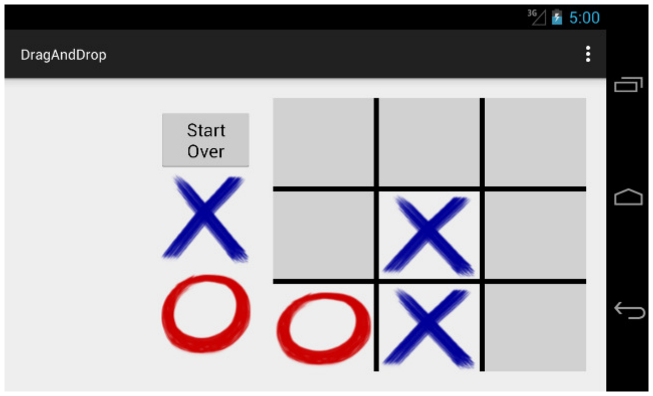
CSCD 372: Android Development
Lab 8: Drag and Drop

Lab Overview

This lab shows you how to use Android Drag and Drop, using a tic-tac-toe game board as an inspiration for that. The user takes a turn by dragging the X or the O onto an open spot on the game board. Our example is a bit different from the usual examples of drag and drop that you'll find on the web, as those invariably move a view from one container to another. We won't actually move the X or the O, but will instead "copy" it when it is dropped. The end result will be only a few short steps from a full implementation of the game. The left screen capture below was taken as an X was being dragged towards the bottom left corner. The alternative to drag and drop for such activities would be to require the user to touch on a piece and then touch on a destination. Boring. You don't need to implement the "Start Over" button you see in these screen captures:





This topic is not covered in lecture. I give you what you need to know in these instructions, but you'll want to be thinking about what I'm asking you to do and may want to look at the API docs for details of certain method calls (although Android Studio will prompt you for parameter types, the intent is not always obvious).

Task 1: A simple auto-scaling layout for the board

1. Start with the Blank or Empty Activity wizard. Either create your own images for the X, O, a blank cell, and the boundary lines, or download the zip set from the course website. We'll pursue a layout for the board that autoscales to the space available while maintaining a square aspect ratio, and I'll show you a way to do it without having to create a full custom view like we did in a previous lab. You don't want to implement your tic-tac-toe grid in a single view here because drag and drop is much harder to accomplish within a single view. Trust me, I know. Make sure your root layout is RelativeLayout. In what follows, you could replace the TableLayout with a Vertical LinearLayout and each TableRow with a Horizontal LinearLayout.

2. Drop a TableLayout onto your RelativeLayout. Then put a TableRow into that TableLayout. This will be the top row of your board. Give this TableRow a width of match_parent, a height of 0, and a weight of 1. Make sure the weightSum property for both is blank.
3. Into that TableRow, place 5 ImageViews. Set the src property of the first, third, and fifth to the blank drawable, and the second and fourth to the vertical line drawable. Each drawable should have a height of match_parent and a width of 0. The weight for each drawable should then be made roughly proportional to the actual drawable widths (e.g., if you specify a weight of 1 for the playing pieces, then the vertical dividers should be about 0.05).
4. Drop another TableRow under the previous one. Give it a width of match_parent, a height of 0, and a weight that is in the same proportion as the height of a horizontal line image should be to the height of a playing piece. Since we set the weight of the previous row to 1, a weight of about 0.05 would be appropriate here. Into this row place just a single horizontal drawable. Set the height to match_parent and the width to 0. Give it a weight of 1.
5. Put in 3 more TableRows, following the approach in step 4 or 5 as appropriate.
6. At this point your board should fill the screen. Here is a way to force it to a square aspect ratio: Download the java source for AspectRatioImageView from the assignment page and add it to your project. Then place a corresponding Custom view widget into your layout just above your TableLayout. Then use the RelativeLayout constraints to line up all 4 edges of the TableLayout with that Custom view. Viola.
7. Note that the only code in AspectRatioImageView is that which forces a scaling with aspect ratio, similar to what you did in your previous lab. You'll also note that the aspect ratio defaults to 1. You could force the width to be half the height by adding the attribute custom:aspectRatio="0.5f" into the XML header for the CustomView (if you do, you'll likely be prompted to add a schema for "res-auto" to the outer layout header).
8. Add a couple more ImageViews to hold the X and O playing pieces, as shown above. Arrange them to look half-decent, i.e. aligned in some sense. You might want to put a bit of margin around the board as well. We're not animals.

