**MASTER Students:**

**Ignea Vlad-Constantin, 31040701PM152697**

**Nistor Carmen, 31040701PM152699**

**„Alexandru Ioan Cuza” University of Iasi**

**Faculty of Economics and Business Administration**

**Master program: *Software Development and Business Information System***



**Scientific coordinator:**

Ph.D. Professor Octavian DOSPINESCU

**GRAPHICS TUTORIAL**

**TABLE OF CONTENTS**

[Introduction 3](#_Toc449544067)

[1. Graphics in Android. Brief overview 3](#_Toc449544068)

[1.1. Canvas and Drawables 3](#_Toc449544069)

[1.2. Hardware Acceleration 4](#_Toc449544070)

[1.3. OpenGL 4](#_Toc449544071)

[2. Concepts and classes used for graphics in Android 5](#_Toc449544072)

[2.1. View class 5](#_Toc449544073)

[2.2. Surface View 5](#_Toc449544074)

[2.3. Canvas class 6](#_Toc449544075)

[2.4. Image View 6](#_Toc449544076)

[2.5. Drawable 6](#_Toc449544077)

[2.6. GL Surface View 8](#_Toc449544078)

[3. Specific aspects regarding the draw of graphics in Android 8](#_Toc449544079)

[3.1. Drawing on Canvas 9](#_Toc449544080)

[3.2. Drawing on a View 10](#_Toc449544081)

[3.3. Drawing on a SurfaceView 10](#_Toc449544082)

[3.4. Drawing 2D and 3D graphics with the Open Graphics Library 11](#_Toc449544083)

[4. Practical examples & code sequences 12](#_Toc449544084)

[5. Screen captures 15](#_Toc449544085)

[6. Frequent troubles, errors, exceptions and ways to solve them 16](#_Toc449544086)

[7. Future directions 16](#_Toc449544087)

[References 17](#_Toc449544088)

# Introduction

Considering the users’ needs in Android for graphics tools, our article represents a brief overview of these instruments. The paper is divided in several sections, as it follows:

* The first section of the paper introduces specific aspects regarding 2D and 3D graphics in Android, including the most frequent concepts and classes used when referring to this subject;
* The second section focuses on practical examples presented in the form of code sequences, as well as the resulted obtained after running the code presented as screen captures;
* The third section describes the most frequent troubles, errors and exception, as well as the solution to these problems when using graphics in Android;
* The last part of the paper is dedicated to future research directions upon users’ graphical demands in Android and possibilities to solve them and the references accessed for writing the paper.

The first step in order to draw custom 2D and 3D graphics in Android is to know exactly the graphical demands of the application (app). More exactly, the developer has to have a deep knowledge of the user’s graphical interests in the app in order to provide the most suitable graphics. In this regard, the most eloquent example is using graphics for a static application and not an interactive one. In this situation, it’s obvious that the graphical requirements of the static application will differ from the interactive one and, as a result, the developer will use different tools to implement them.

After reviewing the main sections of the paper, we are going to focus further on the first part – Specific aspects, concepts and classes for graphics in Android.

# 1. Graphics in Android. Brief overview

Android offers many features which cover multiple areas of interest as application development, internet, media and connectivity. Among them, an important role is played by 2D and 3D capabilities for drawing custom graphics. As graphical demands differ from user to user, Android has several tool available for developers which can be divided in:

## 1.1. Canvas and Drawables

Android provides a set of View widgets that provide general functionality for a wide array of user interfaces. The developers can extend these widgets in order to modify the manner in which they look and behave. Moreover, the developers have the possibility to perform their own 2D rendering using the existent drawing methods from the Canvas class or they can create Drawable objects for things such as textured buttons or frame-by-frame animations[[1]](#footnote-1).

## 1.2. Hardware Acceleration

Starting with the version Android 3.0 most of the drawing realised using Canvas APIs can be hardware accelerated for a future increased performance. From a practical perspective, this means that all drawing operations will be done using the GPU (Graphics Processing Unit) on a View’s canvas. In terms of hardware costs, it is obvious that an increased used of resource will reflect on the use of RAM, the application requiring more RAM in order to function at the highest levels.

