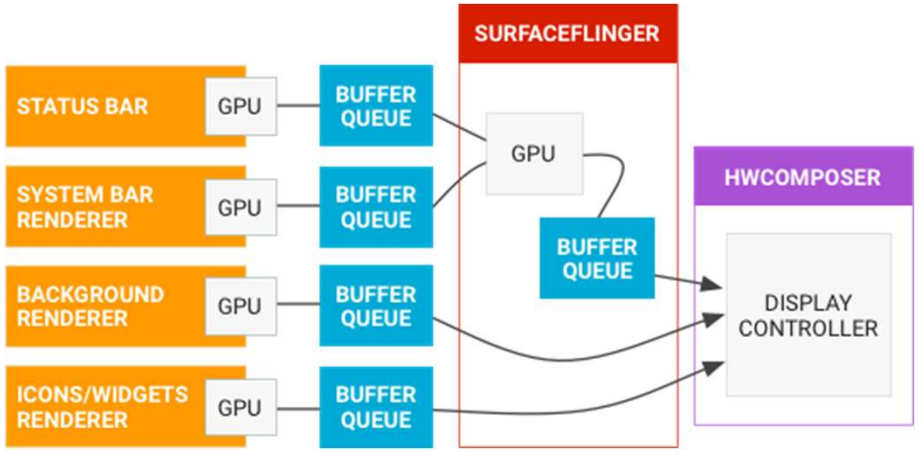
2D GRAPHICS ANDROID

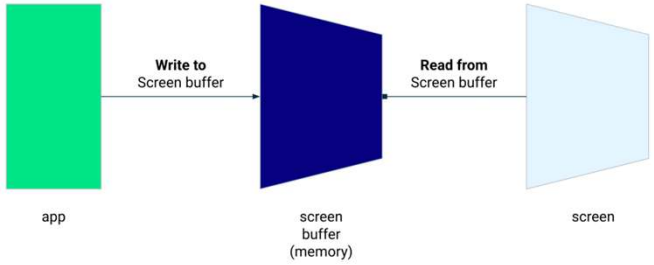
DRAWING

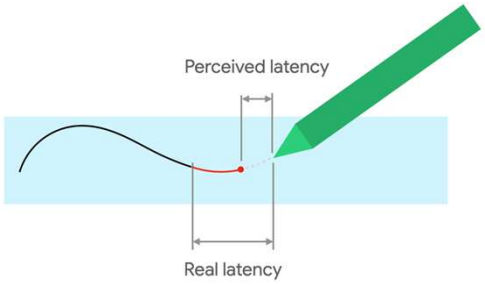
- The GPU (Graphics Processing Unit) is specialized hardware designed to manage and render images, animations, and videos for display on a screen.
- The objects on the left are renderers producing graphics buffers, such as the home screen, status bar, and system UI. SurfaceFlinger is a sw compositor and Hardware Composer is hw.

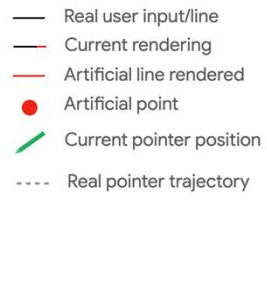


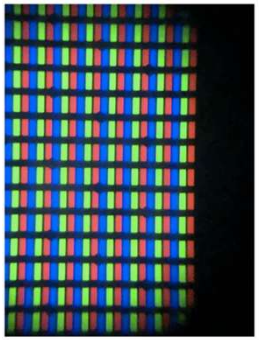
DRAWING

- An app can directly produce content for the screen writing directly on a screen buffer, which is the closest apps can get to drawing directly to the screen.
- Direct drawing reduces the latency between user input and screen output which is useful for time critical applications (like pencil)



