A Deep Dive into Custom Drawing in Android

This blog post was written with the assistance of ChatGPT-4o.

Introduction

In this blog, we' ll explore the `DrawActivity` class, a comprehensive example of implementing a custom drawing view in an Android application. We' ll break down each component and algorithm used, providing a detailed explanation of how everything works together to achieve the desired functionality.

Table of Contents

Overview of DrawActivity

`DrawActivity` is the main activity that handles drawing operations, image cropping, and interaction with other components like fragments and image upload. It provides a user interface where users can draw, undo, redo, and manipulate images.

```
public class DrawActivity extends Activity implements View.OnClickListener {
  // Constants for request codes and fragment IDs
  public static final int CAMERA_RESULT = 1;
  public static final int CROP_RESULT = 2;
  public static final int DRAW_FRAGMENT = 0;
  public static final int RECOG_FRAGMENT = 1;
  public static final int RESULT_FRAGMENT = 2;
  public static final int WAIT_FRAGMENT = 3;
```

```
  public static final int MATERIAL_RESULT = 4;
  public static final String RESULT_JSON = "resultJson";
  public static final int INIT_FLOWER_ID = R.drawable.flower_b;
  public static final int LOGOUT = 0;
  public static final int IMAGE_RESULT = 0;

  // Variables for handling image and drawing operations
  String baseUrl;
  DrawView drawView;
  Bitmap originImg;
  public static DrawActivity instance;
  View dir, clear, cameraView, materialView, scale;
  ImageView undoView, redoView;
  View upload;
  String cropPath;
  Tooltip toolTip;
  int curFragmentId = -1;
  int serverId = -1;
  private Bitmap resultBitmap;
  private RadioGroup radioGroup;
  Fragment curFragment;
  int curDrawMode;
  RadioButton drawBackBtn;
  private Activity cxt;
  Uri curPicUri;
}
```

Initializing the Activity

When the activity is created, various initializations are performed, such as setting up views, loading the initial image, and configuring event listeners.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  instance = this;
```

```
  cxt = this;
  cropPath = PathUtils.getCropPath();
  setContentView(R.layout.draw_layout);
  findView();
  setSize();
  initOriginImage();
  toolTip = new Tooltip(this);
  initUndoRedoEnable();
  setIp();
  initDrawmode();
}
```

findView()

This method initializes the views used in the activity.

```
private void findView() {
  drawView = findViewById(R.id.drawView);
  undoView = findViewById(R.id.undo);
  redoView = findViewById(R.id.redo);
  scale = findViewById(R.id.scale);
  upload = findViewById(R.id.upload);
  clear = findViewById(R.id.clear);
  dir = findViewById(R.id.dir);
  materialView = findViewById(R.id.material);
  cameraView = findViewById(R.id.camera);

  dir.setOnClickListener(this);
  materialView.setOnClickListener(this);
  undoView.setOnClickListener(this);
  scale.setOnClickListener(this);
  redoView.setOnClickListener(this);
  clear.setOnClickListener(this);
  cameraView.setOnClickListener(this);
  upload.setOnClickListener(this);
  initRadio();
}
```

setSize()

Sets the size of the drawing view.

```
private void setSize() {
  setSizeByResourceSize();
  setViewSize(drawView);
}

private void setSizeByResourceSize() {
  int width = getResources().getDimensionPixelSize(R.dimen.draw_width);
  int height = getResources().getDimensionPixelSize(R.dimen.draw_height);
  App.drawWidth = width;
  App.drawHeight = height;
}

private void setViewSize(View v) {
  ViewGroup.LayoutParams lp = v.getLayoutParams();
  lp.width = App.drawWidth;
  lp.height = App.drawHeight;
  v.setLayoutParams(lp);
}
```

initOriginImage()

Loads the initial image that will be used for drawing.

