Displaying Bitmaps in Your UI

This lesson brings together everything from previous lessons, showing you how to load multiple bitmaps into `ViewPager` `(/reference/android/support/v4/view/ViewPager.html)` and `GridView` `(/reference/android/widget/GridView.html)` components using a background thread and bitmap cache, while dealing with concurrency and configuration changes.

THIS LESSON TEACHES YOU TO

1. Load Bitmaps into a ViewPager Implementation
2. Load Bitmaps into a GridView Implementation

YOU SHOULD ALSO READ

- Android Design: Swipe Views
- Android Design: Grid Lists

TRY IT OUT

Download the sample

DisplayingBitmaps.zip

Load Bitmaps into a ViewPager Implementation

The swipe view pattern (/design/patterns/swipe-views.html) is an excellent way to navigate the detail view of an image gallery. You can implement this pattern using a `ViewPager` `(/reference/android/support/v4/view/ViewPager.html)` component backed by a `PagerAdapter` `(/reference/android/support/v4/view/PagerAdapter.html)`. However, a more suitable backing adapter is the subclass `FragmentStatePagerAdapter` `(/reference/android/support/v4/app/FragmentStatePagerAdapter.html)` which automatically destroys and saves state of the `Fragments` `(/reference/android/app/Fragment.html)` in the `ViewPager` `(/reference/android/support/v4/view/ViewPager.html)` as they disappear off-screen, keeping memory usage down.

> **Note:** If you have a smaller number of images and are confident they all fit within the application memory limit, then using a regular `PagerAdapter` `(/reference/android/support/v4/view/PagerAdapter.html)` or `FragmentPagerAdapter` `(/reference/android/support/v4/app/FragmentPagerAdapter.html)` might be more appropriate.

Here's an implementation of a `ViewPager` `(/reference/android/support/v4/view/ViewPager.html)` with `ImageView` `(/reference/android/widget/ImageView.html)` children. The main activity holds the `ViewPager` `(/reference/android/support/v4/view/ViewPager.html)` and the adapter:

```
public class ImageDetailActivity extends FragmentActivity {
    public static final String EXTRA_IMAGE = "extra_image";

    private ImagePagerAdapter mAdapter;
    private ViewPager mPager;

    // A static dataset to back the ViewPager adapter
    public final static Integer[] imageResIds = new Integer[] {
            R.drawable.sample_image_1, R.drawable.sample_image_2, R.drawable.sample_image
            R.drawable.sample_image_4, R.drawable.sample_image_5, R.drawable.sample_image
            R.drawable.sample_image_7, R.drawable.sample_image_8, R.drawable.sample_image

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.image_detail_pager); // Contains just a ViewPager
        mAdapter = new ImagePagerAdapter(getSupportFragmentManager(), imageResIds.length)
        mPager = (ViewPager) findViewById(R.id.pager);
        mPager.setAdapter(mAdapter);
    }

    public static class ImagePagerAdapter extends FragmentStatePagerAdapter {
        private final int mSize;

        public ImagePagerAdapter(FragmentManager fm, int size) {
            super(fm);
            mSize = size;
        }

        @Override
        public int getCount() {
            return mSize;
        }

        @Override
        public Fragment getItem(int position) {
            return ImageDetailFragment.newInstance(position);
        }
    }
}
```

Here is an implementation of the details `Fragment` (/reference/android/app/Fragment.html) which holds the `ImageView` (/reference/android/widget/ImageView.html) children. This might seem like a perfectly reasonable approach, but can you see the drawbacks of this implementation? How could it be improved?

```
public class ImageDetailFragment extends Fragment {
    private static final String IMAGE_DATA_EXTRA = "resId";
    private int mImageNum;
    private ImageView mImageView;

    static ImageDetailFragment newInstance(int imageNum) {
        final ImageDetailFragment f = new ImageDetailFragment();
        final Bundle args = new Bundle();
        args.putInt(IMAGE_DATA_EXTRA, imageNum);
        f.setArguments(args);
        return f;
    }

    // Empty constructor, required as per Fragment docs
    public ImageDetailFragment() {}

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mImageNum = getArguments() != null ? getArguments().getInt(IMAGE_DATA_EXTRA) : -1
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
            Bundle savedInstanceState) {
        // image_detail_fragment.xml contains just an ImageView
        final View v = inflater.inflate(R.layout.image_detail_fragment, container, false)
        mImageView = (ImageView) v.findViewById(R.id.imageView);
        return v;
    }
    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        final int resId = ImageDetailActivity.imageResIds[mImageNum];
        mImageView.setImageResource(resId); // Load image into ImageView
    }
}
```

