```java
/*
 * Copyright (C) 2006 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package android.graphics;

import android.annotation.ColorInt;
import android.annotation.IntDef;
import android.annotation.NonNull;
import android.annotation.Nullable;
import android.annotation.Size;
import android.os.Build;

import dalvik.annotation.optimization.CriticalNative;
import dalvik.annotation.optimization.FastNative;

import libcore.util.NativeAllocationRegistry;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

import javax.microedition.khronos.opengles.GL;

/**
 * The Canvas class holds the "draw" calls. To draw something, you need
 * 4 basic components: 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 colors and styles for the
 * drawing).
 *
 * <div class="special reference">
 * <h3>Developer Guides</h3>
 * <p>For more information about how to use Canvas, read the
 * <a href="{@docRoot}guide/topics/graphics/2d-graphics.html">
 * Canvas and Drawables</a> developer guide.</p></div>
 */
public class Canvas extends BaseCanvas {
    /** @hide */
    public static boolean sCompatibilityRestore = false;
    /** @hide */
    public static boolean sCompatibilitySetBitmap = false;

    /** @hide */
    public long getNativeCanvasWrapper() {
        return mNativeCanvasWrapper;
    }

    /** @hide */
    public boolean isRecordingFor(Object o) { return false; }

    // may be null
    private Bitmap mBitmap;

    // optional field set by the caller
    private DrawFilter mDrawFilter;
```

```
// Maximum bitmap size as defined in Skia's native code
// (see SkCanvas.cpp, SkDraw.cpp)
private static final int MAXMIMUM_BITMAP_SIZE = 32766;

// The approximate size of the native allocation associated with
// a Canvas object.
private static final long NATIVE_ALLOCATION_SIZE = 525;

// Use a Holder to allow static initialization of Canvas in the boot image.
private static class NoImagePreloadHolder {
    public static final NativeAllocationRegistry sRegistry = new NativeAllocationRegistry(
            Canvas.class.getClassLoader(), nGetNativeFinalizer(), NATIVE_ALLOCATION_SIZE);
}

// This field is used to finalize the native Canvas properly
private Runnable mFinalizer;

/**
 * Construct an empty raster canvas. Use setBitmap() to specify a bitmap to
 * draw into.  The initial target density is {@link Bitmap#DENSITY_NONE};
 * this will typically be replaced when a target bitmap is set for the
 * canvas.
 */
public Canvas() {
    if (!isHardwareAccelerated()) {
        // 0 means no native bitmap
        mNativeCanvasWrapper = nInitRaster(null);
        mFinalizer = NoImagePreloadHolder.sRegistry.registerNativeAllocation(
                this, mNativeCanvasWrapper);
    } else {
        mFinalizer = null;
    }
}

/**
 * Construct a canvas with the specified bitmap to draw into. The bitmap
 * must be mutable.
 *
 * <p>The initial target density of the canvas is the same as the given
 * bitmap's density.
 *
 * @param bitmap Specifies a mutable bitmap for the canvas to draw into.
 */
public Canvas(@NonNull Bitmap bitmap) {
    if (!bitmap.isMutable()) {
        throw new IllegalStateException("Immutable bitmap passed to Canvas constructor");
    }
    throwIfCannotDraw(bitmap);
    mNativeCanvasWrapper = nInitRaster(bitmap);
    mFinalizer = NoImagePreloadHolder.sRegistry.registerNativeAllocation(
            this, mNativeCanvasWrapper);
    mBitmap = bitmap;
    mDensity = bitmap.mDensity;
}

/** @hide */
public Canvas(long nativeCanvas) {
    if (nativeCanvas == 0) {
        throw new IllegalStateException();
    }
    mNativeCanvasWrapper = nativeCanvas;
    mFinalizer = NoImagePreloadHolder.sRegistry.registerNativeAllocation(
            this, mNativeCanvasWrapper);
    mDensity = Bitmap.getDefaultDensity();
}

/**
 * Returns null.
 *
```

```
     * @deprecated This method is not supported and should not be invoked.
     *
     * @hide
     */
    @Deprecated
    protected GL getGL() {
        return null;
    }

    /**
     * Indicates whether this Canvas uses hardware acceleration.
     *
     * Note that this method does not define what type of hardware acceleration
     * may or may not be used.
     *
     * @return True if drawing operations are hardware accelerated,
     *         false otherwise.
     */
    public boolean isHardwareAccelerated() {
        return false;
    }

    /**
     * Specify a bitmap for the canvas to draw into. All canvas state such as
     * layers, filters, and the save/restore stack are reset. Additionally,
     * the canvas' target density is updated to match that of the bitmap.
     *
     * Prior to API level {@value Build.VERSION_CODES#O} the current matrix and
     * clip stack were preserved.
     *
     * @param bitmap Specifies a mutable bitmap for the canvas to draw into.
     * @see #setDensity(int)
     * @see #getDensity()
     */
    public void setBitmap(@Nullable Bitmap bitmap) {
        if (isHardwareAccelerated()) {
            throw new RuntimeException("Can't set a bitmap device on a HW accelerated canvas");
        }

        Matrix preservedMatrix = null;
        if (bitmap != null && sCompatibilitySetBitmap) {
            preservedMatrix = getMatrix();
        }

        if (bitmap == null) {
            nSetBitmap(mNativeCanvasWrapper, null);
            mDensity = Bitmap.DENSITY_NONE;
        } else {
            if (!bitmap.isMutable()) {
                throw new IllegalStateException();
            }
            throwIfCannotDraw(bitmap);

            nSetBitmap(mNativeCanvasWrapper, bitmap);
            mDensity = bitmap.mDensity;
        }

        if (preservedMatrix != null) {
            setMatrix(preservedMatrix);
        }

        mBitmap = bitmap;
    }

    /** @hide */
    public void setHighContrastText(boolean highContrastText) {
        nSetHighContrastText(mNativeCanvasWrapper, highContrastText);
    }

    /** @hide */
```

```
public void insertReorderBarrier() {}

/** @hide */
public void insertInorderBarrier() {}

/**
 * Return true if the device that the current layer draws into is opaque
 * (i.e. does not support per-pixel alpha).
 *
 * @return true if the device that the current layer draws into is opaque
 */
public boolean isOpaque() {
    return nIsOpaque(mNativeCanvasWrapper);
}

/**
 * Returns the width of the current drawing layer
 *
 * @return the width of the current drawing layer
 */
public int getWidth() {
    return nGetWidth(mNativeCanvasWrapper);
}

/**
 * Returns the height of the current drawing layer
 *
 * @return the height of the current drawing layer
 */
public int getHeight() {
    return nGetHeight(mNativeCanvasWrapper);
}

/**
 * <p>Returns the target density of the canvas.  The default density is
 * derived from the density of its backing bitmap, or
 * {@link Bitmap#DENSITY_NONE} if there is not one.</p>
 *
 * @return Returns the current target density of the canvas, which is used
 * to determine the scaling factor when drawing a bitmap into it.
 *
 * @see #setDensity(int)
 * @see Bitmap#getDensity()
 */
public int getDensity() {
    return mDensity;
}

/**
 * <p>Specifies the density for this Canvas' backing bitmap.  This modifies
 * the target density of the canvas itself, as well as the density of its
 * backing bitmap via {@link Bitmap#setDensity(int) Bitmap.setDensity(int)}.
 *
 * @param density The new target density of the canvas, which is used
 * to determine the scaling factor when drawing a bitmap into it.  Use
 * {@link Bitmap#DENSITY_NONE} to disable bitmap scaling.
 *
 * @see #getDensity()
 * @see Bitmap#setDensity(int)
 */
public void setDensity(int density) {
    if (mBitmap != null) {
        mBitmap.setDensity(density);
    }
    mDensity = density;
}

/** @hide */
public void setScreenDensity(int density) {
    mScreenDensity = density;
```

```
}

/**
 * Returns the maximum allowed width for bitmaps drawn with this canvas.
 * Attempting to draw with a bitmap wider than this value will result
 * in an error.
 *
 * @see #getMaximumBitmapHeight()
 */
public int getMaximumBitmapWidth() {
    return MAXMIMUM_BITMAP_SIZE;
}

/**
 * Returns the maximum allowed height for bitmaps drawn with this canvas.
 * Attempting to draw with a bitmap taller than this value will result
 * in an error.
 *
 * @see #getMaximumBitmapWidth()
 */
public int getMaximumBitmapHeight() {
    return MAXMIMUM_BITMAP_SIZE;
}

// the SAVE_FLAG constants must match their native equivalents

/** @hide */
@IntDef(flag = true,
        value = {
            ALL_SAVE_FLAG
        })
@Retention(RetentionPolicy.SOURCE)
public @interface Saveflags {}

/**
 * Restore the current matrix when restore() is called.
 *
 * @deprecated Use the flagless version of {@link #save()}, {@link #saveLayer(RectF, Paint)} or
 *             {@link #saveLayerAlpha(RectF, int)}. For saveLayer() calls the matrix
 *             was always restored for {@link #isHardwareAccelerated() Hardware accelerated}
 *             canvases and as of API level {@value Build.VERSION_CODES#O} that is the default
 *             behavior for all canvas types.
 */
public static final int MATRIX_SAVE_FLAG = 0x01;

/**
 * Restore the current clip when restore() is called.