Depending of the Target API level the hardware acceleration can be enabled explicitly, if the Target API level is <=14, or by default, if the Target API level is >=14[[2]](#footnote-2).

Usually, if the applications is based only on standard views and Drawables, turning in the hardware acceleration globally it should not have adverse drawing effects. Nevertheless, as hardware acceleration is not available for all of the 2D drawing operations in Android, turning on this feature can have a negative impact on some of the custom views or drawing calls. In section 6 – *Frequent troubles, errors and exceptions* we are going to provide further details upon this matter.

## 1.3. OpenGL

Android supports OpenGL ES 1.0 and 2.0, with Android framework APIs as well as natively with the Native Development Kit (NDK). Using the framework APIs is best suited when adding a few graphical enhancements to the application which are not supported with the Canvas APIs, or if a platform independence is desired but not high performance as well.

There is a performance hit in using the framework APIs compared to the NDK, so for many graphic intensive applications such as games, using the NDK is beneficial. OpenGL with the NDK is also useful if there is a lot of native code that has to be ported over to Android.

OpenGL is a cross-platform graphics API that specifies a standard software interface for 3D graphics processing hardware. OpenGL ES is a flavor of the OpenGL specification intended for embedded devices. Android supports several versions of the OpenGL ES API as it follows:[[3]](#footnote-3)

* OpenGL ES 1.0 and 1.1 - This API specification is supported by Android 1.0 and higher.
* OpenGL ES 2.0 - This API specification is supported by Android 2.2 (API level 8) and higher.
* OpenGL ES 3.0 - This API specification is supported by Android 4.3 (API level 18) and higher.
* OpenGL ES 3.1 - This API specification is supported by Android 5.0 (API level 21) and higher.

After the brief presentation of options available for drawing graphics in Android we are going to discuss further the specific concepts and classes used.

# 2. Concepts and classes used for graphics in Android

When drawing graphics in Android there are several classes that can be used, the difference being made by the type of draw chosen by the developers. In this section we will focus on the most frequent classes encountered. In this regard, we consider the main options available for creating graphics, namely:

* Draw with a Canvas;
* Draw on a View;
* Draw on a SurfaceView;
* Drawables.

In reference with the type of method chosen for drawing the graphics the developer can use the following classes:

## 2.1. View class

The view class is the basic building block for user interface components. A View occupies a rectangular area on the screen and is responsible for drawing and event handling. View is the base class for widgets, which are used to create interactive UI components (buttons, text fields, etc.) [[4]](#footnote-4).

In order to create a custom view, the developer has to provide overrides for some of the standard methods that the framework calls on all views. Usually, not all of the methods have to be overridden, just overriding onDraw(android.graphics.Canvas) method being enough.

## 2.2. Surface View

The SurfaceView class provides a dedicated drawing surface embedded inside of a view hierarchy. Using this class the developer can control the format of this surface and, if wanted, its size. Usually, SurfaceView is used when placing the surface at the correct location on the screen[[5]](#footnote-5).

## 2.3. Canvas class

The Canvas class holds the "draw" calls. To draw something, 4 basic components are needed: A Bitmap to hold the pixels, a Canvas to host the draw calls (writing into the bitmap), a drawing primitive (e.g. Rect, Path, text, Bitmap), and a paint (to describe the colours and styles for the drawing)[[6]](#footnote-6).

## 2.4. Image View

The image view class is used when displaying an arbitrary image, such as an icon. The class can load images from various sources (such as resources or content providers), takes care of computing its measurement from the image so that it can be used in any layout manager, and provides various display options such as scaling and tinting[[7]](#footnote-7).

## 2.5. Drawable

A Drawable is a general abstraction for "something that can be drawn." Most often you will deal with Drawable as the type of resource retrieved for drawing things to the screen; the Drawable class provides a generic API for dealing with an underlying visual resource that may take a variety of forms. Unlike a View, a Drawable does not have any facility to receive events or otherwise interact with the user.