```
private void initOriginImage() {
  Bitmap bitmap = BitmapFactory.decodeResource(getResources(), INIT_FLOWER_ID);
  String imgPath = PathUtils.getCameraPath();
  BitmapUtils.saveBitmapToPath(bitmap, imgPath);
  Uri uri1 = Uri.fromFile(new File(imgPath));
  setImageByUri(uri1);
}
```

Handling Image Operations

The activity handles various image operations, such as setting the image by URI, cropping, and saving the drawn bitmap.

setImageByUri(Uri uri)

Loads an image from a given URI and prepares it for drawing.

```
private void setImageByUri(final Uri uri) {
  new Handler().postDelayed(new Runnable() {
    @Override
    public void run() {
      curPicUri = uri;
      Bitmap bitmap = null;
      try {
        if (uri != null) {
          bitmap = BitmapUtils.getBitmapByUri(DrawActivity.this, uri);
        }
      } catch (Exception e) {
        e.printStackTrace();
      }

      int originW = bitmap.getWidth();
      int originH = bitmap.getHeight();
      if (originW != App.drawWidth || originH != App.drawHeight) {
        float originRadio = originW * 1.0f / originH;
        float radio = App.drawWidth * 1.0f / App.drawHeight;
        if (Math.abs(originRadio - radio) < 0.01) {
          Bitmap originBm = bitmap;
          bitmap = Bitmap.createScaledBitmap(originBm, App.drawWidth, App.drawHeight, false);
          originBm.recycle();
        } else {
          cropIt(uri);
          return;
        }
      }
      ImageLoader imageLoader = ImageLoader.getInstance();
      imageLoader.addOrReplaceToMemoryCache("origin", bitmap);
      originImg = bitmap;
      serverId = -1;

      drawView.setSrcBitmap(originImg);
      showDrawFragment(App.ALL_INFO);
```

```
      curDrawMode = App.DRAW_FORE;
    }
  }, 500);
}
```

cropIt(Uri uri)

Starts an image cropping activity.

```
public void cropIt(Uri uri) {
  Crop.startPhotoCrop(this, uri, cropPath, CROP_RESULT);
}
```

saveBitmap()

Saves the drawn bitmap to a file and uploads it to the server.

```
public void saveBitmap() {
  Bitmap handBitmap = drawView.getHandBitmap();
  Bitmap originBitmap = drawView.getSrcBitmap();
  saveBitmapToFileAndUpload(handBitmap, originBitmap);
}
```

saveBitmapToFileAndUpload(Bitmap handBitmap, Bitmap originBitmap)

Saves bitmaps to files and uploads them asynchronously.

```
private void saveBitmapToFileAndUpload(Bitmap handBitmap, Bitmap originBitmap) {
  final String originPath = PathUtils.getOriginPath();
  BitmapUtils.saveBitmapToPath(originBitmap, originPath);
  final String handPath = PathUtils.getHandPath();
  BitmapUtils.saveBitmapToPath(handBitmap, handPath);
  new AsyncTask<Void, Void, Void>() {
    boolean res;
    Bitmap foreBitmap;
    Bitmap backBitmap;

    @Override
    protected void onPreExecute() {
      super.onPreExecute();
      showWaitFragment();
    }
```

```
@Override
protected Void doInBackground(Void... params) {
  try {
    if (baseUrl == null) {
      throw new Exception("baseUrl is null");
    }
    String jsonRes = UploadImage.upload(baseUrl, serverId, Web.STATUS_CONTINUE, originPath, handPat
    getJsonData(jsonRes);
    res = true;
  } catch (Exception e) {
    res = false;
    e.printStackTrace();
  }
  return null;
}

private void getJsonData(String jsonRes) throws Exception {
  JSONObject json = new JSONObject(jsonRes);
  if (serverId == -1) {
    serverId = json.getInt(Web.ID);
  }
  String foreUrl = json.getString(Web.FORE);
  String backUrl = json.getString(Web.BACK);
  String resultUrl = json.getString(Web.RESULT);
  foreBitmap = Web.getBitmapFromUrlByStream1(foreUrl, 0);
  backBitmap = Web.getBitmapFromUrlByStream1(backUrl, 0);
  resultBitmap = Web.getBitmapFromUrlByStream1(resultUrl, 0);
}

@Override
protected void onPostExecute(Void aVoid) {
  super.onPostExecute(aVoid);
  if (res) {
    showRecogFragment(foreBitmap, backBitmap);
  } else {
    Utils.toast(DrawActivity.this, R.string.server_error);
```

```
        recogNo();
      }
    }
  }.execute();
}
```

Fragment Management

The activity manages

different fragments to handle various states of the application, such as drawing, recognition, and waiting.

showDrawFragment(int infoId)

Displays the drawing fragment.

