<declare-styleable name="CircleImageView">  
 <attr name="civ\_border\_width" format="dimension" />  
 <attr name="civ\_border\_color" format="color" />  
 <attr name="civ\_border\_overlay" format="boolean" />  
 <attr name="civ\_fill\_color" format="color" />  
</declare-styleable>

/\*  
 \* Copyright 2014 - 2017 Henning Dodenhof  
 \*  
 \* 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 com.xinli.vkeeper.views;  
  
import android.content.Context;  
import android.content.res.TypedArray;  
import android.graphics.Bitmap;  
import android.graphics.BitmapShader;  
import android.graphics.Canvas;  
import android.graphics.Color;  
import android.graphics.ColorFilter;  
import android.graphics.Matrix;  
import android.graphics.Paint;  
import android.graphics.RectF;  
import android.graphics.Shader;  
import android.graphics.drawable.BitmapDrawable;  
import android.graphics.drawable.ColorDrawable;  
import android.graphics.drawable.Drawable;  
import android.net.Uri;  
import android.support.annotation.ColorInt;  
import android.support.annotation.ColorRes;  
import android.support.annotation.DrawableRes;  
import android.util.AttributeSet;  
import android.widget.ImageView;  
  
import com.xinli.vkeeper.R;  
  
  
public class CircleImageView extends ImageView {  
  
 private static final ScaleType *SCALE\_TYPE* = ScaleType.*CENTER\_CROP*;  
  
 private static final Bitmap.Config *BITMAP\_CONFIG* = Bitmap.Config.*ARGB\_8888*;  
 private static final int *COLORDRAWABLE\_DIMENSION* = 2;  
  
 private static final int *DEFAULT\_BORDER\_WIDTH* = 0;  
 private static final int *DEFAULT\_BORDER\_COLOR* = Color.*BLACK*;  
 private static final int *DEFAULT\_FILL\_COLOR* = Color.*TRANSPARENT*;  
 private static final boolean *DEFAULT\_BORDER\_OVERLAY* = false;  
  
 private final RectF mDrawableRect = new RectF();  
 private final RectF mBorderRect = new RectF();  
  
 private final Matrix mShaderMatrix = new Matrix();  
 private final Paint mBitmapPaint = new Paint();  
 private final Paint mBorderPaint = new Paint();  
 private final Paint mFillPaint = new Paint();  
  
 private int mBorderColor = *DEFAULT\_BORDER\_COLOR*;  
 private int mBorderWidth = *DEFAULT\_BORDER\_WIDTH*;  
 private int mFillColor = *DEFAULT\_FILL\_COLOR*;  
  
 private Bitmap mBitmap;  
 private BitmapShader mBitmapShader;  
 private int mBitmapWidth;  
 private int mBitmapHeight;  
  
 private float mDrawableRadius;  
 private float mBorderRadius;  
  
 private ColorFilter mColorFilter;  
  
 private boolean mReady;  
 private boolean mSetupPending;  
 private boolean mBorderOverlay;  
 private boolean mDisableCircularTransformation;  
  
 public CircleImageView(Context context) {  
 super(context);  
  
 init();  
 }  
  
 public CircleImageView(Context context, AttributeSet attrs) {  
 this(context, attrs, 0);  
 }  
  
 public CircleImageView(Context context, AttributeSet attrs, int defStyle) {  
 super(context, attrs, defStyle);  
  
 TypedArray a = context.obtainStyledAttributes(attrs, R.styleable.CircleImageView, defStyle, 0);  
  
 mBorderWidth = a.getDimensionPixelSize(R.styleable.*CircleImageView\_civ\_border\_width*, *DEFAULT\_BORDER\_WIDTH*);  
 mBorderColor = a.getColor(R.styleable.*CircleImageView\_civ\_border\_color*, *DEFAULT\_BORDER\_COLOR*);  
 mBorderOverlay = a.getBoolean(R.styleable.*CircleImageView\_civ\_border\_overlay*, *DEFAULT\_BORDER\_OVERLAY*);  
 mFillColor = a.getColor(R.styleable.*CircleImageView\_civ\_fill\_color*, *DEFAULT\_FILL\_COLOR*);  
  
 a.recycle();  
  
 init();  
 }  
  
 private void init() {  
 super.setScaleType(*SCALE\_TYPE*);  
 mReady = true;  
  
 if (mSetupPending) {  
 setup();  
 mSetupPending = false;  
 }  
 }  
  
 @Override  
 public ScaleType getScaleType() {  
 return *SCALE\_TYPE*;  
 }  
  
 @Override  
 public void setScaleType(ScaleType scaleType) {  
 if (scaleType != *SCALE\_TYPE*) {  
 throw new IllegalArgumentException(String.*format*("ScaleType %s not supported.", scaleType));  
 }  
 }  
  
 @Override  
 public void setAdjustViewBounds(boolean adjustViewBounds) {  
 if (adjustViewBounds) {  
 throw new IllegalArgumentException("adjustViewBounds not supported.");  
 }  
 }  
  