Though usually not visible to the application, Drawables may take a variety of forms:

* **Bitmap**: the simplest Drawable, a PNG or JPEG image.
* **Nine Patch**: an extension to the PNG format allows it to specify information about how to stretch it and place things inside of it.
* **Shape**: contains simple drawing commands instead of a raw bitmap, allowing it to resize better in some cases.
* **Layers**: a compound drawable, which draws multiple underlying drawables on top of each other.
* **States**: a compound drawable that selects one of a set of drawables based on its state.
* **Levels**: a compound drawable that selects one of a set of drawables based on its level.
* **Scale**: a compound drawable with a single child drawable, whose overall size is modified based on the current level.

In Android, the Drawable provides classes to manage a variety of visual elements that are intended for display only, such as bitmaps and gradients. These elements are often used by widgets as background images or simply as indicators (for example, a volume level indicator).

**Table 1. Classes from the Drawable package in Android**

| Class | Description |
| --- | --- |
| * 1. *AnimatedStateListDrawable* | * Drawable containing a set of Drawable keyframes where the currently displayed keyframe is chosen based on the current state set. |
| * 1. *Animatable2* | * Abstract class that drawables supporting animations and callbacks should extend. |
| * 1. *Drawable.Callback* | * Implement this interface if you want to create an animated drawable that extends [Drawable](http://developer.android.com/reference/android/graphics/drawable/Drawable.html). |
| * 1. *Icon.OnDrawableLoadedListener* | * Implement this interface to receive a callback when [loadDrawableAsync](http://developer.android.com/reference/android/graphics/drawable/Icon.html#loadDrawableAsync(android.content.Context, android.graphics.drawable.Icon.OnDrawableLoadedListener, android.os.Handler)) is finished and your Drawable is ready. |
| * 1. *Animatable2.AnimationCallback* |  |
| * 1. *AnimatedStateListDrawable* | * Drawable containing a set of Drawable keyframes where the currently displayed keyframe is chosen based on the current state set. |
| * 1. *AnimatedVectorDrawable* | * This class uses [ObjectAnimator](http://developer.android.com/reference/android/animation/ObjectAnimator.html) and [AnimatorSet](http://developer.android.com/reference/android/animation/AnimatorSet.html) to animate the properties of a [VectorDrawable](http://developer.android.com/reference/android/graphics/drawable/VectorDrawable.html) to create an animated drawable. |
| * 1. *AnimationDrawable* | * An object used to create frame-by-frame animations, defined by a series of Drawable objects, which can be used as a View object's background. |
| * 1. *BitmapDrawable* | * A Drawable that wraps a bitmap and can be tiled, stretched, or aligned. |
| * 1. *ClipDrawable* | * A Drawable that clips another Drawable based on this Drawable's current level value. |
| * 1. *ColorDrawable* | * A specialized Drawable that fills the Canvas with a specified color. |
| * 1. *Drawable* | * A Drawable is a general abstraction for "something that can be drawn." Most often you will deal with Drawable as the type of resource retrieved for drawing things to the screen; the Drawable class provides a generic API for dealing with an underlying visual resource that may take a variety of forms. |
| * 1. *Drawable.ConstantState* | * This abstract class is used by [Drawable](http://developer.android.com/reference/android/graphics/drawable/Drawable.html)s to store shared constant state and data between Drawables. |
| * 1. *DrawableContainer* | * A helper class that contains several [Drawable](http://developer.android.com/reference/android/graphics/drawable/Drawable.html)s and selects which one to use. |
| * 1. *DrawableContainer.DrawableContainerState* | * A ConstantState that can contain several [Drawable](http://developer.android.com/reference/android/graphics/drawable/Drawable.html)s. |
| * 1. *DrawableWrapper* | * Drawable container with only one child element. |
| * 1. *GradientDrawable* | * A Drawable with a color gradient for buttons, backgrounds, etc. |
| * 1. *Icon* | * An umbrella container for several serializable graphics representations, including Bitmaps, compressed bitmap images (e.g. |
| * 1. *InsetDrawable* | * A Drawable that insets another Drawable by a specified distance. |
| * 1. *LayerDrawable* | * A Drawable that manages an array of other Drawables. |
| * 1. *LevelListDrawable* | * A resource that manages a number of alternate Drawables, each assigned a maximum numerical value. |
| * 1. *NinePatchDrawable* | * A resizeable bitmap, with stretchable areas that you define. |
| * 1. *PaintDrawable* | * Drawable that draws its bounds in the given paint, with optional rounded corners. |
| * 1. *PictureDrawable* | * Drawable subclass that wraps a Picture, allowing the picture to be used wherever a Drawable is supported. |
| * 1. *RippleDrawable* | * Drawable that shows a ripple effect in response to state changes. |
| * 1. *RotateDrawable* | * A Drawable that can rotate another Drawable based on the current level value. |
| * 1. *ScaleDrawable* | * A Drawable that changes the size of another Drawable based on its current level value. |
| * 1. *ShapeDrawable* | * A Drawable object that draws primitive shapes. |
| * 1. *ShapeDrawable.ShaderFactory* | * Base class defines a factory object that is called each time the drawable is resized (has a new width or height). |
| * 1. *StateListDrawable* | * Lets you assign a number of graphic images to a single Drawable and swap out the visible item by a string ID value. |
| * 1. *TransitionDrawable* | * An extension of LayerDrawables that is intended to cross-fade between the first and second layer. |
| * 1. *VectorDrawable* | * This lets you create a drawable based on an XML vector graphic. |