```
private void showDrawFragment(int infoId) {
  curFragmentId = DRAW_FRAGMENT;
  curFragment = new DrawFragment(infoId);
  showFragment(curFragment);
}
```

showWaitFragment()

Displays the waiting fragment.

```
private void showWaitFragment() {
  curFragmentId = WAIT_FRAGMENT;
  showFragment(new WaitFragment());
}
```

showFragment(Fragment fragment)

Replaces the current fragment with the specified fragment.

```
private void showFragment(Fragment fragment) {
  FragmentTransaction trans = getFragmentManager().beginTransaction();
  trans.replace(R.id.rightLayout, fragment);
  trans.commit();
}
```

Event Handling

The activity handles various user interactions, such as button clicks and menu selections.

onClick(View v)

Handles click events for different views.

```
@Override
public void onClick(View v) {
  int id = v.getId();
  if (id == R.id.drawOk) {
    if (drawView.isDrawFinish()) {
      saveBitmap();
    } else {
      Utils.alertDialog(this, R.string.please_draw_finish);
    }
  } else if (id == R.id.recogOk) {
    recogOk();
  } else if (id == R.id.recogNo) {
    recogNo();
  } else if (id == R.id.dir) {
    Utils.getGalleryPhoto(this, IMAGE_RESULT);
  } else if (id == R.id.clear) {
    clearEverything();
  } else if (id == R.id.undo) {
    drawView.undo();
  } else if (id == R.id.redo) {
    drawView.redo();
  } else if (id == R.id.camera) {
    Utils.takePhoto(cxt, CAMERA_RESULT);
  } else if (id == R.id.material) {
    goMaterial();
  } else if (id == R.id.upload) {
    com.lzw.commons.Utils.goActivity(cxt, PhotoActivity.class);
  } else if (id == R.id.scale) {
    cropIt(curPicUri);
  }
}
```

onActivityResult(int requestCode, int resultCode, Intent data)

Handles results from other activities, such as image selection or cropping.

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
  if (resultCode != RESULT_CANCELED) {
    Uri uri;
    switch (requestCode) {
      case IMAGE_RESULT:
        if (data != null) {
          setImageByUri(data.getData());
        }
        break;
      case CAMERA_RESULT:
        setImageByUri(Utils.getCameraUri());
        break;
      case CROP_RESULT:
        uri = Uri.fromFile(new File(cropPath));
        setImageByUri(uri);
        break;
      case MATERIAL_RESULT:
        setImageByUri(data.getData());
    }
  }
}
```

Undo and Redo Functionality

The activity provides undo and redo functionality for the drawing operations.

initUndoRedoEnable()

Initializes the undo and redo functionality by setting a callback.

```
void initUndoRedoEnable() {
  drawView.history.setCallBack(new History.CallBack() {
    @Override
    public void onHistoryChanged() {
      setUndoRedoEnable();
```

```
      if (curFragmentId != DRAW_FRAGMENT) {
        showDrawFragment(curDrawMode);
      }
    }
  });
}

void setUndoRedoEnable() {
  redoView.setEnabled(drawView.history.canRedo());
  undoView.setEnabled(drawView.history.canUndo());
}
```

Custom DrawView

`DrawView` is a custom view that handles drawing operations, touch events, and scaling.

onTouchEvent(MotionEvent event)