Hopefully you noticed the issue: the images are being read from resources on the UI thread, which can lead to an application hanging and being force closed. Using an `AsyncTask` (/reference/android/os/AsyncTask.html) as described in the Processing Bitmaps Off the UI Thread (process-bitmap.html) lesson, it's straightforward to move image loading and processing to a background thread:

```
public class ImageDetailActivity extends FragmentActivity {
    ...

    public void loadBitmap(int resId, ImageView imageView) {
        mImageView.setImageResource(R.drawable.image_placeholder);
        BitmapWorkerTask task = new BitmapWorkerTask(mImageView);
        task.execute(resId);
    }

    ... // include BitmapWorkerTask class
}

public class ImageDetailFragment extends Fragment {
    ...

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        if (ImageDetailActivity.class.isInstance(getActivity())) {
            final int resId = ImageDetailActivity.imageResIds[mImageNum];
            // Call out to ImageDetailActivity to load the bitmap in a background thread
            ((ImageDetailActivity) getActivity()).loadBitmap(resId, mImageView);
        }
    }
}
```

Any additional processing (such as resizing or fetching images from the network) can take place in the `BitmapWorkerTask` (process-bitmap.html#BitmapWorkerTask) without affecting responsiveness of the main UI. If the background thread is doing more than just loading an image directly from disk, it can also be beneficial to add a memory and/or disk cache as described in the lesson Caching Bitmaps (cache-bitmap.html#memory-cache). Here's the additional modifications for a memory cache:

```
public class ImageDetailActivity extends FragmentActivity {
    ...
    private LruCache<String, Bitmap> mMemoryCache;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        ...
        // initialize LruCache as per Use a Memory Cache section
    }

    public void loadBitmap(int resId, ImageView imageView) {
        final String imageKey = String.valueOf(resId);

        final Bitmap bitmap = mMemoryCache.get(imageKey);
        if (bitmap != null) {
            mImageView.setImageBitmap(bitmap);
        } else {
            mImageView.setImageResource(R.drawable.image_placeholder);
            BitmapWorkerTask task = new BitmapWorkerTask(mImageView);
            task.execute(resId);
        }
    }

    ... // include updated BitmapWorkerTask from Use a Memory Cache section
}
```

Putting all these pieces together gives you a responsive `ViewPager` (/reference/android/support/v4/view/ViewPager.html) implementation with minimal image loading latency and the ability to do as much or as little background processing on your images as needed.

Load Bitmaps into a GridView Implementation

The grid list building block (/design/building-blocks/grid-lists.html) is useful for showing image data sets and can be implemented using a `GridView` (/reference/android/widget/GridView.html) component in which many images can be on-screen at any one time and many more need to be ready to appear if the user scrolls up or down. When implementing this type of control, you must ensure the UI remains fluid, memory usage remains under control and concurrency is handled correctly (due to the way `GridView` (/reference/android/widget/GridView.html) recycles its children views).

To start with, here is a standard `GridView` (/reference/android/widget/GridView.html) implementation with `ImageView` (/reference/android/widget/ImageView.html) children placed inside a `Fragment` (/reference/android/app/Fragment.html). Again, this might seem like a perfectly reasonable approach, but what would make it better?

```
public class ImageGridFragment extends Fragment implements AdapterView.OnItemClickListene
    private ImageAdapter mAdapter;

    // A static dataset to back the GridView adapter
    public final static Integer[] imageResIds = new Integer[] {
            R.drawable.sample_image_1, R.drawable.sample_image_2, R.drawable.sample_image
            R.drawable.sample_image_4, R.drawable.sample_image_5, R.drawable.sample_image
            R.drawable.sample_image_7, R.drawable.sample_image_8, R.drawable.sample_image

    // Empty constructor as per Fragment docs
    public ImageGridFragment() {}

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mAdapter = new ImageAdapter(getActivity());
    }

    @Override
    public View onCreateView(
            LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
        final View v = inflater.inflate(R.layout.image_grid_fragment, container, false);
        final GridView mGridView = (GridView) v.findViewById(R.id.gridView);
        mGridView.setAdapter(mAdapter);
        mGridView.setOnItemClickListener(this);
        return v;
    }

    @Override
    public void onItemClick(AdapterView<?> parent, View v, int position, long id) {
        final Intent i = new Intent(getActivity(), ImageDetailActivity.class);
        i.putExtra(ImageDetailActivity.EXTRA_IMAGE, position);
        startActivity(i);
    }

    private class ImageAdapter extends BaseAdapter {
        private final Context mContext;

        public ImageAdapter(Context context) {
            super();
            mContext = context;
        }

        @Override
        public int getCount() {
            return imageResIds.length;
        }

        @Override
        public Object getItem(int position) {
            return imageResIds[position];
        }

        @Override
        public long getItemId(int position) {
            return position;
        }

        @Override
        public View getView(int position, View convertView, ViewGroup container) {
            ImageView imageView;
            if (convertView == null) { // if it's not recycled, initialize some attribute
                imageView = new ImageView(mContext);
                imageView.setScaleType(ImageView.ScaleType.CENTER_CROP);
                imageView.setLayoutParams(new GridView.LayoutParams(
                        LayoutParams.MATCH_PARENT, LayoutParams.MATCH_PARENT));
            } else {
                imageView = (ImageView) convertView;
            }
            imageView.setImageResource(imageResIds[position]); // Load image into ImageVi
            return imageView;
        }
    }
}
```