Source: http://developer.android.com/reference/android/graphics/drawable/package-summary.html, accessed 24.04.2016, 13.24.

## 2.6. GL Surface View

A GLSurfaceView is a class in the android framework that allows the creation and manipulation of graphics with the OpenGL ES API. Basically, it represents an implementation of SurfaceView that uses the dedicated surface for displaying OpenGL rendering. A GLSurfaceView provides the following features:[[8]](#footnote-8)

* Manages a surface, which is a special piece of memory that can be composited into the Android view system.
* Manages an EGL display, which enables OpenGL to render into a surface.
* Accepts a user-provided Renderer object that does the actual rendering.
* Renders on a dedicated thread to decouple rendering performance from the UI thread.
* Supports both on-demand and continuous rendering.
* Optionally wraps, traces, and/or error-checks the renderer's OpenGL calls.

As it can be noticed from the brief review of the classes above, Android offers multiple option for creating graphics consistent with the user needs in the application. The next section offers some explanations regarding the use of several of these classes in Android in dependence with the manner chosen by the developer.

# 3. Specific aspects regarding the draw of graphics in Android

For drawing 2D graphics, the Android framework APIS offers a set of 2D drawing APIs that allow the developers to set their own custom graphics on canvas or to change the existing Views in order to customize their look and feel according to users preferences.

In Android, there are two main modes available for drawing 2D graphics, namely:[[9]](#footnote-9)

1. Draw your graphics or animations into a View object from your layout. In this manner, the drawing of your graphics is handled by the system's normal View hierarchy drawing process — you simply define the graphics to go inside the View.
2. Draw your graphics directly to a Canvas. This way, you personally call the appropriate class's [onDraw()](http://developer.android.com/reference/android/view/View.html#onDraw(android.graphics.Canvas)) method (passing it your Canvas), or one of the Canvas draw...() methods (like[drawPicture()](http://developer.android.com/reference/android/graphics/Canvas.html#drawPicture(android.graphics.Picture, android.graphics.Rect))). In doing so, you are also in control of any animation.

Option "a," drawing to a View, is your best choice when you want to draw simple graphics that do not need to change dynamically and are not part of a performance-intensive game. For example, you should draw your graphics into a View when you want to display a static graphic or predefined animation, within an otherwise static application. Read [Drawables](http://developer.android.com/guide/topics/graphics/2d-graphics.html#drawables)for more information.

Option "b," drawing to a Canvas, is better when your application needs to regularly re-draw itself. Applications such as video games should be drawing to the Canvas on its own. However, there's more than one way to do this:

* In the same thread as your UI Activity, wherein you create a custom View component in your layout, call [invalidate()](http://developer.android.com/reference/android/view/View.html#invalidate()) and then handle the [onDraw()](http://developer.android.com/reference/android/view/View.html#onDraw(android.graphics.Canvas)) callback.
* Or, in a separate thread, wherein you manage a [SurfaceView](http://developer.android.com/reference/android/view/SurfaceView.html) and perform draws to the Canvas as fast as your thread is capable (you do not need to request invalidate()).