Handles touch events for drawing and scaling.

```
@Override
public boolean onTouchEvent(MotionEvent event) {
  if (!scaleMode) {
    handleDrawTouchEvent(event);
  } else {
    handleScaleTouchEvent(event);
  }
  return true;
}

private void handleDrawTouchEvent(MotionEvent event) {
  int action = event.getAction();
  float x = event.getX();
  float y = event.getY();
  if (action == MotionEvent.ACTION_DOWN) {
    path.moveTo(x, y);
  } else if (action == MotionEvent.ACTION_MOVE) {
    path.quadTo(preX, preY, x, y);
```

```
  } else if (action == MotionEvent.ACTION_UP) {
    Matrix matrix1 = new Matrix();
    matrix.invert(matrix1);
    path.transform(matrix1);
    paint.setStrokeWidth(strokeWidth * 1.0f / totalRatio);
    history.saveToStack(path, paint);
    cacheCanvas.drawPath(path, paint);
    paint.setStrokeWidth(strokeWidth);
    path.reset();
  }
  setPrev(event);
  invalidate();
}

private void handleScaleTouchEvent(MotionEvent event) {
  switch (event.getActionMasked()) {
    case MotionEvent.ACTION_POINTER_DOWN:
      lastFingerDist = calFingerDistance(event);
      break;
    case MotionEvent.ACTION_MOVE:
      if (event.getPointerCount() == 1) {
        handleMove(event);
      } else if (event.getPointerCount() == 2) {
        handleZoom(event);
      }
      break;
    case MotionEvent.ACTION_UP:
    case MotionEvent.ACTION_POINTER_UP:
      lastMoveX = -1;
      lastMoveY = -1;
      break;
    default:
      break;
  }
}
```

```
private void handleMove(MotionEvent event) {
  float moveX = event.getX();
  float moveY = event.getY();
  if (lastMoveX == -1 && lastMoveY == -1) {
    lastMoveX = moveX;
    lastMoveY = moveY;
  }
  moveDistX = (int) (moveX - lastMoveX);
  moveDistY = (int) (moveY - lastMoveY);
  if (moveDistX + totalTranslateX > 0 || moveDistX + totalTranslateX + curBitmapWidth < width) {
    moveDistX = 0;
  }
  if (moveDistY + totalTranslateY > 0 || moveDistY + totalTranslateY + curBitmapHeight < height) {
    moveDistY = 0;
  }
  status = STATUS_MOVE;
  invalidate();
  lastMoveX = moveX;
  lastMoveY = moveY;
}

private void handleZoom(MotionEvent event) {
  float fingerDist = calFingerDistance(event);
  calFingerCenter(event);
  if (fingerDist > lastFingerDist) {
    status = STATUS_ZOOM_OUT;
  } else {
    status = STATUS_ZOOM_IN;
  }
  scaledRatio = fingerDist * 1.0f / lastFingerDist;
  totalRatio = totalRatio * scaledRatio;
  if (totalRatio < initRatio) {
    totalRatio = initRatio;
  } else if (totalRatio > initRatio * 4) {
    totalRatio = initRatio * 4;
  }
```

```
  lastFingerDist = fingerDist;
  invalidate();
}
```

onDraw(Canvas canvas)

Draws the current state of the view.

```
@Override
protected void onDraw(Canvas canvas) {
  super.onDraw(canvas);
  if (scaleMode) {
    switch (status) {
      case STATUS_MOVE:
        move(canvas);
        break;
      case STATUS_ZOOM_IN:
      case STATUS_ZOOM_OUT:
        zoom(canvas);
        break;
      default:
        if (cacheBm != null) {
          canvas.drawBitmap(cacheBm, matrix, null);
          canvas.drawPath(path, paint);
        }
    }
  } else {
    if (cacheBm != null) {
      canvas.drawBitmap(cacheBm, matrix, null);
      canvas.drawPath(path, paint);
    }
  }
}
```

move(Canvas canvas)