 @Override  
 protected void onDraw(Canvas canvas) {  
 if (mDisableCircularTransformation) {  
 super.onDraw(canvas);  
 return;  
 }  
  
 if (mBitmap == null) {  
 return;  
 }  
  
 if (mFillColor != Color.*TRANSPARENT*) {  
 canvas.drawCircle(mDrawableRect.centerX(), mDrawableRect.centerY(), mDrawableRadius, mFillPaint);  
 }  
 canvas.drawCircle(mDrawableRect.centerX(), mDrawableRect.centerY(), mDrawableRadius, mBitmapPaint);  
 if (mBorderWidth > 0) {  
 canvas.drawCircle(mBorderRect.centerX(), mBorderRect.centerY(), mBorderRadius, mBorderPaint);  
 }  
 }  
  
 @Override  
 protected void onSizeChanged(int w, int h, int oldw, int oldh) {  
 super.onSizeChanged(w, h, oldw, oldh);  
 setup();  
 }  
  
 @Override  
 public void setPadding(int left, int top, int right, int bottom) {  
 super.setPadding(left, top, right, bottom);  
 setup();  
 }  
  
 @Override  
 public void setPaddingRelative(int start, int top, int end, int bottom) {  
 super.setPaddingRelative(start, top, end, bottom);  
 setup();  
 }  
  
 public int getBorderColor() {  
 return mBorderColor;  
 }  
  
 public void setBorderColor(@ColorInt int borderColor) {  
 if (borderColor == mBorderColor) {  
 return;  
 }  
  
 mBorderColor = borderColor;  
 mBorderPaint.setColor(mBorderColor);  
 invalidate();  
 }  
  
 */\*\*  
 \** ***@deprecated*** *Use {****@link*** *#setBorderColor(int)} instead  
 \*/* @Deprecated  
 public void setBorderColorResource(@ColorRes int borderColorRes) {  
 setBorderColor(getContext().getResources().getColor(borderColorRes));  
 }  
  
 */\*\*  
 \* Return the color drawn behind the circle-shaped drawable.  
 \*  
 \** ***@return*** *The color drawn behind the drawable  
 \*  
 \** ***@deprecated*** *Fill color support is going to be removed in the future  
 \*/* @Deprecated  
 public int getFillColor() {  
 return mFillColor;  
 }  
  
 */\*\*  
 \* Set a color to be drawn behind the circle-shaped drawable. Note that  
 \* this has no effect if the drawable is opaque or no drawable is set.  
 \*  
 \** ***@param*** *fillColor The color to be drawn behind the drawable  
 \*  
 \** ***@deprecated*** *Fill color support is going to be removed in the future  
 \*/* @Deprecated  
 public void setFillColor(@ColorInt int fillColor) {  
 if (fillColor == mFillColor) {  
 return;  
 }  
  
 mFillColor = fillColor;  
 mFillPaint.setColor(fillColor);  
 invalidate();  
 }  
  
 */\*\*  
 \* Set a color to be drawn behind the circle-shaped drawable. Note that  
 \* this has no effect if the drawable is opaque or no drawable is set.  
 \*  
 \** ***@param*** *fillColorRes The color resource to be resolved to a color and  
 \* drawn behind the drawable  
 \*  
 \** ***@deprecated*** *Fill color support is going to be removed in the future  
 \*/* @Deprecated  
 public void setFillColorResource(@ColorRes int fillColorRes) {  
 setFillColor(getContext().getResources().getColor(fillColorRes));  
 }  
  
 public int getBorderWidth() {  
 return mBorderWidth;  
 }  
  
 public void setBorderWidth(int borderWidth) {  
 if (borderWidth == mBorderWidth) {  
 return;  
 }  
  
 mBorderWidth = borderWidth;  
 setup();  
 }  
  
 public boolean isBorderOverlay() {  
 return mBorderOverlay;  
 }  
  
 public void setBorderOverlay(boolean borderOverlay) {  
 if (borderOverlay == mBorderOverlay) {  
 return;  
 }  
  
 mBorderOverlay = borderOverlay;  
 setup();  
 }  
  
 public boolean isDisableCircularTransformation() {  
 return mDisableCircularTransformation;  
 }  
  
 public void setDisableCircularTransformation(boolean disableCircularTransformation) {  
 if (mDisableCircularTransformation == disableCircularTransformation) {  
 return;  
 }  
  
 mDisableCircularTransformation = disableCircularTransformation;  
 initializeBitmap();  
 }  
  
 @Override  
 public void setImageBitmap(Bitmap bm) {  
 super.setImageBitmap(bm);  
 initializeBitmap();  
 }  
  
 @Override  
 public void setImageDrawable(Drawable drawable) {  
 super.setImageDrawable(drawable);  
 initializeBitmap();  
 }  
  
 @Override  
 public void setImageResource(@DrawableRes int resId) {  
 super.setImageResource(resId);  
 initializeBitmap();  
 }  
  
 @Override  
 public void setImageURI(Uri uri) {  
 super.setImageURI(uri);  
 initializeBitmap();  
 }  
  