## 3.1. Drawing on Canvas

When you're writing an application in which you would like to perform specialized drawing and/or control the animation of graphics, you should do so by drawing through a Canvas. A Canvas works for you as a pretense, or interface, to the actual surface upon which your graphics will be drawn — it holds all of your "draw" calls. Via the Canvas, your drawing is actually performed upon an underlying Bitmap, which is placed into the window.

In the event that you're drawing within the onDraw() callback method, the Canvas is provided for you and you need only place your drawing calls upon it. You can also acquire a Canvas from SurfaceHolder.lockCanvas(), when dealing with a SurfaceView object. (Both of these scenarios are discussed in the following sections.) However, if you need to create a new Canvas, then you must define the Bitmap upon which drawing will actually be performed. The Bitmap is always required for a Canvas. You can set up a new Canvas like this:

Now your Canvas will draw onto the defined Bitmap. After drawing upon it with the Canvas, you can then carry your Bitmap to another Canvas with one of the Canvas.drawBitmap(Bitmap,...) methods. It's recommended that you ultimately draw your final graphics through a Canvas offered to you by View.onDraw() or SurfaceHolder.lockCanvas() (see the following sections).

The Canvas class has its own set of drawing methods that you can use, like drawBitmap(...), drawRect(...), drawText(...), and many more. Other classes that you might use also have draw() methods. For example, you'll probably have some Drawable objects that you want to put on the Canvas. Drawable has its own draw() method that takes your Canvas as an argument.

## 3.2. Drawing on a View

If your application does not require a significant amount of processing or frame-rate speed (perhaps for a chess game, a snake game, or another slowly-animated application), then you should consider creating a custom View component and drawing with a Canvas in View.onDraw(). The most convenient aspect of doing so is that the Android framework will provide you with a pre-defined Canvas to which you will place your drawing calls[[10]](#footnote-10).

To start, extend the View class (or descendant thereof) and define the onDraw() callback method. This method will be called by the Android framework to request that your View draw itself. This is where you will perform all your calls to draw through the Canvas, which is passed to you through the onDraw() callback.

The Android framework will only call onDraw() as necessary. Each time that your application is prepared to be drawn, you must request your View be invalidated by calling invalidate(). This indicates that you'd like your View to be drawn and Android will then call your onDraw() method (though is not guaranteed that the callback will be instantaneous).

Inside your View component's onDraw(), use the Canvas given to you for all your drawing, using various Canvas.draw...() methods, or other class draw() methods that take your Canvas as an argument. Once your onDraw() is complete, the Android framework will use your Canvas to draw a Bitmap handled by the system.

## 3.3. Drawing on a SurfaceView

The SurfaceView is a special subclass of View that offers a dedicated drawing surface within the View hierarchy. The aim is to offer this drawing surface to an application's secondary thread, so that the application isn't required to wait until the system's View hierarchy is ready to draw. Instead, a secondary thread that has reference to a SurfaceView can draw to its own Canvas at its own pace[[11]](#footnote-11).

To begin, you need to create a new class that extends SurfaceView. The class should also implement SurfaceHolder.Callback. This subclass is an interface that will notify you with information about the underlying Surface, such as when it is created, changed, or destroyed. These events are important so that you know when you can start drawing, whether you need to make adjustments based on new surface properties, and when to stop drawing and potentially kill some tasks. Inside your SurfaceView class is also a good place to define your secondary Thread class, which will perform all the drawing procedures to your Canvas.

Instead of handling the Surface object directly, you should handle it via a SurfaceHolder. So, when your SurfaceView is initialized, get the SurfaceHolder by calling getHolder(). You should then notify the SurfaceHolder that you'd like to receive SurfaceHolder callbacks (from SurfaceHolder.Callback) by calling addCallback() (pass it this). Then override each of the SurfaceHolder.Callback methods inside your SurfaceView class.