 *
 * @deprecated Use the flagless version of {@link #save()}, {@link #saveLayer(RectF, Paint)} or
 *             {@link #saveLayerAlpha(RectF, int)}. For saveLayer() calls the clip
 *             was always restored for {@link #isHardwareAccelerated() Hardware accelerated}
 *             canvases and as of API level {@value Build.VERSION_CODES#O} that is the default
 *             behavior for all canvas types.
 */
public static final int CLIP_SAVE_FLAG = 0x02;

/**
 * The layer requires a per-pixel alpha channel.
 *
 * @deprecated This flag is ignored. Use the flagless version of {@link #saveLayer(RectF, Paint)}
 *             {@link #saveLayerAlpha(RectF, int)}.
 */
public static final int HAS_ALPHA_LAYER_SAVE_FLAG = 0x04;

/**
 * The layer requires full 8-bit precision for each color channel.
 *
 * @deprecated This flag is ignored. Use the flagless version of {@link #saveLayer(RectF, Paint)}
 *             {@link #saveLayerAlpha(RectF, int)}.
 */
```

```
public static final int FULL_COLOR_LAYER_SAVE_FLAG = 0x08;

/**
 * Clip drawing to the bounds of the offscreen layer, omit at your own peril.
 * <p class="note"><strong>Note:</strong> it is strongly recommended to not
 * omit this flag for any call to <code>saveLayer()</code> and
 * <code>saveLayerAlpha()</code> variants. Not passing this flag generally
 * triggers extremely poor performance with hardware accelerated rendering.
 *
 * @deprecated This flag results in poor performance and the same effect can be achieved with
 *             a single layer or multiple draw commands with different clips.
 *
 */
public static final int CLIP_TO_LAYER_SAVE_FLAG = 0x10;

/**
 * Restore everything when restore() is called (standard save flags).
 * <p class="note"><strong>Note:</strong> for performance reasons, it is
 * strongly recommended to pass this - the complete set of flags - to any
 * call to <code>saveLayer()</code> and <code>saveLayerAlpha()</code>
 * variants.
 *
 * <p class="note"><strong>Note:</strong> all methods that accept this flag
 * have flagless versions that are equivalent to passing this flag.
 */
public static final int ALL_SAVE_FLAG = 0x1F;

/**
 * Saves the current matrix and clip onto a private stack.
 * <p>
 * Subsequent calls to translate,scale,rotate,skew,concat or clipRect,
 * clipPath will all operate as usual, but when the balancing call to
 * restore() is made, those calls will be forgotten, and the settings that
 * existed before the save() will be reinstated.
 *
 * @return The value to pass to restoreToCount() to balance this save()
 */
public int save() {
    return nSave(mNativeCanvasWrapper, MATRIX_SAVE_FLAG | CLIP_SAVE_FLAG);
}

/**
 * Based on saveFlags, can save the current matrix and clip onto a private
 * stack.
 * <p class="note"><strong>Note:</strong> if possible, use the
 * parameter-less save(). It is simpler and faster than individually
 * disabling the saving of matrix or clip with this method.
 * <p>
 * Subsequent calls to translate,scale,rotate,skew,concat or clipRect,
 * clipPath will all operate as usual, but when the balancing call to
 * restore() is made, those calls will be forgotten, and the settings that
 * existed before the save() will be reinstated.
 *
 * @deprecated Use {@link #save()} instead.
 * @param saveFlags flag bits that specify which parts of the Canvas state
 *                  to save/restore
 * @return The value to pass to restoreToCount() to balance this save()
 */
public int save(@Saveflags int saveFlags) {
    return nSave(mNativeCanvasWrapper, saveFlags);
}

/**
 * This behaves the same as save(), but in addition it allocates and
 * redirects drawing to an offscreen bitmap.
 * <p class="note"><strong>Note:</strong> this method is very expensive,
 * incurring more than double rendering cost for contained content. Avoid
 * using this method, especially if the bounds provided are large, or if
 * the {@link #CLIP_TO_LAYER_SAVE_FLAG} is omitted from the
 * {@code saveFlags} parameter. It is recommended to use a
```

```
     * {@link android.view.View#LAYER_TYPE_HARDWARE hardware layer} on a View
     * to apply an xfermode, color filter, or alpha, as it will perform much
     * better than this method.
     * <p>
     * All drawing calls are directed to a newly allocated offscreen bitmap.
     * Only when the balancing call to restore() is made, is that offscreen
     * buffer drawn back to the current target of the Canvas (either the
     * screen, it's target Bitmap, or the previous layer).
     * <p>
     * Attributes of the Paint - {@link Paint#getAlpha() alpha},
     * {@link Paint#getXfermode() Xfermode}, and
     * {@link Paint#getColorFilter() ColorFilter} are applied when the
     * offscreen bitmap is drawn back when restore() is called.
     *
     * @deprecated Use {@link #saveLayer(RectF, Paint)} instead.
     * @param bounds May be null. The maximum size the offscreen bitmap
     *               needs to be (in local coordinates)
     * @param paint  This is copied, and is applied to the offscreen when
     *               restore() is called.
     * @param saveFlags see _SAVE_FLAG constants, generally {@link #ALL_SAVE_FLAG} is recommended
     *               for performance reasons.
     * @return       value to pass to restoreToCount() to balance this save()
     */
    public int saveLayer(@Nullable RectF bounds, @Nullable Paint paint, @Saveflags int saveFlags) {
        if (bounds == null) {
            bounds = new RectF(getClipBounds());
        }
        return saveLayer(bounds.left, bounds.top, bounds.right, bounds.bottom, paint, saveFlags);
    }

    /**
     * This behaves the same as save(), but in addition it allocates and
     * redirects drawing to an offscreen rendering target.
     * <p class="note"><strong>Note:</strong> this method is very expensive,
     * incurring more than double rendering cost for contained content. Avoid
     * using this method when possible and instead use a
     * {@link android.view.View#LAYER_TYPE_HARDWARE hardware layer} on a View
     * to apply an xfermode, color filter, or alpha, as it will perform much
     * better than this method.
     * <p>
     * All drawing calls are directed to a newly allocated offscreen rendering target.
     * Only when the balancing call to restore() is made, is that offscreen
     * buffer drawn back to the current target of the Canvas (which can potentially be a previous
     * layer if these calls are nested).
     * <p>
     * Attributes of the Paint - {@link Paint#getAlpha() alpha},
     * {@link Paint#getXfermode() Xfermode}, and
     * {@link Paint#getColorFilter() ColorFilter} are applied when the
     * offscreen rendering target is drawn back when restore() is called.
     *
     * @param bounds May be null. The maximum size the offscreen render target
     *               needs to be (in local coordinates)
     * @param paint  This is copied, and is applied to the offscreen when
     *               restore() is called.
     * @return       value to pass to restoreToCount() to balance this save()
     */
    public int saveLayer(@Nullable RectF bounds, @Nullable Paint paint) {
        return saveLayer(bounds, paint, ALL_SAVE_FLAG);
    }

    /**
     * Helper version of saveLayer() that takes 4 values rather than a RectF.
     *
     * @deprecated Use {@link #saveLayer(float, float, float, float, Paint)} instead.
     */
    public int saveLayer(float left, float top, float right, float bottom, @Nullable Paint paint,
            @Saveflags int saveFlags) {
        return nSaveLayer(mNativeCanvasWrapper, left, top, right, bottom,
                paint != null ? paint.getNativeInstance() : 0,
                saveFlags);
```

```
}

/**
 * Convenience for {@link #saveLayer(RectF, Paint)} that takes the four float coordinates of the
 * bounds rectangle.
 */
public int saveLayer(float left, float top, float right, float bottom, @Nullable Paint paint) {
    return saveLayer(left, top, right, bottom, paint, ALL_SAVE_FLAG);
}

/**
 * This behaves the same as save(), but in addition it allocates and
 * redirects drawing to an offscreen bitmap.
 * <p class="note"><strong>Note:</strong> this method is very expensive,
 * incurring more than double rendering cost for contained content. Avoid
 * using this method, especially if the bounds provided are large, or if
 * the {@link #CLIP_TO_LAYER_SAVE_FLAG} is omitted from the
 * {@code saveFlags} parameter. It is recommended to use a
 * {@link android.view.View#LAYER_TYPE_HARDWARE hardware layer} on a View
 * to apply an xfermode, color filter, or alpha, as it will perform much
 * better than this method.
 * <p>
 * All drawing calls are directed to a newly allocated offscreen bitmap.
 * Only when the balancing call to restore() is made, is that offscreen
 * buffer drawn back to the current target of the Canvas (either the
 * screen, it's target Bitmap, or the previous layer).
 * <p>
 * The {@code alpha} parameter is applied when the offscreen bitmap is
 * drawn back when restore() is called.
 *
 * @deprecated Use {@link #saveLayerAlpha(RectF, int)} instead.
 * @param bounds    The maximum size the offscreen bitmap needs to be
 *                  (in local coordinates)
 * @param alpha     The alpha to apply to the offscreen when it is
 *                  drawn during restore()
 * @param saveFlags see _SAVE_FLAG constants, generally {@link #ALL_SAVE_FLAG} is recommended
 *                  for performance reasons.
 * @return          value to pass to restoreToCount() to balance this call
 */
public int saveLayerAlpha(@Nullable RectF bounds, int alpha, @Saveflags int saveFlags) {
    if (bounds == null) {
        bounds = new RectF(getClipBounds());
    }
    return saveLayerAlpha(bounds.left, bounds.top, bounds.right, bounds.bottom, alpha, saveFlags);
}

/**
 * Convenience for {@link #saveLayer(RectF, Paint)} but instead of taking a entire Paint object
 * it takes only the {@code alpha} parameter.