Task 2: Implement OnTouchListener

1. Modify your Activity class so that it implements OnTouchListener. In your onCreate() method, register as the listener for the X and O playing pieces. When one of these is touched you will be initiating a drag operation on it.
2. When either the X or the O is dropped onto a blank cell, we will want to change the image for that cell to the corresponding mark. You could set a member variable that identifies the object being dragged, but an easier approach is to have the object being dragged carry along information about the resource image that it is using, so that we can copy that from it once it is dropped. So, in addition to registering as a listener, set a tag on those objects to the corresponding drawable resource id.
3. For your onTouch() method, if the incoming action is ACTION_DOWN, create a new DragShadowBuilder object, passing it the incoming view (which is the view that was touched).
4. Then call startDrag() on the view, which takes the following arguments:
   a. A clipping region on the portion of the view to be dragged. That can be null in this case, as we will be dragging an image of the entire ImageView.
   b. The ShadowBuilder object.
   c. A "local state" object intended to carry any necessary information to the drop point. It is entirely up to the programmer whether to use this and how. We could have used this field to carry the drawable id, instead of using a Tag attached to the ImageView, but attaching a Tag to each mark at

creation time saves us from having to figure out which item is being dragged. So in this case I simply pass the View being dragged.

   d. An integer flag. This is apparently for future enhancement as no flags are currently defined, so just pass a 0.

5. For the ACTION_DOWN case, return a true, indicating that you have handled this touch event. Otherwise return a false in order to allow other listeners to take action for other types of touch events.

   BTW, for some drag and drop applications you might want the touched view to disappear while its shadow is being dragged, so that it would appear to be moving. In that case you would call setVisibility(View.INVISIBLE) on it. In this case we don't want that (it's more like a copy).

Task 3: Implement OnDragListener

1. Modify your Activity class so that it implements OnDragListener.
2. Register a listener with the 9 ImageViews that make up the cells of the grid (I made my Activity the listener and used the same listener for all 9 cells). This will result in a call to the onDrag() method when an object is being dragged or dropped over any of these cells. As long as you're talking to the ImageViews, put a tag on each containing its current drawable id as well. You may or may not end up needing that, depending on a choice you make below, but for sure you would find that convenient if you were to finish coding a complete game of tic-tac-toe (which isn't being required here).
3. In onDrag(), if the incoming event is ACTION_DROP, grab a reference to the view being dragged by calling getLocalState() on the event (we passed this when we called startDrag() above). Get the drawable ID that it is using from its Tag. Use that drawable id to change the image resource attached to the ImageView being dropped on, which is passed to onDrag() as the View object. Change the tag on that view to the same resource id, in case we ever need to determine the content of the cell (which may be soon). Return true from this method to indicate that you processed the event, even if you took no action. For example, you're not going to take any action in response to ACTION_DRAG_ENTERED, but you still need to return true in order to receive subsequent events that you're interested in (like ACTION_DROP).
4. Do a test run. Try dragging an X to a cell, and then try dragging an O over the top of an X. It should work, but we don't want to allow that.

Task 4: Disable cheating in the form of replacing a piece already placed

1. We have several options here. One easy thing would be to de-register as a drag listener for cells once they have had an X or an O dropped on them. Unfortunately, while there are method calls to unregister as a listener for some interfaces, OnDragListener is not currently one of them.
2. One could force that by removing the current ImageView object from the layout and replacing it with a new ImageView object, and simply neglecting to register as a listener for the new ImageView. In that case, however, we would have to find where the current ImageView (the one we're dropping on) is in the layout. That isn't particularly hard (I've tested it for feasibility), but it is simpler to leave the ImageView object alone and simply change its image resource, which we did above. Feel free to accomplish this however you like, but I recommend you do the following ...
3. In onDrag(), grab the Tag for the view. If it is anything other than the drawable for your blank cell, simply return without changing the resource ID. I also chose to throw up a Toast in this case, telling the user that they cannot play on this cell. It doesn't really matter whether you return false or true here, as this is a Drop event, which occurs at the end of a drag operation.

4. Do a test run. The user should now be prevented from overwriting a previous mark. Now try navigating away with the home button and back using the recent apps list (or simply rotate your screen and back). We now need to make sure the game board state is restored.

Task 5: Restore on configuration change

1. You hopefully noticed from an earlier lab that we can change the content of, say, an EditText, at runtime, and Android will restore it to that changed state on a configuration change for us. So why can't we change the resource attached to an ImageView and have Android restore it for us? Good question. It just doesn't, at least not at the time this was written. Apparently Android views these widgets as static in nature, so they are restored to what they were initialized to in the layout, if anything.
2. Oh well, override onSaveInstanceState() and walk through the 9 cells, storing their current drawables to the outgoing bundle. Then in onCreate(), if the bundle is not null, restore them to the ImageView. Don't forget to store them to the ImageView's Tag as well. You'll also need to register as a drag listener (for at least the blank cells, and for all if you're using my recommended approach). Test to make sure your restoration code works.

Task 6: Finish Up

1. I'm not requiring that you finish a full tic-tac-toe game implementation because the rest would not be particular to Android and I'd rather you put the time into your project. I am asking, however, that you generate a separate layout for landscape, is rather trivial. An obvious choice for landscape is to put the playing pieces to the side of the grid, rather than above it. An easy approach is to generate a new main layout (activity_main or content_main, depending on whether or not you're using an ActionBar or a Toolbar) and locate it in the layout-land directory. Then copy the XML from your other layout and make a few changes to the properties in order to get the two inner layouts to line up side-by-side.
2. Test to make sure you can see everything in the landscape orientation. Make sure you have an About Toast, and submit your main activity java source, the xml for either layout, and your apk file.