In order to draw to the Surface Canvas from within your second thread, you must pass the thread your SurfaceHandler and retrieve the Canvas with lockCanvas(). You can now take the Canvas given to you by the SurfaceHolder and do your necessary drawing upon it. Once you're done drawing with the Canvas, call unlockCanvasAndPost(), passing it your Canvas object. The Surface will now draw the Canvas as you left it. Perform this sequence of locking and unlocking the canvas each time you want to redraw.

## 3.4. Drawing 2D and 3D graphics with the Open Graphics Library

Android supports OpenGL both through its framework API and the Native Development Kit (NDK).

In order to draw graphics with OpenGL ES in your Android application, you must create a view container for them. One of the more straight-forward ways to do this is to implement both a GLSurfaceView and a GLSurfaceView.Renderer. A GLSurfaceView is a view container for graphics drawn with OpenGL and GLSurfaceView.Renderer controls what is drawn within that view. For more information about these classes, see the OpenGL ES developer guide.

GLSurfaceView is just one way to incorporate OpenGL ES graphics into your application. For a full-screen or near-full screen graphics view, it is a reasonable choice. Developers who want to incorporate OpenGL ES graphics in a small portion of their layouts should take a look at TextureView. For real, do-it-yourself developers, it is also possible to build up an OpenGL ES view using SurfaceView, but this requires writing quite a bit of additional code[[12]](#footnote-12).

# 4. Practical examples & code sequences

Two-dimensional graphics in Android are implemented differently than dynamic, interactive, or three-dimensional graphics. Android has 2D drawing APIs for creating this kind of graphics. Using these APIs, there are two ways of implementing graphics:

* Drawing to a View
* Drawing on a Canvas

Further we will focus on displaying and image and text onto Canvas[[13]](#footnote-13).

1. Create a new Project with the activity named MySurfaceViewActivity. This project displays an image and text on the Canvas. The example shows how an image can be moved dynamically in the canvas. The sample code listens for the touch events and renders an image on the touched coordinates of the screen

package com.surface.view;

import android.app.Activity;

import android.content.Context;

import android.graphics.Bitmap;

import android.graphics.BitmapFactory;

import android.graphics.Canvas;

import android.graphics.Color;

import android.graphics.Paint;

import android.os.Bundle;

import android.view.SurfaceHolder;

import android.view.SurfaceView;

import android.view.Window;

public class MySurfaceViewActivity extends Activity {

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

requestWindowFeature(Window.FEATURE\_NO\_TITLE);

setContentView(new MySurface(this));

}

}

2. Next, create an inner class that extends SurfaceView and implements SurfaceHolder.callback and implement the three methods as shown below:

class MySurface extends SurfaceView implements SurfaceHolder.Callback {

@Override

public void surfaceChanged(SurfaceHolder holder, int format, int width, int height) {}

@Override

public void surfaceCreated(SurfaceHolder holder) {}

@Override

public void surfaceDestroyed(SurfaceHolder holder) {}

3. Next, implement the callback methods and call the application’s secondary thread inside a constructor.

class MySurface extends SurfaceView implements SurfaceHolder.Callback {

private SecondThread thread;

//Initial position of the image

private int x = 100;

private int y = 200;

public MySurface(Context context) {

super(context);

getHolder().addCallback(this);

thread = new SecondThread(getHolder(), this);

}

@Override

public void surfaceChanged(SurfaceHolder holder, int format, int width, int height) {

}

@Override

public void surfaceCreated(SurfaceHolder holder) {

thread.setRunning(true);

thread.start();

}

@Override

public void surfaceDestroyed(SurfaceHolder holder) {

boolean retry = true;

thread.setRunning(false);

while (retry) {

try {

thread.join();

retry = false;

} catch (InterruptedException e) {}

}

}

}

4. Create the secondary thread in your application where the constructor requires parameters like mySurface and the SurfaceHolder.

lass SecondThread extends Thread {

private SurfaceHolder surfaceHolder;

private MySurface mySurface;

private boolean \_run = false;

public SecondThread(SurfaceHolder surfaceHolder, MySurface mySurface) {

this.surfaceHolder = surfaceHolder;

this.mySurface = mySurface;