 *
 * @param bounds    The maximum size the offscreen bitmap needs to be
 *                  (in local coordinates)
 * @param alpha     The alpha to apply to the offscreen when it is
 *                  drawn during restore()
 */
public int saveLayerAlpha(@Nullable RectF bounds, int alpha) {
    return saveLayerAlpha(bounds, alpha, ALL_SAVE_FLAG);
}

/**
 * Helper for saveLayerAlpha() that takes 4 values instead of a RectF.
 *
 * @deprecated Use {@link #saveLayerAlpha(float, float, float, float, int)} instead.
 */
public int saveLayerAlpha(float left, float top, float right, float bottom, int alpha,
        @Saveflags int saveFlags) {
    alpha = Math.min(255, Math.max(0, alpha));
    return nSaveLayerAlpha(mNativeCanvasWrapper, left, top, right, bottom,
                                alpha, saveFlags);
}
```

```
/**
 * Convenience for {@link #saveLayerAlpha(RectF, int)} that takes the four float coordinates of
 * the bounds rectangle.
 */
public int saveLayerAlpha(float left, float top, float right, float bottom, int alpha) {
    return saveLayerAlpha(left, top, right, bottom, alpha, ALL_SAVE_FLAG);
}

/**
 * This call balances a previous call to save(), and is used to remove all
 * modifications to the matrix/clip state since the last save call. It is
 * an error to call restore() more times than save() was called.
 */
public void restore() {
    if (!nRestore(mNativeCanvasWrapper)
            && (!sCompatibilityRestore || !isHardwareAccelerated())) {
        throw new IllegalStateException("Underflow in restore - more restores than saves");
    }
}

/**
 * Returns the number of matrix/clip states on the Canvas' private stack.
 * This will equal # save() calls - # restore() calls.
 */
public int getSaveCount() {
    return nGetSaveCount(mNativeCanvasWrapper);
}

/**
 * Efficient way to pop any calls to save() that happened after the save
 * count reached saveCount. It is an error for saveCount to be less than 1.
 *
 * Example:
 *    int count = canvas.save();
 *    ... // more calls potentially to save()
 *    canvas.restoreToCount(count);
 *    // now the canvas is back in the same state it was before the initial
 *    // call to save().
 *
 * @param saveCount The save level to restore to.
 */
public void restoreToCount(int saveCount) {
    if (saveCount < 1) {
        if (!sCompatibilityRestore || !isHardwareAccelerated()) {
            // do nothing and throw without restoring
            throw new IllegalArgumentException(
                    "Underflow in restoreToCount - more restores than saves");
        }
        // compat behavior - restore as far as possible
        saveCount = 1;
    }
    nRestoreToCount(mNativeCanvasWrapper, saveCount);
}

/**
 * Preconcat the current matrix with the specified translation
 *
 * @param dx The distance to translate in X
 * @param dy The distance to translate in Y
*/
public void translate(float dx, float dy) {
    if (dx == 0.0f && dy == 0.0f) return;
    nTranslate(mNativeCanvasWrapper, dx, dy);
}

/**
 * Preconcat the current matrix with the specified scale.
 *
 * @param sx The amount to scale in X
```

```
     * @param sy The amount to scale in Y
     */
    public void scale(float sx, float sy) {
        if (sx == 1.0f && sy == 1.0f) return;
        nScale(mNativeCanvasWrapper, sx, sy);
    }

    /**
     * Preconcat the current matrix with the specified scale.
     *
     * @param sx The amount to scale in X
     * @param sy The amount to scale in Y
     * @param px The x-coord for the pivot point (unchanged by the scale)
     * @param py The y-coord for the pivot point (unchanged by the scale)
     */
    public final void scale(float sx, float sy, float px, float py) {
        if (sx == 1.0f && sy == 1.0f) return;
        translate(px, py);
        scale(sx, sy);
        translate(-px, -py);
    }

    /**
     * Preconcat the current matrix with the specified rotation.
     *
     * @param degrees The amount to rotate, in degrees
     */
    public void rotate(float degrees) {
        if (degrees == 0.0f) return;
        nRotate(mNativeCanvasWrapper, degrees);
    }

    /**
     * Preconcat the current matrix with the specified rotation.
     *
     * @param degrees The amount to rotate, in degrees
     * @param px The x-coord for the pivot point (unchanged by the rotation)
     * @param py The y-coord for the pivot point (unchanged by the rotation)
     */
    public final void rotate(float degrees, float px, float py) {
        if (degrees == 0.0f) return;
        translate(px, py);
        rotate(degrees);
        translate(-px, -py);
    }

    /**
     * Preconcat the current matrix with the specified skew.
     *
     * @param sx The amount to skew in X
     * @param sy The amount to skew in Y
     */
    public void skew(float sx, float sy) {
        if (sx == 0.0f && sy == 0.0f) return;
        nSkew(mNativeCanvasWrapper, sx, sy);
    }

    /**
     * Preconcat the current matrix with the specified matrix. If the specified
     * matrix is null, this method does nothing.
     *
     * @param matrix The matrix to preconcatenate with the current matrix
     */
    public void concat(@Nullable Matrix matrix) {
        if (matrix != null) nConcat(mNativeCanvasWrapper, matrix.native_instance);
    }

    /**
     * Completely replace the current matrix with the specified matrix. If the
     * matrix parameter is null, then the current matrix is reset to identity.
```

```
     *
     * <strong>Note:</strong> it is recommended to use {@link #concat(Matrix)},
     * {@link #scale(float, float)}, {@link #translate(float, float)} and
     * {@link #rotate(float)} instead of this method.
     *
     * @param matrix The matrix to replace the current matrix with. If it is
     *               null, set the current matrix to identity.
     *
     * @see #concat(Matrix)
     */
    public void setMatrix(@Nullable Matrix matrix) {
        nSetMatrix(mNativeCanvasWrapper,
                        matrix == null ? 0 : matrix.native_instance);
    }

    /**
     * Return, in ctm, the current transformation matrix. This does not alter
     * the matrix in the canvas, but just returns a copy of it.
     *
     * @deprecated {@link #isHardwareAccelerated() Hardware accelerated} canvases may have any
     * matrix when passed to a View or Drawable, as it is implementation defined where in the
     * hierarchy such canvases are created. It is recommended in such cases to either draw contents
     * irrespective of the current matrix, or to track relevant transform state outside of the
     * canvas.
     */
    @Deprecated
    public void getMatrix(@NonNull Matrix ctm) {
        nGetMatrix(mNativeCanvasWrapper, ctm.native_instance);
    }

    /**
     * Return a new matrix with a copy of the canvas' current transformation
     * matrix.
     *
     * @deprecated {@link #isHardwareAccelerated() Hardware accelerated} canvases may have any
     * matrix when passed to a View or Drawable, as it is implementation defined where in the
     * hierarchy such canvases are created. It is recommended in such cases to either draw contents
     * irrespective of the current matrix, or to track relevant transform state outside of the
     * canvas.
     */
    @Deprecated
    public final @NonNull Matrix getMatrix() {
        Matrix m = new Matrix();
        //noinspection deprecation
        getMatrix(m);
        return m;
    }

    /**
     * Modify the current clip with the specified rectangle.
     *
     * @param rect The rect to intersect with the current clip
     * @param op How the clip is modified
     * @return true if the resulting clip is non-empty
     *
     * @deprecated Region.Op values other than {@link Region.Op#INTERSECT} and
     * {@link Region.Op#DIFFERENCE} have the ability to expand the clip. The canvas clipping APIs
     * are intended to only expand the clip as a result of a restore operation. This enables a view
     * parent to clip a canvas to clearly define the maximal drawing area of its children. The
     * recommended alternative calls are {@link #clipRect(RectF)} and {@link #clipOutRect(RectF)};
     */
    @Deprecated
    public boolean clipRect(@NonNull RectF rect, @NonNull Region.Op op) {
        return nClipRect(mNativeCanvasWrapper, rect.left, rect.top, rect.right, rect.bottom,
                op.nativeInt);
    }

    /**
     * Modify the current clip with the specified rectangle, which is
     * expressed in local coordinates.
```

```
     *
     * @param rect The rectangle to intersect with the current clip.
     * @param op How the clip is modified
     * @return true if the resulting clip is non-empty
     *
     * @deprecated Region.Op values other than {@link Region.Op#INTERSECT} and
     * {@link Region.Op#DIFFERENCE} have the ability to expand the clip. The canvas clipping APIs
     * are intended to only expand the clip as a result of a restore operation. This enables a view
     * parent to clip a canvas to clearly define the maximal drawing area of its children. The
     * recommended alternative calls are {@link #clipRect(Rect)} and {@link #clipOutRect(Rect)};
     */
    @Deprecated
    public boolean clipRect(@NonNull Rect rect, @NonNull Region.Op op) {
        return nClipRect(mNativeCanvasWrapper, rect.left, rect.top, rect.right, rect.bottom,
                op.nativeInt);
    }

    /**
     * Intersect the current clip with the specified rectangle, which is
     * expressed in local coordinates.
     *
     * @param rect The rectangle to intersect with the current clip.