Once again, the problem with this implementation is that the image is being set in the UI thread. While this may work for small, simple images (due to system resource loading and caching), if any additional processing needs to be done, your UI grinds to a halt.

The same asynchronous processing and caching methods from the previous section can be implemented here. However, you also need to wary of concurrency issues as the `GridView` `(/reference/android/widget/GridView.html)` recycles its children views. To handle this, use the techniques discussed in the Processing Bitmaps Off the UI Thread (process-bitmap.html#concurrency) lesson. Here is the updated solution:

```
public class ImageGridFragment extends Fragment implements AdapterView.OnItemClickListene
    ...

    private class ImageAdapter extends BaseAdapter {
        ...

        @Override
        public View getView(int position, View convertView, ViewGroup container) {
            ...
            loadBitmap(imageResIds[position], imageView)
            return imageView;
        }
    }

    public void loadBitmap(int resId, ImageView imageView) {
        if (cancelPotentialWork(resId, imageView)) {
            final BitmapWorkerTask task = new BitmapWorkerTask(imageView);
            final AsyncDrawable asyncDrawable =
                    new AsyncDrawable(getResources(), mPlaceHolderBitmap, task);
            imageView.setImageDrawable(asyncDrawable);
            task.execute(resId);
        }
    }

    static class AsyncDrawable extends BitmapDrawable {
        private final WeakReference<BitmapWorkerTask> bitmapWorkerTaskReference;

        public AsyncDrawable(Resources res, Bitmap bitmap,
                BitmapWorkerTask bitmapWorkerTask) {
            super(res, bitmap);
            bitmapWorkerTaskReference =
                new WeakReference<BitmapWorkerTask>(bitmapWorkerTask);
        }

        public BitmapWorkerTask getBitmapWorkerTask() {
            return bitmapWorkerTaskReference.get();
        }
    }

    public static boolean cancelPotentialWork(int data, ImageView imageView) {
        final BitmapWorkerTask bitmapWorkerTask = getBitmapWorkerTask(imageView);

        if (bitmapWorkerTask != null) {
            final int bitmapData = bitmapWorkerTask.data;
            if (bitmapData != data) {
                // Cancel previous task
                bitmapWorkerTask.cancel(true);
            } else {
                // The same work is already in progress
                return false;
            }
        }
        // No task associated with the ImageView, or an existing task was cancelled
        return true;
    }

    private static BitmapWorkerTask getBitmapWorkerTask(ImageView imageView) {
       if (imageView != null) {
           final Drawable drawable = imageView.getDrawable();
           if (drawable instanceof AsyncDrawable) {
               final AsyncDrawable asyncDrawable = (AsyncDrawable) drawable;
```

```
                return asyncDrawable.getBitmapWorkerTask();
            }
        }
        return null;
    }

    ... // include updated BitmapWorkerTask class
```

Note: The same code can easily be adapted to work with `ListView` (/reference/android/widget/ListView.html) as well.

This implementation allows for flexibility in how the images are processed and loaded without impeding the smoothness of the UI. In the background task you can load images from the network or resize large digital camera photos and the images appear as the tasks finish processing.

For a full example of this and other concepts discussed in this lesson, please see the included sample application.