}

public void setRunning(boolean run) {

\_run = run;

}

@Override

public void run() {

Canvas c;

while (\_run) {

c = null;

try {

c = surfaceHolder.lockCanvas(null);

synchronized (surfaceHolder) {

mySurface.onDraw(c);

}

} finally {

if (c != null) {

surfaceHolder.unlockCanvasAndPost(c);

}

}

}

}

}

5. Now draw the graphics using the Canvas object:

public void onDraw(Canvas canvas) {

Paint paint = new Paint();

paint.setColor(Color.WHITE);

paint.setTextSize(20);

paint.setAntiAlias(true);

canvas.drawColor(Color.BLUE);

canvas.drawText("Hello Android", 10, 20, paint);

Bitmap image = BitmapFactory.decodeResource(getResources(), R.drawable.ic\_launcher);

//renders image using x and y parameter x and y value is filled by the touch //event

canvas.drawBitmap(image, x, y, null);

}

6. The code renders the image using x, y parameters. The x, y parameters values are filled using touch events listener as shown in the following code:

@Override

public boolean onTouchEvent(MotionEvent event) {

if (event.getAction() == MotionEvent.ACTION\_MOVE) {

x = (int) event.getX();

y = (int) event.getY();

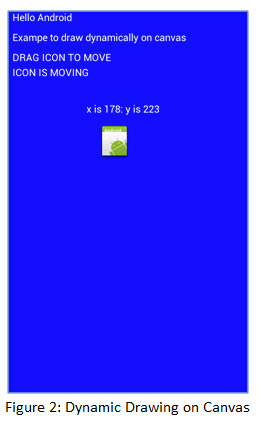
}

return true;

}

# 5. Screen captures

The image and text displayed onto Canvas will be the one in Figure 1.



**Figure 1. Dynamic Drawing on Canvas**

The image is rendered by the code using x, y parameters.

# 6. Frequent troubles, errors, exceptions and ways to solve them

If an application uses only standard views and Drawables, turning it on globally should not cause any adverse drawing effects. However, because hardware acceleration is not supported for all of the 2D drawing operations, turning it on might affect some of your custom views or drawing calls. Problems usually manifest themselves as invisible elements, exceptions, or wrongly rendered pixels. To remedy this, Android gives the option to enable or disable hardware acceleration at multiple levels.

If an application performs custom drawing, testing the application on actual hardware devices with hardware acceleration turned on could help in finding any problems.

# 7. Future directions

Vulkan is a low-overhead, cross platform 3D graphics and compute API first announced at GDC 2015 by the Khronos Group and officially released (the stable version 1.0.11) on 22 April 2016. The Vulkan API was initially referred to as the "next generation OpenGL initiative" by Khronos, but use of those names were discontinued once the Vulkan name was announced.

Like OpenGL, Vulkan targets high-performance realtime 3D graphics applications such as games and interactive media across all platforms, and offers higher performance and lower CPU usage, much like Direct3D 12 and Mantle[[14]](#footnote-14).

Vulkan is intended to provide a variety of advantages over other APIs as well as its spiritual predecessor, OpenGL. Vulkan offers lower overhead, more direct control over the GPU, and lower CPU usage. Intended advantages include:

* Reduced driver overhead, reducing CPU workloads.[19]
* Reduced load on CPUs through the use of batching, leaving the CPU free to do additional computation or rendering than otherwise.[20]
* Intelligent and even CPU scaling for multi-core CPUs, which are by far the majority type of CPU on the market.

The Android platform includes an Android-specific implementation of the Vulkan API specification from the Khronos Group. Vulkan is a low-overhead, cross-platform API for high-performance, 3D graphics. It provides tools for creating high-quality, real-time graphics in applications. Vulkan also provides advantages such as reducing CPU overhead and providing support for the SPIR-V Binary Intermediate language.