     * @return true if the resulting clip is non-empty
     */
    public boolean clipRect(@NonNull RectF rect) {
        return nClipRect(mNativeCanvasWrapper, rect.left, rect.top, rect.right, rect.bottom,
                Region.Op.INTERSECT.nativeInt);
    }

    /**
     * Set the clip to the difference of the current clip and the specified rectangle, which is
     * expressed in local coordinates.
     *
     * @param rect The rectangle to perform a difference op with the current clip.
     * @return true if the resulting clip is non-empty
     */
    public boolean clipOutRect(@NonNull RectF rect) {
        return nClipRect(mNativeCanvasWrapper, rect.left, rect.top, rect.right, rect.bottom,
                Region.Op.DIFFERENCE.nativeInt);
    }

    /**
     * Intersect the current clip with the specified rectangle, which is
     * expressed in local coordinates.
     *
     * @param rect The rectangle to intersect with the current clip.
     * @return true if the resulting clip is non-empty
     */
    public boolean clipRect(@NonNull Rect rect) {
        return nClipRect(mNativeCanvasWrapper, rect.left, rect.top, rect.right, rect.bottom,
                Region.Op.INTERSECT.nativeInt);
    }

    /**
     * Set the clip to the difference of the current clip and the specified rectangle, which is
     * expressed in local coordinates.
     *
     * @param rect The rectangle to perform a difference op with the current clip.
     * @return true if the resulting clip is non-empty
     */
    public boolean clipOutRect(@NonNull Rect rect) {
        return nClipRect(mNativeCanvasWrapper, rect.left, rect.top, rect.right, rect.bottom,
                Region.Op.DIFFERENCE.nativeInt);
    }

    /**
     * Modify the current clip with the specified rectangle, which is
     * expressed in local coordinates.
     *
     * @param left   The left side of the rectangle to intersect with the
```

```
     *               current clip
     * @param top    The top of the rectangle to intersect with the current
     *               clip
     * @param right  The right side of the rectangle to intersect with the
     *               current clip
     * @param bottom The bottom of the rectangle to intersect with the current
     *               clip
     * @param op     How the clip is modified
     * @return       true if the resulting clip is non-empty
     *
     * @deprecated Region.Op values other than {@link Region.Op#INTERSECT} and
     * {@link Region.Op#DIFFERENCE} have the ability to expand the clip. The canvas clipping APIs
     * are intended to only expand the clip as a result of a restore operation. This enables a view
     * parent to clip a canvas to clearly define the maximal drawing area of its children. The
     * recommended alternative calls are {@link #clipRect(float,float,float,float)} and
     * {@link #clipOutRect(float,float,float,float)};
     */
    @Deprecated
    public boolean clipRect(float left, float top, float right, float bottom,
            @NonNull Region.Op op) {
        return nClipRect(mNativeCanvasWrapper, left, top, right, bottom, op.nativeInt);
    }

    /**
     * Intersect the current clip with the specified rectangle, which is
     * expressed in local coordinates.
     *
     * @param left   The left side of the rectangle to intersect with the
     *               current clip
     * @param top    The top of the rectangle to intersect with the current clip
     * @param right  The right side of the rectangle to intersect with the
     *               current clip
     * @param bottom The bottom of the rectangle to intersect with the current
     *               clip
     * @return       true if the resulting clip is non-empty
     */
    public boolean clipRect(float left, float top, float right, float bottom) {
        return nClipRect(mNativeCanvasWrapper, left, top, right, bottom,
                Region.Op.INTERSECT.nativeInt);
    }

    /**
     * Set the clip to the difference of the current clip and the specified rectangle, which is
     * expressed in local coordinates.
     *
     * @param left   The left side of the rectangle used in the difference operation
     * @param top    The top of the rectangle used in the difference operation
     * @param right  The right side of the rectangle used in the difference operation
     * @param bottom The bottom of the rectangle used in the difference operation
     * @return       true if the resulting clip is non-empty
     */
    public boolean clipOutRect(float left, float top, float right, float bottom) {
        return nClipRect(mNativeCanvasWrapper, left, top, right, bottom,
                Region.Op.DIFFERENCE.nativeInt);
    }

    /**
     * Intersect the current clip with the specified rectangle, which is
     * expressed in local coordinates.
     *
     * @param left   The left side of the rectangle to intersect with the
     *               current clip
     * @param top    The top of the rectangle to intersect with the current clip
     * @param right  The right side of the rectangle to intersect with the
     *               current clip
     * @param bottom The bottom of the rectangle to intersect with the current
     *               clip
     * @return       true if the resulting clip is non-empty
     */
    public boolean clipRect(int left, int top, int right, int bottom) {
```

```
    return nClipRect(mNativeCanvasWrapper, left, top, right, bottom,
            Region.Op.INTERSECT.nativeInt);
}

/**
 * Set the clip to the difference of the current clip and the specified rectangle, which is
 * expressed in local coordinates.
 *
 * @param left   The left side of the rectangle used in the difference operation
 * @param top    The top of the rectangle used in the difference operation
 * @param right  The right side of the rectangle used in the difference operation
 * @param bottom The bottom of the rectangle used in the difference operation
 * @return       true if the resulting clip is non-empty
 */
public boolean clipOutRect(int left, int top, int right, int bottom) {
    return nClipRect(mNativeCanvasWrapper, left, top, right, bottom,
            Region.Op.DIFFERENCE.nativeInt);
}

/**
    * Modify the current clip with the specified path.
 *
 * @param path The path to operate on the current clip
 * @param op   How the clip is modified
 * @return     true if the resulting is non-empty
 *
 * @deprecated Region.Op values other than {@link Region.Op#INTERSECT} and
 * {@link Region.Op#DIFFERENCE} have the ability to expand the clip. The canvas clipping APIs
 * are intended to only expand the clip as a result of a restore operation. This enables a view
 * parent to clip a canvas to clearly define the maximal drawing area of its children. The
 * recommended alternative calls are {@link #clipPath(Path)} and
 * {@link #clipOutPath(Path)};
 */
@Deprecated
public boolean clipPath(@NonNull Path path, @NonNull Region.Op op) {
    return nClipPath(mNativeCanvasWrapper, path.readOnlyNI(), op.nativeInt);
}

/**
 * Intersect the current clip with the specified path.
 *
 * @param path The path to intersect with the current clip
 * @return     true if the resulting clip is non-empty
 */
public boolean clipPath(@NonNull Path path) {
    return clipPath(path, Region.Op.INTERSECT);
}

/**
 * Set the clip to the difference of the current clip and the specified path.
 *
 * @param path The path used in the difference operation
 * @return     true if the resulting clip is non-empty
 */
public boolean clipOutPath(@NonNull Path path) {
    return clipPath(path, Region.Op.DIFFERENCE);
}

/**
 * Modify the current clip with the specified region. Note that unlike
 * clipRect() and clipPath() which transform their arguments by the
 * current matrix, clipRegion() assumes its argument is already in the
 * coordinate system of the current layer's bitmap, and so not
 * transformation is performed.
 *
 * @param region The region to operate on the current clip, based on op
 * @param op How the clip is modified
 * @return true if the resulting is non-empty
 *
 * @removed
```

```
     * @deprecated Unlike all other clip calls this API does not respect the
     *             current matrix. Use {@link #clipRect(Rect)} as an alternative.
     */
    @Deprecated
    public boolean clipRegion(@NonNull Region region, @NonNull Region.Op op) {
        return false;
    }

    /**
     * Intersect the current clip with the specified region. Note that unlike
     * clipRect() and clipPath() which transform their arguments by the
     * current matrix, clipRegion() assumes its argument is already in the
     * coordinate system of the current layer's bitmap, and so not
     * transformation is performed.
     *
     * @param region The region to operate on the current clip, based on op
     * @return true if the resulting is non-empty
     *
     * @removed
     * @deprecated Unlike all other clip calls this API does not respect the
     *             current matrix. Use {@link #clipRect(Rect)} as an alternative.
     */
    @Deprecated
    public boolean clipRegion(@NonNull Region region) {
        return false;
    }

    public @Nullable DrawFilter getDrawFilter() {
        return mDrawFilter;
    }

    public void setDrawFilter(@Nullable DrawFilter filter) {
        long nativeFilter = 0;
        if (filter != null) {
            nativeFilter = filter.mNativeInt;
        }
        mDrawFilter = filter;
        nSetDrawFilter(mNativeCanvasWrapper, nativeFilter);
    }

    /**
     * Constant values used as parameters to {@code quickReject()} calls. These values
     * specify how much space around the shape should be accounted for, depending on whether
     * the shaped area is antialiased or not.
     *
     * @see #quickReject(float, float, float, float, EdgeType)
     * @see #quickReject(Path, EdgeType)
     * @see #quickReject(RectF, EdgeType)
     */
    public enum EdgeType {

        /**
         * Black-and-White: Treat edges by just rounding to nearest pixel boundary
         */
        BW(0),  //!< treat edges by just rounding to nearest pixel boundary

        /**
         * Antialiased: Treat edges by rounding-out, since they may be antialiased
         */
        AA(1);

        EdgeType(int nativeInt) {
            this.nativeInt = nativeInt;
        }

        /**
         * @hide
         */
        public final int nativeInt;
    }
```

```
/**
 * Return true if the specified rectangle, after being transformed by the
 * current matrix, would lie completely outside of the current clip. Call
 * this to check if an area you intend to draw into is clipped out (and
 * therefore you can skip making the draw calls).