 @Override  
 public void setColorFilter(ColorFilter cf) {  
 if (cf == mColorFilter) {  
 return;  
 }  
  
 mColorFilter = cf;  
 applyColorFilter();  
 invalidate();  
 }  
  
 @Override  
 public ColorFilter getColorFilter() {  
 return mColorFilter;  
 }  
  
 private void applyColorFilter() {  
 if (mBitmapPaint != null) {  
 mBitmapPaint.setColorFilter(mColorFilter);  
 }  
 }  
  
 private Bitmap getBitmapFromDrawable(Drawable drawable) {  
 if (drawable == null) {  
 return null;  
 }  
  
 if (drawable instanceof BitmapDrawable) {  
 return ((BitmapDrawable) drawable).getBitmap();  
 }  
  
 try {  
 Bitmap bitmap;  
  
 if (drawable instanceof ColorDrawable) {  
 bitmap = Bitmap.*createBitmap*(*COLORDRAWABLE\_DIMENSION*, *COLORDRAWABLE\_DIMENSION*, *BITMAP\_CONFIG*);  
 } else {  
 bitmap = Bitmap.*createBitmap*(drawable.getIntrinsicWidth(), drawable.getIntrinsicHeight(), *BITMAP\_CONFIG*);  
 }  
  
 Canvas canvas = new Canvas(bitmap);  
 drawable.setBounds(0, 0, canvas.getWidth(), canvas.getHeight());  
 drawable.draw(canvas);  
 return bitmap;  
 } catch (Exception e) {  
 e.printStackTrace();  
 return null;  
 }  
 }  
  
 private void initializeBitmap() {  
 if (mDisableCircularTransformation) {  
 mBitmap = null;  
 } else {  
 mBitmap = getBitmapFromDrawable(getDrawable());  
 }  
 setup();  
 }  
  
 private void setup() {  
 if (!mReady) {  
 mSetupPending = true;  
 return;  
 }  
  
 if (getWidth() == 0 && getHeight() == 0) {  
 return;  
 }  
  
 if (mBitmap == null) {  
 invalidate();  
 return;  
 }  
  
 mBitmapShader = new BitmapShader(mBitmap, Shader.TileMode.*CLAMP*, Shader.TileMode.*CLAMP*);  
  
 mBitmapPaint.setAntiAlias(true);  
 mBitmapPaint.setShader(mBitmapShader);  
  
 mBorderPaint.setStyle(Paint.Style.*STROKE*);  
 mBorderPaint.setAntiAlias(true);  
 mBorderPaint.setColor(mBorderColor);  
 mBorderPaint.setStrokeWidth(mBorderWidth);  
  
 mFillPaint.setStyle(Paint.Style.*FILL*);  
 mFillPaint.setAntiAlias(true);  
 mFillPaint.setColor(mFillColor);  
  
 mBitmapHeight = mBitmap.getHeight();  
 mBitmapWidth = mBitmap.getWidth();  
  
 mBorderRect.set(calculateBounds());  
 mBorderRadius = Math.*min*((mBorderRect.height() - mBorderWidth) / 2.0f, (mBorderRect.width() - mBorderWidth) / 2.0f);  
  
 mDrawableRect.set(mBorderRect);  
 if (!mBorderOverlay && mBorderWidth > 0) {  
 mDrawableRect.inset(mBorderWidth - 1.0f, mBorderWidth - 1.0f);  
 }  
 mDrawableRadius = Math.*min*(mDrawableRect.height() / 2.0f, mDrawableRect.width() / 2.0f);  
  
 applyColorFilter();  
 updateShaderMatrix();  
 invalidate();  
 }  
  
 private RectF calculateBounds() {  
 int availableWidth = getWidth() - getPaddingLeft() - getPaddingRight();  
 int availableHeight = getHeight() - getPaddingTop() - getPaddingBottom();  
  
 int sideLength = Math.*min*(availableWidth, availableHeight);  
  
 float left = getPaddingLeft() + (availableWidth - sideLength) / 2f;  
 float top = getPaddingTop() + (availableHeight - sideLength) / 2f;  
  
 return new RectF(left, top, left + sideLength, top + sideLength);  
 }  
  
 private void updateShaderMatrix() {  
 float scale;  
 float dx = 0;  
 float dy = 0;  
  
 mShaderMatrix.set(null);  
  
 if (mBitmapWidth \* mDrawableRect.height() > mDrawableRect.width() \* mBitmapHeight) {  
 scale = mDrawableRect.height() / (float) mBitmapHeight;  
 dx = (mDrawableRect.width() - mBitmapWidth \* scale) \* 0.5f;  
 } else {  
 scale = mDrawableRect.width() / (float) mBitmapWidth;  
 dy = (mDrawableRect.height() - mBitmapHeight \* scale) \* 0.5f;  
 }  
  
 mShaderMatrix.setScale(scale, scale);  
 mShaderMatrix.postTranslate((int) (dx + 0.5f) + mDrawableRect.left, (int) (dy + 0.5f) + mDrawableRect.top);  
  
 mBitmapShader.setLocalMatrix(mShaderMatrix);  
 }  
  
}