# References

* 1. Gavalas, D., & Economou, D. (2011). Development platforms for mobile applications: Status and trends. *Software, IEEE*, *28*(1), 77-86.
  2. http://developer.android.com/guide/topics/graphics/overview.html, accessed 23.04.2016, 20.56.
  3. http://developer.android.com/guide/topics/graphics/hardware-accel.html, accessed 23.04.2016, 21.34.
  4. http://developer.android.com/guide/topics/graphics/opengl.html, accessed 23.04.2016, 22.07.
  5. http://developer.android.com/reference/android/view/View.html, accessed 24.04.2016, 10.14.
  6. http://developer.android.com/reference/android/view/SurfaceView.html, accessed 24.04.2016, 10.32.
  7. http://developer.android.com/reference/android/graphics/Canvas.html, accessed 24.04.2016, 10.49.
  8. http://developer.android.com/reference/android/widget/ImageView.html, accessed 24.04.2016, 11.15.
  9. http://developer.android.com/reference/android/graphics/drawable/package-summary.html, accessed 24.04.2016, 13.24.
  10. http://developer.android.com/guide/topics/graphics/opengl.html, accessed 25.04.2016, 11.15.
  11. http://developer.android.com/guide/topics/graphics/2d-graphics.html, accessed 25.04.2016, 12.02.
  12. http://developer.android.com/guide/topics/graphics/2d-graphics.html, accessed 25.04.2016, 12.14.
  13. http://developer.android.com/guide/topics/graphics/2d-graphics.html, accessed 25.04.2016, 12.37.
  14. http://developer.android.com/training/graphics/opengl/environment.html, accessed 25.04.2016, 15.21.
  15. http://developer.samsung.com/technical-doc/view.do;jsessionid=rpVXutfwvY0Ju-ec61jeirHJJ8IQvKCwyScEoHC0tPSA37KAAZOY!-491879268?v=T000000096, accessed 25.04.2016, 17.43.
  16. https://en.wikipedia.org/wiki/Vulkan\_(API), accessed 25.04.2016, 17.23.

1. http://developer.android.com/guide/topics/graphics/overview.html, accessed 23.04.2016, 20.56. [↑](#footnote-ref-1)
2. http://developer.android.com/guide/topics/graphics/hardware-accel.html, accessed 23.04.2016, 21.34. [↑](#footnote-ref-2)
3. http://developer.android.com/guide/topics/graphics/opengl.html, accessed 23.04.2016, 22.07. [↑](#footnote-ref-3)
4. http://developer.android.com/reference/android/view/View.html, accessed 24.04.2016, 10.14. [↑](#footnote-ref-4)
5. http://developer.android.com/reference/android/view/SurfaceView.html, accessed 24.04.2016, 10.32. [↑](#footnote-ref-5)
6. http://developer.android.com/reference/android/graphics/Canvas.html, accessed 24.04.2016, 10.49. [↑](#footnote-ref-6)
7. http://developer.android.com/reference/android/widget/ImageView.html, accessed 24.04.2016, 11.15. [↑](#footnote-ref-7)
8. http://developer.android.com/guide/topics/graphics/opengl.html, accessed 25.04.2016, 11.15. [↑](#footnote-ref-8)
9. http://developer.android.com/guide/topics/graphics/2d-graphics.html, accessed 25.04.2016, 12.02. [↑](#footnote-ref-9)
10. http://developer.android.com/guide/topics/graphics/2d-graphics.html, accessed 25.04.2016, 12.14. [↑](#footnote-ref-10)
11. http://developer.android.com/guide/topics/graphics/2d-graphics.html, accessed 25.04.2016, 12.37. [↑](#footnote-ref-11)
12. http://developer.android.com/training/graphics/opengl/environment.html, accessed 25.04.2016, 15.21. [↑](#footnote-ref-12)
13. http://developer.samsung.com/technical-doc/

    view.do;jsessionid=rpVXutfwvY0Ju-ec61jeirHJJ8IQvKCwyScEoHC0tPSA37KAAZOY!-491879268?v=T000000096, accessed 25.04.2016, 17.43. [↑](#footnote-ref-13)
14. https://en.wikipedia.org/wiki/Vulkan\_(API), accessed 25.04.2016, 17.23. [↑](#footnote-ref-14)