 *
 * @param rect  the rect to compare with the current clip
 * @param type  {@link Canvas.EdgeType#AA} if the path should be considered antialiased,
 *              since that means it may affect a larger area (more pixels) than
 *              non-antialiased ({@link Canvas.EdgeType#BW}).
 * @return      true if the rect (transformed by the canvas' matrix)
 *              does not intersect with the canvas' clip
 */
public boolean quickReject(@NonNull RectF rect, @NonNull EdgeType type) {
    return nQuickReject(mNativeCanvasWrapper,
            rect.left, rect.top, rect.right, rect.bottom);
}

/**
 * Return true if the specified path, after being transformed by the
 * current matrix, would lie completely outside of the current clip. Call
 * this to check if an area you intend to draw into is clipped out (and
 * therefore you can skip making the draw calls). Note: for speed it may
 * return false even if the path itself might not intersect the clip
 * (i.e. the bounds of the path intersects, but the path does not).
 *
 * @param path        The path to compare with the current clip
 * @param type        {@link Canvas.EdgeType#AA} if the path should be considered antialiased,
 *                    since that means it may affect a larger area (more pixels) than
 *                    non-antialiased ({@link Canvas.EdgeType#BW}).
 * @return            true if the path (transformed by the canvas' matrix)
 *                    does not intersect with the canvas' clip
 */
public boolean quickReject(@NonNull Path path, @NonNull EdgeType type) {
    return nQuickReject(mNativeCanvasWrapper, path.readOnlyNI());
}

/**
 * Return true if the specified rectangle, after being transformed by the
 * current matrix, would lie completely outside of the current clip. Call
 * this to check if an area you intend to draw into is clipped out (and
 * therefore you can skip making the draw calls).
 *
 * @param left        The left side of the rectangle to compare with the
 *                    current clip
 * @param top         The top of the rectangle to compare with the current
 *                    clip
 * @param right       The right side of the rectangle to compare with the
 *                    current clip
 * @param bottom      The bottom of the rectangle to compare with the
 *                    current clip
 * @param type        {@link Canvas.EdgeType#AA} if the path should be considered antialiased,
 *                    since that means it may affect a larger area (more pixels) than
 *                    non-antialiased ({@link Canvas.EdgeType#BW}).
 * @return            true if the rect (transformed by the canvas' matrix)
 *                    does not intersect with the canvas' clip
 */
public boolean quickReject(float left, float top, float right, float bottom,
        @NonNull EdgeType type) {
    return nQuickReject(mNativeCanvasWrapper, left, top, right, bottom);
}

/**
 * Return the bounds of the current clip (in local coordinates) in the
 * bounds parameter, and return true if it is non-empty. This can be useful
 * in a way similar to quickReject, in that it tells you that drawing
 * outside of these bounds will be clipped out.
 *
 * @param bounds Return the clip bounds here. If it is null, ignore it but
```

```
     *               still return true if the current clip is non-empty.
     * @return true if the current clip is non-empty.
     */
    public boolean getClipBounds(@Nullable Rect bounds) {
        return nGetClipBounds(mNativeCanvasWrapper, bounds);
    }

    /**
     * Retrieve the bounds of the current clip (in local coordinates).
     *
     * @return the clip bounds, or [0, 0, 0, 0] if the clip is empty.
     */
    public final @NonNull Rect getClipBounds() {
        Rect r = new Rect();
        getClipBounds(r);
        return r;
    }

    /**
     * Save the canvas state, draw the picture, and restore the canvas state.
     * This differs from picture.draw(canvas), which does not perform any
     * save/restore.
     *
     * <p>
     * <strong>Note:</strong> This forces the picture to internally call
     * {@link Picture#endRecording} in order to prepare for playback.
     *
     * @param picture  The picture to be drawn
     */
    public void drawPicture(@NonNull Picture picture) {
        picture.endRecording();
        int restoreCount = save();
        picture.draw(this);
        restoreToCount(restoreCount);
    }

    /**
     * Draw the picture, stretched to fit into the dst rectangle.
     */
    public void drawPicture(@NonNull Picture picture, @NonNull RectF dst) {
        save();
        translate(dst.left, dst.top);
        if (picture.getWidth() > 0 && picture.getHeight() > 0) {
            scale(dst.width() / picture.getWidth(), dst.height() / picture.getHeight());
        }
        drawPicture(picture);
        restore();
    }

    /**
     * Draw the picture, stretched to fit into the dst rectangle.
     */
    public void drawPicture(@NonNull Picture picture, @NonNull Rect dst) {
        save();
        translate(dst.left, dst.top);
        if (picture.getWidth() > 0 && picture.getHeight() > 0) {
            scale((float) dst.width() / picture.getWidth(),
                    (float) dst.height() / picture.getHeight());
        }
        drawPicture(picture);
        restore();
    }

    public enum VertexMode {
        TRIANGLES(0),
        TRIANGLE_STRIP(1),
        TRIANGLE_FAN(2);

        VertexMode(int nativeInt) {
            this.nativeInt = nativeInt;
```

```
    }

    /**
     * @hide
     */
    public final int nativeInt;
}

/**
 * Releases the resources associated with this canvas.
 *
 * @hide
 */
public void release() {
    mNativeCanvasWrapper = 0;
    if (mFinalizer != null) {
        mFinalizer.run();
        mFinalizer = null;
    }
}

/**
 * Free up as much memory as possible from private caches (e.g. fonts, images)
 *
 * @hide
 */
public static void freeCaches() {
    nFreeCaches();
}

/**
 * Free up text layout caches
 *
 * @hide
 */
public static void freeTextLayoutCaches() {
    nFreeTextLayoutCaches();
}

private static native void nFreeCaches();
private static native void nFreeTextLayoutCaches();
private static native long nInitRaster(Bitmap bitmap);
private static native long nGetNativeFinalizer();

// ---------------- @FastNative ------------------

@FastNative
private static native void nSetBitmap(long canvasHandle, Bitmap bitmap);

@FastNative
private static native boolean nGetClipBounds(long nativeCanvas, Rect bounds);

// ---------------- @CriticalNative ------------------

@CriticalNative
private static native boolean nIsOpaque(long canvasHandle);
@CriticalNative
private static native void nSetHighContrastText(long renderer, boolean highContrastText);
@CriticalNative
private static native int nGetWidth(long canvasHandle);
@CriticalNative
private static native int nGetHeight(long canvasHandle);

@CriticalNative
private static native int nSave(long canvasHandle, int saveFlags);
@CriticalNative
private static native int nSaveLayer(long nativeCanvas, float l, float t, float r, float b,
        long nativePaint, int layerFlags);
@CriticalNative
private static native int nSaveLayerAlpha(long nativeCanvas, float l, float t, float r, float b,
```

```
        int alpha, int layerFlags);
@CriticalNative
private static native boolean nRestore(long canvasHandle);
@CriticalNative
private static native void nRestoreToCount(long canvasHandle, int saveCount);
@CriticalNative
private static native int nGetSaveCount(long canvasHandle);

@CriticalNative
private static native void nTranslate(long canvasHandle, float dx, float dy);
@CriticalNative
private static native void nScale(long canvasHandle, float sx, float sy);
@CriticalNative
private static native void nRotate(long canvasHandle, float degrees);
@CriticalNative
private static native void nSkew(long canvasHandle, float sx, float sy);
@CriticalNative
private static native void nConcat(long nativeCanvas, long nativeMatrix);
@CriticalNative
private static native void nSetMatrix(long nativeCanvas, long nativeMatrix);
@CriticalNative
private static native boolean nClipRect(long nativeCanvas,
        float left, float top, float right, float bottom, int regionOp);
@CriticalNative
private static native boolean nClipPath(long nativeCanvas, long nativePath, int regionOp);
@CriticalNative
private static native void nSetDrawFilter(long nativeCanvas, long nativeFilter);
@CriticalNative
private static native void nGetMatrix(long nativeCanvas, long nativeMatrix);
@CriticalNative
private static native boolean nQuickReject(long nativeCanvas, long nativePath);
@CriticalNative
private static native boolean nQuickReject(long nativeCanvas, float left, float top,
        float right, float bottom);


// ---------------- Draw Methods ------------------

/**
 * <p>
 * Draw the specified arc, which will be scaled to fit inside the specified oval.
 * </p>
 * <p>
 * If the start angle is negative or >= 360, the start angle is treated as start angle modulo
 * 360.
 * </p>
 * <p>
 * If the sweep angle is >= 360, then the oval is drawn completely. Note that this differs
 * slightly from SkPath::arcTo, which treats the sweep angle modulo 360. If the sweep angle is
 * negative, the sweep angle is treated as sweep angle modulo 360
 * </p>
 * <p>
 * The arc is drawn clockwise. An angle of 0 degrees correspond to the geometric angle of 0
 * degrees (3 o'clock on a watch.)
 * </p>
 *
 * @param oval The bounds of oval used to define the shape and size of the arc
 * @param startAngle Starting angle (in degrees) where the arc begins
 * @param sweepAngle Sweep angle (in degrees) measured clockwise
 * @param useCenter If true, include the center of the oval in the arc, and close it if it is
 *            being stroked. This will draw a wedge
 * @param paint The paint used to draw the arc
 */
public void drawArc(@NonNull RectF oval, float startAngle, float sweepAngle, boolean useCenter,
        @NonNull Paint paint) {
    super.drawArc(oval, startAngle, sweepAngle, useCenter, paint);
}

/**
 * <p>
```

```
     * Draw the specified arc, which will be scaled to fit inside the specified oval.