Handles the move operation during scaling.

```
private void move(Canvas canvas) {
  matrix.reset();
  matrix.postScale(totalRatio, totalRatio);
```

```
  totalTranslateX = moveDistX + totalTranslateX;
  totalTranslateY = moveDistY + totalTranslateY;
  matrix.postTranslate(totalTranslateX, totalTranslateY);
  canvas.drawBitmap(cacheBm, matrix, null);
}
```

zoom(Canvas canvas)

Handles the zoom operation during scaling.

```
private void zoom(Canvas canvas) {
  matrix.reset();
  matrix.postScale(totalRatio, totalRatio);
  int scaledWidth = (int) (cacheBm.getWidth() * totalRatio);
  int scaledHeight = (int) (cacheBm.getHeight() * totalRatio);
  int translateX;
  int translateY;
  if (curBitmapWidth < width) {
    translateX = (width - scaledWidth) / 2;
  } else {
    translateX = (int) (centerPointX + (totalTranslateX - centerPointX) * scaledRatio);
    if (translateX > 0) {
      translateX = 0;
    } else if (scaledWidth + translateX < width) {
      translateX = width - scaledWidth;
    }
  }
  if (curBitmapHeight < height) {
    translateY = (height - scaledHeight) / 2;
  } else {
    translateY = (int) (centerPointY + (totalTranslateY - centerPointY) * scaledRatio);
    if (translateY > 0) {
      translateY = 0;
    } else if (scaledHeight + translateY < height) {
      translateY = height - scaledHeight;
    }
  }
  totalTranslateX = translateX;
```

```
  total

TranslateY = translateY;
  curBitmapWidth = scaledWidth;
  curBitmapHeight = scaledHeight;
  matrix.postTranslate(translateX, translateY);
  canvas.drawBitmap(cacheBm, matrix, null);
}
```

History Management

The `History` class manages the drawing history for undo and redo functionality.

saveToStack(Path path, Paint paint)

Saves the current path and paint to the stack.

```
public void saveToStack(Path path, Paint paint) {
  Draw draw = new Draw();
  draw.path = new Path(path);
  draw.paint = new Paint(paint);
  saveToStack(draw);
}

public void saveToStack(Draw draw) {
  curPos++;
  while (histroy.size() > curPos) {
    histroy.pop();
  }
  histroy.push(draw);
  if (callBack != null) {
    callBack.onHistoryChanged();
  }
}
```

getBitmapAtDraw(int n)

Returns a bitmap representing the state at a given point in the history.

```
public Bitmap getBitmapAtDraw(int n) {
```

```
  Canvas canvas = new Canvas();
  Bitmap bm = Utils.getCopyBitmap(srcBitmap);
  canvas.setBitmap(bm);
  for (int i = 0; i <= n; i++) {
    Draw draw = histroy.get(i);
    canvas.drawPath(draw.path, draw.paint);
  }
  return bm;
}
```

undo()

Performs the undo operation.

```
public Bitmap undo() throws UnsupportedOperationException {
  if (canUndo()) {
    curPos--;
    if (callBack != null) {
      callBack.onHistoryChanged();
    }
    return getBitmapAtDraw(curPos);
  } else {
    throw new UnsupportedOperationException("don't have undo record");
  }
}
```

redo()

Performs the redo operation.

```
public Bitmap redo() throws UnsupportedOperationException {
  if (canRedo()) {
    curPos++;
    if (callBack != null) {
      callBack.onHistoryChanged();
    }
    return getBitmapAtDraw(curPos);
  } else {
    throw new UnsupportedOperationException("don't have redo record");
  }
}
```

canUndo()

Checks if undo is possible.

```
public boolean canUndo() {
  return curPos > 0;
}
```

canRedo()

Checks if redo is possible.

```
public boolean canRedo() {
  return curPos + 1 < histroy.size();
}
```

Conclusion

`DrawActivity` and its related classes provide a comprehensive example of implementing a custom drawing view in Android. It demonstrates various techniques, including handling touch events, managing drawing history, and integrating with other components like fragments and async tasks. By understanding each component and algorithm, you can leverage these techniques in your own applications to create powerful and interactive drawing features.