     * </p>
     * <p>
     * If the start angle is negative or >= 360, the start angle is treated as start angle modulo
     * 360.
     * </p>
     * <p>
     * If the sweep angle is >= 360, then the oval is drawn completely. Note that this differs
     * slightly from SkPath::arcTo, which treats the sweep angle modulo 360. If the sweep angle is
     * negative, the sweep angle is treated as sweep angle modulo 360
     * </p>
     * <p>
     * The arc is drawn clockwise. An angle of 0 degrees correspond to the geometric angle of 0
     * degrees (3 o'clock on a watch.)
     * </p>
     *
     * @param startAngle Starting angle (in degrees) where the arc begins
     * @param sweepAngle Sweep angle (in degrees) measured clockwise
     * @param useCenter If true, include the center of the oval in the arc, and close it if it is
     *            being stroked. This will draw a wedge
     * @param paint The paint used to draw the arc
     */
    public void drawArc(float left, float top, float right, float bottom, float startAngle,
            float sweepAngle, boolean useCenter, @NonNull Paint paint) {
        super.drawArc(left, top, right, bottom, startAngle, sweepAngle, useCenter, paint);
    }

    /**
     * Fill the entire canvas' bitmap (restricted to the current clip) with the specified ARGB
     * color, using srcover porterduff mode.
     *
     * @param a alpha component (0..255) of the color to draw onto the canvas
     * @param r red component (0..255) of the color to draw onto the canvas
     * @param g green component (0..255) of the color to draw onto the canvas
     * @param b blue component (0..255) of the color to draw onto the canvas
     */
    public void drawARGB(int a, int r, int g, int b) {
        super.drawARGB(a, r, g, b);
    }

    /**
     * Draw the specified bitmap, with its top/left corner at (x,y), using the specified paint,
     * transformed by the current matrix.
     * <p>
     * Note: if the paint contains a maskfilter that generates a mask which extends beyond the
     * bitmap's original width/height (e.g. BlurMaskFilter), then the bitmap will be drawn as if it
     * were in a Shader with CLAMP mode. Thus the color outside of the original width/height will be
     * the edge color replicated.
     * <p>
     * If the bitmap and canvas have different densities, this function will take care of
     * automatically scaling the bitmap to draw at the same density as the canvas.
     *
     * @param bitmap The bitmap to be drawn
     * @param left The position of the left side of the bitmap being drawn
     * @param top The position of the top side of the bitmap being drawn
     * @param paint The paint used to draw the bitmap (may be null)
     */
    public void drawBitmap(@NonNull Bitmap bitmap, float left, float top, @Nullable Paint paint) {
        super.drawBitmap(bitmap, left, top, paint);
    }

    /**
     * Draw the specified bitmap, scaling/translating automatically to fill the destination
     * rectangle. If the source rectangle is not null, it specifies the subset of the bitmap to
     * draw.
     * <p>
     * Note: if the paint contains a maskfilter that generates a mask which extends beyond the
     * bitmap's original width/height (e.g. BlurMaskFilter), then the bitmap will be drawn as if it
     * were in a Shader with CLAMP mode. Thus the color outside of the original width/height will be
     * the edge color replicated.
```

```
     * <p>
     * This function <em>ignores the density associated with the bitmap</em>. This is because the
     * source and destination rectangle coordinate spaces are in their respective densities, so must
     * already have the appropriate scaling factor applied.
     *
     * @param bitmap The bitmap to be drawn
     * @param src May be null. The subset of the bitmap to be drawn
     * @param dst The rectangle that the bitmap will be scaled/translated to fit into
     * @param paint May be null. The paint used to draw the bitmap
     */
    public void drawBitmap(@NonNull Bitmap bitmap, @Nullable Rect src, @NonNull RectF dst,
            @Nullable Paint paint) {
        super.drawBitmap(bitmap, src, dst, paint);
    }

    /**
     * Draw the specified bitmap, scaling/translating automatically to fill the destination
     * rectangle. If the source rectangle is not null, it specifies the subset of the bitmap to
     * draw.
     * <p>
     * Note: if the paint contains a maskfilter that generates a mask which extends beyond the
     * bitmap's original width/height (e.g. BlurMaskFilter), then the bitmap will be drawn as if it
     * were in a Shader with CLAMP mode. Thus the color outside of the original width/height will be
     * the edge color replicated.
     * <p>
     * This function <em>ignores the density associated with the bitmap</em>. This is because the
     * source and destination rectangle coordinate spaces are in their respective densities, so must
     * already have the appropriate scaling factor applied.
     *
     * @param bitmap The bitmap to be drawn
     * @param src May be null. The subset of the bitmap to be drawn
     * @param dst The rectangle that the bitmap will be scaled/translated to fit into
     * @param paint May be null. The paint used to draw the bitmap
     */
    public void drawBitmap(@NonNull Bitmap bitmap, @Nullable Rect src, @NonNull Rect dst,
            @Nullable Paint paint) {
        super.drawBitmap(bitmap, src, dst, paint);
    }

    /**
     * Treat the specified array of colors as a bitmap, and draw it. This gives the same result as
     * first creating a bitmap from the array, and then drawing it, but this method avoids
     * explicitly creating a bitmap object which can be more efficient if the colors are changing
     * often.
     *
     * @param colors Array of colors representing the pixels of the bitmap
     * @param offset Offset into the array of colors for the first pixel
     * @param stride The number of colors in the array between rows (must be >= width or <= -width).
     * @param x The X coordinate for where to draw the bitmap
     * @param y The Y coordinate for where to draw the bitmap
     * @param width The width of the bitmap
     * @param height The height of the bitmap
     * @param hasAlpha True if the alpha channel of the colors contains valid values. If false, the
     *                 alpha byte is ignored (assumed to be 0xFF for every pixel).
     * @param paint May be null. The paint used to draw the bitmap
     * @deprecated Usage with a {@link #isHardwareAccelerated() hardware accelerated} canvas
     *             requires an internal copy of color buffer contents every time this method is
     *             called. Using a Bitmap avoids this copy, and allows the application to more
     *             explicitly control the lifetime and copies of pixel data.
     */
    @Deprecated
    public void drawBitmap(@NonNull int[] colors, int offset, int stride, float x, float y,
            int width, int height, boolean hasAlpha, @Nullable Paint paint) {
        super.drawBitmap(colors, offset, stride, x, y, width, height, hasAlpha, paint);
    }

    /**
     * Legacy version of drawBitmap(int[] colors, ...) that took ints for x,y
     *
     * @deprecated Usage with a {@link #isHardwareAccelerated() hardware accelerated} canvas
```

```
     *             requires an internal copy of color buffer contents every time this method is
     *             called. Using a Bitmap avoids this copy, and allows the application to more
     *             explicitly control the lifetime and copies of pixel data.
     */
    @Deprecated
    public void drawBitmap(@NonNull int[] colors, int offset, int stride, int x, int y,
            int width, int height, boolean hasAlpha, @Nullable Paint paint) {
        super.drawBitmap(colors, offset, stride, x, y, width, height, hasAlpha, paint);
    }

    /**
     * Draw the bitmap using the specified matrix.
     *
     * @param bitmap The bitmap to draw
     * @param matrix The matrix used to transform the bitmap when it is drawn
     * @param paint May be null. The paint used to draw the bitmap
     */
    public void drawBitmap(@NonNull Bitmap bitmap, @NonNull Matrix matrix, @Nullable Paint paint) {
        super.drawBitmap(bitmap, matrix, paint);
    }

    /**
     * Draw the bitmap through the mesh, where mesh vertices are evenly distributed across the
     * bitmap. There are meshWidth+1 vertices across, and meshHeight+1 vertices down. The verts
     * array is accessed in row-major order, so that the first meshWidth+1 vertices are distributed
     * across the top of the bitmap from left to right. A more general version of this method is
     * drawVertices().
     *
     * @param bitmap The bitmap to draw using the mesh
     * @param meshWidth The number of columns in the mesh. Nothing is drawn if this is 0
     * @param meshHeight The number of rows in the mesh. Nothing is drawn if this is 0
     * @param verts Array of x,y pairs, specifying where the mesh should be drawn. There must be at
     *             least (meshWidth+1) * (meshHeight+1) * 2 + vertOffset values in the array
     * @param vertOffset Number of verts elements to skip before drawing
     * @param colors May be null. Specifies a color at each vertex, which is interpolated across the
     *             cell, and whose values are multiplied by the corresponding bitmap colors. If not
     *             null, there must be at least (meshWidth+1) * (meshHeight+1) + colorOffset values
     *             in the array.
     * @param colorOffset Number of color elements to skip before drawing
     * @param paint May be null. The paint used to draw the bitmap
     */
    public void drawBitmapMesh(@NonNull Bitmap bitmap, int meshWidth, int meshHeight,
            @NonNull float[] verts, int vertOffset, @Nullable int[] colors, int colorOffset,
            @Nullable Paint paint) {
        super.drawBitmapMesh(bitmap, meshWidth, meshHeight, verts, vertOffset, colors, colorOffset,
                paint);
    }

    /**
     * Draw the specified circle using the specified paint. If radius is <= 0, then nothing will be
     * drawn. The circle will be filled or framed based on the Style in the paint.
     *
     * @param cx The x-coordinate of the center of the cirle to be drawn
     * @param cy The y-coordinate of the center of the cirle to be drawn
     * @param radius The radius of the cirle to be drawn
     * @param paint The paint used to draw the circle
     */
    public void drawCircle(float cx, float cy, float radius, @NonNull Paint paint) {
        super.drawCircle(cx, cy, radius, paint);
    }

    /**
     * Fill the entire canvas' bitmap (restricted to the current clip) with the specified color,
     * using srcover porterduff mode.
     *
     * @param color the color to draw onto the canvas
     */
    public void drawColor(@ColorInt int color) {
        super.drawColor(color);
    }
```

```
/**
 * Fill the entire canvas' bitmap (restricted to the current clip) with the specified color and
 * porter-duff xfermode.
 *
 * @param color the color to draw with
 * @param mode the porter-duff mode to apply to the color
 */
public void drawColor(@ColorInt int color, @NonNull PorterDuff.Mode mode) {
    super.drawColor(color, mode);
}

/**
 * Draw a line segment with the specified start and stop x,y coordinates, using the specified
 * paint.
 * <p>
 * Note that since a line is always "framed", the Style is ignored in the paint.
 * </p>
 * <p>
 * Degenerate lines (length is 0) will not be drawn.
 * </p>
 *
 * @param startX The x-coordinate of the start point of the line
 * @param startY The y-coordinate of the start point of the line
 * @param paint The paint used to draw the line
 */
public void drawLine(float startX, float startY, float stopX, float stopY,
        @NonNull Paint paint) {
    super.drawLine(startX, startY, stopX, stopY, paint);
}

/**
 * Draw a series of lines. Each line is taken from 4 consecutive values in the pts array. Thus
 * to draw 1 line, the array must contain at least 4 values. This is logically the same as
 * drawing the array as follows: drawLine(pts[0], pts[1], pts[2], pts[3]) followed by
 * drawLine(pts[4], pts[5], pts[6], pts[7]) and so on.
 *
 * @param pts Array of points to draw [x0 y0 x1 y1 x2 y2 ...]
 * @param offset Number of values in the array to skip before drawing.
 * @param count The number of values in the array to process, after skipping "offset" of them.
 *            Since each line uses 4 values, the number of "lines" that are drawn is really
 *            (count >> 2).
 * @param paint The paint used to draw the points
 */
public void drawLines(@Size(multiple = 4) @NonNull float[] pts, int offset, int count,
        @NonNull Paint paint) {
    super.drawLines(pts, offset, count, paint);
}

public void drawLines(@Size(multiple = 4) @NonNull float[] pts, @NonNull Paint paint) {
    super.drawLines(pts, paint);
}

/**
 * Draw the specified oval using the specified paint. The oval will be filled or framed based on
 * the Style in the paint.
 *
 * @param oval The rectangle bounds of the oval to be drawn
 */
public void drawOval(@NonNull RectF oval, @NonNull Paint paint) {
    super.drawOval(oval, paint);
}

/**
 * Draw the specified oval using the specified paint. The oval will be filled or framed based on
 * the Style in the paint.
 */
public void drawOval(float left, float top, float right, float bottom, @NonNull Paint paint) {
    super.drawOval(left, top, right, bottom, paint);
}
```

```
/**
 * Fill the entire canvas' bitmap (restricted to the current clip) with the specified paint.
 * This is equivalent (but faster) to drawing an infinitely large rectangle with the specified
 * paint.
 *
 * @param paint The paint used to draw onto the canvas
 */
public void drawPaint(@NonNull Paint paint) {
    super.drawPaint(paint);
}

/**
 * Draws the specified bitmap as an N-patch (most often, a 9-patches.)
 *
 * @param patch The ninepatch object to render
 * @param dst The destination rectangle.
 * @param paint The paint to draw the bitmap with. may be null
 * @hide
 */
public void drawPatch(@NonNull NinePatch patch, @NonNull Rect dst, @Nullable Paint paint) {
    super.drawPatch(patch, dst, paint);
}

/**
 * Draws the specified bitmap as an N-patch (most often, a 9-patches.)
 *
 * @param patch The ninepatch object to render
 * @param dst The destination rectangle.
 * @param paint The paint to draw the bitmap with. may be null
 * @hide
 */
public void drawPatch(@NonNull NinePatch patch, @NonNull RectF dst, @Nullable Paint paint) {
    super.drawPatch(patch, dst, paint);
}

/**
 * Draw the specified path using the specified paint. The path will be filled or framed based on
 * the Style in the paint.
 *
 * @param path The path to be drawn
 * @param paint The paint used to draw the path
 */
public void drawPath(@NonNull Path path, @NonNull Paint paint) {
    super.drawPath(path, paint);
}

/**
 * Helper for drawPoints() for drawing a single point.
 */
public void drawPoint(float x, float y, @NonNull Paint paint) {
    super.drawPoint(x, y, paint);
}

/**
 * Draw a series of points. Each point is centered at the coordinate specified by pts[], and its
 * diameter is specified by the paint's stroke width (as transformed by the canvas' CTM), with
 * special treatment for a stroke width of 0, which always draws exactly 1 pixel (or at most 4
 * if antialiasing is enabled). The shape of the point is controlled by the paint's Cap type.
 * The shape is a square, unless the cap type is Round, in which case the shape is a circle.
 *
 * @param pts Array of points to draw [x0 y0 x1 y1 x2 y2 ...]
 * @param offset Number of values to skip before starting to draw.
 * @param count The number of values to process, after skipping offset of them. Since one point
 *              uses two values, the number of "points" that are drawn is really (count >> 1).
 * @param paint The paint used to draw the points
 */
public void drawPoints(@Size(multiple = 2) float[] pts, int offset, int count,
        @NonNull Paint paint) {
    super.drawPoints(pts, offset, count, paint);
```

```
}

/**
 * Helper for drawPoints() that assumes you want to draw the entire array
 */
public void drawPoints(@Size(multiple = 2) @NonNull float[] pts, @NonNull Paint paint) {
    super.drawPoints(pts, paint);
}

/**
 * Draw the text in the array, with each character's origin specified by the pos array.
 *
 * @param text The text to be drawn
 * @param index The index of the first character to draw
 * @param count The number of characters to draw, starting from index.
 * @param pos Array of [x,y] positions, used to position each character
 * @param paint The paint used for the text (e.g. color, size, style)
 * @deprecated This method does not support glyph composition and decomposition and should
 *             therefore not be used to render complex scripts. It also doesn't handle
 *             supplementary characters (eg emoji).
 */
@Deprecated
public void drawPosText(@NonNull char[] text, int index, int count,
        @NonNull @Size(multiple = 2) float[] pos,
        @NonNull Paint paint) {
    super.drawPosText(text, index, count, pos, paint);
}

/**
 * Draw the text in the array, with each character's origin specified by the pos array.
 *
 * @param text The text to be drawn
 * @param pos Array of [x,y] positions, used to position each character
 * @param paint The paint used for the text (e.g. color, size, style)
 * @deprecated This method does not support glyph composition and decomposition and should
 *             therefore not be used to render complex scripts. It also doesn't handle
 *             supplementary characters (eg emoji).
 */
@Deprecated
public void drawPosText(@NonNull String text, @NonNull @Size(multiple = 2) float[] pos,
        @NonNull Paint paint) {
    super.drawPosText(text, pos, paint);
}

/**
 * Draw the specified Rect using the specified paint. The rectangle will be filled or framed
 * based on the Style in the paint.
 *
 * @param rect The rect to be drawn
 * @param paint The paint used to draw the rect
 */
public void drawRect(@NonNull RectF rect, @NonNull Paint paint) {
    super.drawRect(rect, paint);
}

/**
 * Draw the specified Rect using the specified Paint. The rectangle will be filled or framed
 * based on the Style in the paint.
 *
 * @param r The rectangle to be drawn.
 * @param paint The paint used to draw the rectangle
 */
public void drawRect(@NonNull Rect r, @NonNull Paint paint) {
    super.drawRect(r, paint);
}

/**
 * Draw the specified Rect using the specified paint. The rectangle will be filled or framed
 * based on the Style in the paint.
 *
```

```
     * @param left The left side of the rectangle to be drawn
     * @param top The top side of the rectangle to be drawn
     * @param right The right side of the rectangle to be drawn
     * @param bottom The bottom side of the rectangle to be drawn
     * @param paint The paint used to draw the rect
     */
    public void drawRect(float left, float top, float right, float bottom, @NonNull Paint paint) {
        super.drawRect(left, top, right, bottom, paint);
    }

    /**
     * Fill the entire canvas' bitmap (restricted to the current clip) with the specified RGB color,
     * using srcover porterduff mode.
     *
     * @param r red component (0..255) of the color to draw onto the canvas
     * @param g green component (0..255) of the color to draw onto the canvas
     * @param b blue component (0..255) of the color to draw onto the canvas
     */
    public void drawRGB(int r, int g, int b) {
        super.drawRGB(r, g, b);
    }

    /**
     * Draw the specified round-rect using the specified paint. The roundrect will be filled or
     * framed based on the Style in the paint.
     *
     * @param rect The rectangular bounds of the roundRect to be drawn
     * @param rx The x-radius of the oval used to round the corners
     * @param ry The y-radius of the oval used to round the corners
     * @param paint The paint used to draw the roundRect
     */
    public void drawRoundRect(@NonNull RectF rect, float rx, float ry, @NonNull Paint paint) {
        super.drawRoundRect(rect, rx, ry, paint);
    }

    /**
     * Draw the specified round-rect using the specified paint. The roundrect will be filled or
     * framed based on the Style in the paint.
     *
     * @param rx The x-radius of the oval used to round the corners
     * @param ry The y-radius of the oval used to round the corners
     * @param paint The paint used to draw the roundRect
     */
    public void drawRoundRect(float left, float top, float right, float bottom, float rx, float ry,
            @NonNull Paint paint) {
        super.drawRoundRect(left, top, right, bottom, rx, ry, paint);
    }

    /**
     * Draw the text, with origin at (x,y), using the specified paint. The origin is interpreted
     * based on the Align setting in the paint.
     *
     * @param text The text to be drawn
     * @param x The x-coordinate of the origin of the text being drawn
     * @param y The y-coordinate of the baseline of the text being drawn
     * @param paint The paint used for the text (e.g. color, size, style)
     */
    public void drawText(@NonNull char[] text, int index, int count, float x, float y,
            @NonNull Paint paint) {
        super.drawText(text, index, count, x, y, paint);
    }

    /**
     * Draw the text, with origin at (x,y), using the specified paint. The origin is interpreted
     * based on the Align setting in the paint.
     *
     * @param text The text to be drawn
     * @param x The x-coordinate of the origin of the text being drawn
     * @param y The y-coordinate of the baseline of the text being drawn
     * @param paint The paint used for the text (e.g. color, size, style)
```

```
     */
    public void drawText(@NonNull String text, float x, float y, @NonNull Paint paint) {
        super.drawText(text, x, y, paint);
    }

    /**
     * Draw the text, with origin at (x,y), using the specified paint. The origin is interpreted
     * based on the Align setting in the paint.
     *
     * @param text The text to be drawn
     * @param start The index of the first character in text to draw
     * @param end (end - 1) is the index of the last character in text to draw
     * @param x The x-coordinate of the origin of the text being drawn
     * @param y The y-coordinate of the baseline of the text being drawn
     * @param paint The paint used for the text (e.g. color, size, style)
     */
    public void drawText(@NonNull String text, int start, int end, float x, float y,
            @NonNull Paint paint) {
        super.drawText(text, start, end, x, y, paint);
    }

    /**
     * Draw the specified range of text, specified by start/end, with its origin at (x,y), in the
     * specified Paint. The origin is interpreted based on the Align setting in the Paint.
     *
     * @param text The text to be drawn
     * @param start The index of the first character in text to draw
     * @param end (end - 1) is the index of the last character in text to draw
     * @param x The x-coordinate of origin for where to draw the text
     * @param y The y-coordinate of origin for where to draw the text
     * @param paint The paint used for the text (e.g. color, size, style)
     */
    public void drawText(@NonNull CharSequence text, int start, int end, float x, float y,
            @NonNull Paint paint) {
        super.drawText(text, start, end, x, y, paint);
    }

    /**
     * Draw the text, with origin at (x,y), using the specified paint, along the specified path. The
     * paint's Align setting determins where along the path to start the text.
     *
     * @param text The text to be drawn
     * @param path The path the text should follow for its baseline
     * @param hOffset The distance along the path to add to the text's starting position
     * @param vOffset The distance above(-) or below(+) the path to position the text
     * @param paint The paint used for the text (e.g. color, size, style)
     */
    public void drawTextOnPath(@NonNull char[] text, int index, int count, @NonNull Path path,
            float hOffset, float vOffset, @NonNull Paint paint) {
        super.drawTextOnPath(text, index, count, path, hOffset, vOffset, paint);
    }

    /**
     * Draw the text, with origin at (x,y), using the specified paint, along the specified path. The
     * paint's Align setting determins where along the path to start the text.
     *
     * @param text The text to be drawn
     * @param path The path the text should follow for its baseline
     * @param hOffset The distance along the path to add to the text's starting position
     * @param vOffset The distance above(-) or below(+) the path to position the text
     * @param paint The paint used for the text (e.g. color, size, style)
     */
    public void drawTextOnPath(@NonNull String text, @NonNull Path path, float hOffset,
            float vOffset, @NonNull Paint paint) {
        super.drawTextOnPath(text, path, hOffset, vOffset, paint);
    }

    /**
     * Draw a run of text, all in a single direction, with optional context for complex text
     * shaping.
```

```
     * <p>
     * See {@link #drawTextRun(CharSequence, int, int, int, int, float, float, boolean, Paint)} for
     * more details. This method uses a character array rather than CharSequence to represent the
     * string. Also, to be consistent with the pattern established in {@link #drawText}, in this
     * method {@code count} and {@code contextCount} are used rather than offsets of the end
     * position; {@code count = end - start, contextCount = contextEnd -
     * contextStart}.
     *
     * @param text the text to render
     * @param index the start of the text to render
     * @param count the count of chars to render
     * @param contextIndex the start of the context for shaping. Must be no greater than index.
     * @param contextCount the number of characters in the context for shaping. contexIndex +
     *            contextCount must be no less than index + count.
     * @param x the x position at which to draw the text
     * @param y the y position at which to draw the text
     * @param isRtl whether the run is in RTL direction
     * @param paint the paint
     */
    public void drawTextRun(@NonNull char[] text, int index, int count, int contextIndex,
            int contextCount, float x, float y, boolean isRtl, @NonNull Paint paint) {
        super.drawTextRun(text, index, count, contextIndex, contextCount, x, y, isRtl, paint);
    }

    /**
     * Draw a run of text, all in a single direction, with optional context for complex text
     * shaping.
     * <p>
     * The run of text includes the characters from {@code start} to {@code end} in the text. In
     * addition, the range {@code contextStart} to {@code contextEnd} is used as context for the
     * purpose of complex text shaping, such as Arabic text potentially shaped differently based on
     * the text next to it.
     * <p>
     * All text outside the range {@code contextStart..contextEnd} is ignored. The text between
     * {@code start} and {@code end} will be laid out and drawn.
     * <p>
     * The direction of the run is explicitly specified by {@code isRtl}. Thus, this method is
     * suitable only for runs of a single direction. Alignment of the text is as determined by the
     * Paint's TextAlign value. Further, {@code 0 <= contextStart <= start <= end <= contextEnd
     * <= text.length} must hold on entry.
     * <p>
     * Also see {@link android.graphics.Paint#getRunAdvance} for a corresponding method to measure
     * the text; the advance width of the text drawn matches the value obtained from that method.
     *
     * @param text the text to render
     * @param start the start of the text to render. Data before this position can be used for
     *            shaping context.
     * @param end the end of the text to render. Data at or after this position can be used for
     *            shaping context.
     * @param contextStart the index of the start of the shaping context
     * @param contextEnd the index of the end of the shaping context
     * @param x the x position at which to draw the text
     * @param y the y position at which to draw the text
     * @param isRtl whether the run is in RTL direction
     * @param paint the paint
     * @see #drawTextRun(char[], int, int, int, int, float, float, boolean, Paint)
     */
    public void drawTextRun(@NonNull CharSequence text, int start, int end, int contextStart,
            int contextEnd, float x, float y, boolean isRtl, @NonNull Paint paint) {
        super.drawTextRun(text, start, end, contextStart, contextEnd, x, y, isRtl, paint);
    }

    /**
     * Draw the array of vertices, interpreted as triangles (based on mode). The verts array is
     * required, and specifies the x,y pairs for each vertex. If texs is non-null, then it is used
     * to specify the coordinate in shader coordinates to use at each vertex (the paint must have a
     * shader in this case). If there is no texs array, but there is a color array, then each color
     * is interpolated across its corresponding triangle in a gradient. If both texs and colors
     * arrays are present, then they behave as before, but the resulting color at each pixels is the
     * result of multiplying the colors from the shader and the color-gradient together. The indices
```

```
     * array is optional, but if it is present, then it is used to specify the index of each
     * triangle, rather than just walking through the arrays in order.
     *
     * @param mode How to interpret the array of vertices
     * @param vertexCount The number of values in the vertices array (and corresponding texs and
     *            colors arrays if non-null). Each logical vertex is two values (x, y), vertexCount
     *            must be a multiple of 2.
     * @param verts Array of vertices for the mesh
     * @param vertOffset Number of values in the verts to skip before drawing.
     * @param texs May be null. If not null, specifies the coordinates to sample into the current
     *            shader (e.g. bitmap tile or gradient)
     * @param texOffset Number of values in texs to skip before drawing.
     * @param colors May be null. If not null, specifies a color for each vertex, to be interpolated
     *            across the triangle.
     * @param colorOffset Number of values in colors to skip before drawing.
     * @param indices If not null, array of indices to reference into the vertex (texs, colors)
     *            array.
     * @param indexCount number of entries in the indices array (if not null).
     * @param paint Specifies the shader to use if the texs array is non-null.
     */
    public void drawVertices(@NonNull VertexMode mode, int vertexCount, @NonNull float[] verts,
            int vertOffset, @Nullable float[] texs, int texOffset, @Nullable int[] colors,
            int colorOffset, @Nullable short[] indices, int indexOffset, int indexCount,
            @NonNull Paint paint) {
        super.drawVertices(mode, vertexCount, verts, vertOffset, texs, texOffset,
                colors, colorOffset, indices, indexOffset, indexCount, paint);
    }
}
```