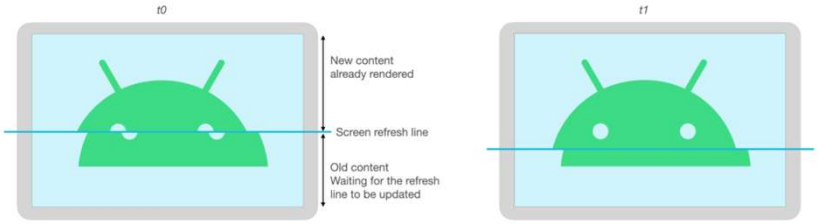


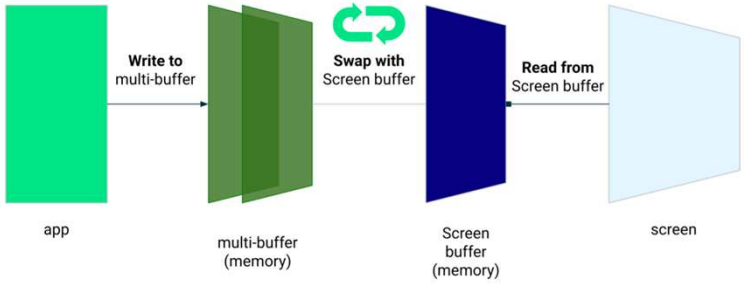




DRAWING

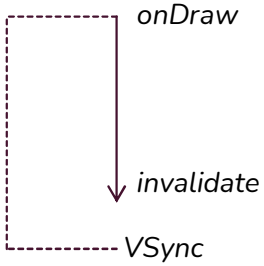
- However, direct drawing on the screen buffer may result in tearing, which happens when the screen refreshes while the screen buffer is being modified at the same time.
- A portion of the screen shows new data, while another shows old data.
- To avoid tearing multi buffers and a synch hw signal is used (vertical signal, or VSync)
- When a new Vsync signal is generated, the app (via render thread), sends drawing commands to the GPU that populates a new buffer (back buffer)
- At the next VSync boundary the back buffer becomes the video ram (front buffer)
- Supplementary buffers can be used to cache content

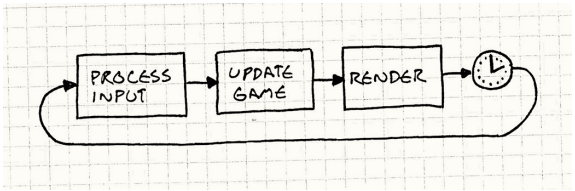


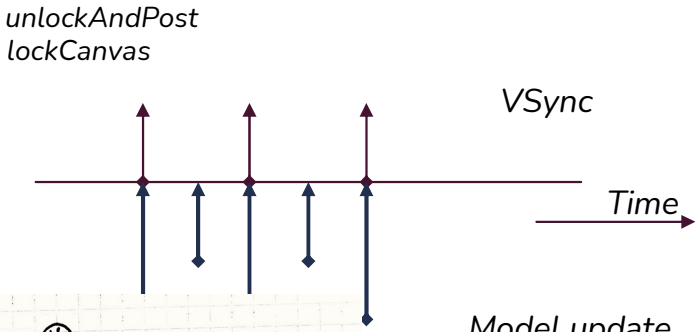


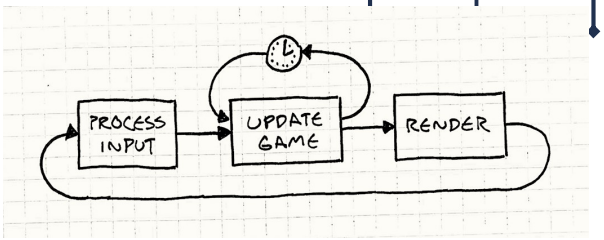
ANIMATION STRATEGIES

- Consider a time varying image like in a video game
- In VSync based animation the game update rate is tied to the frame rate. This is usually good for no complex graphics, where the update logic runs in the UI thread
- In time step animation the game state can be updated at a (usually) higher rate, and this reduces integration errors. The update logic runs on a separate thread
- Synchronization between the two threads are needed





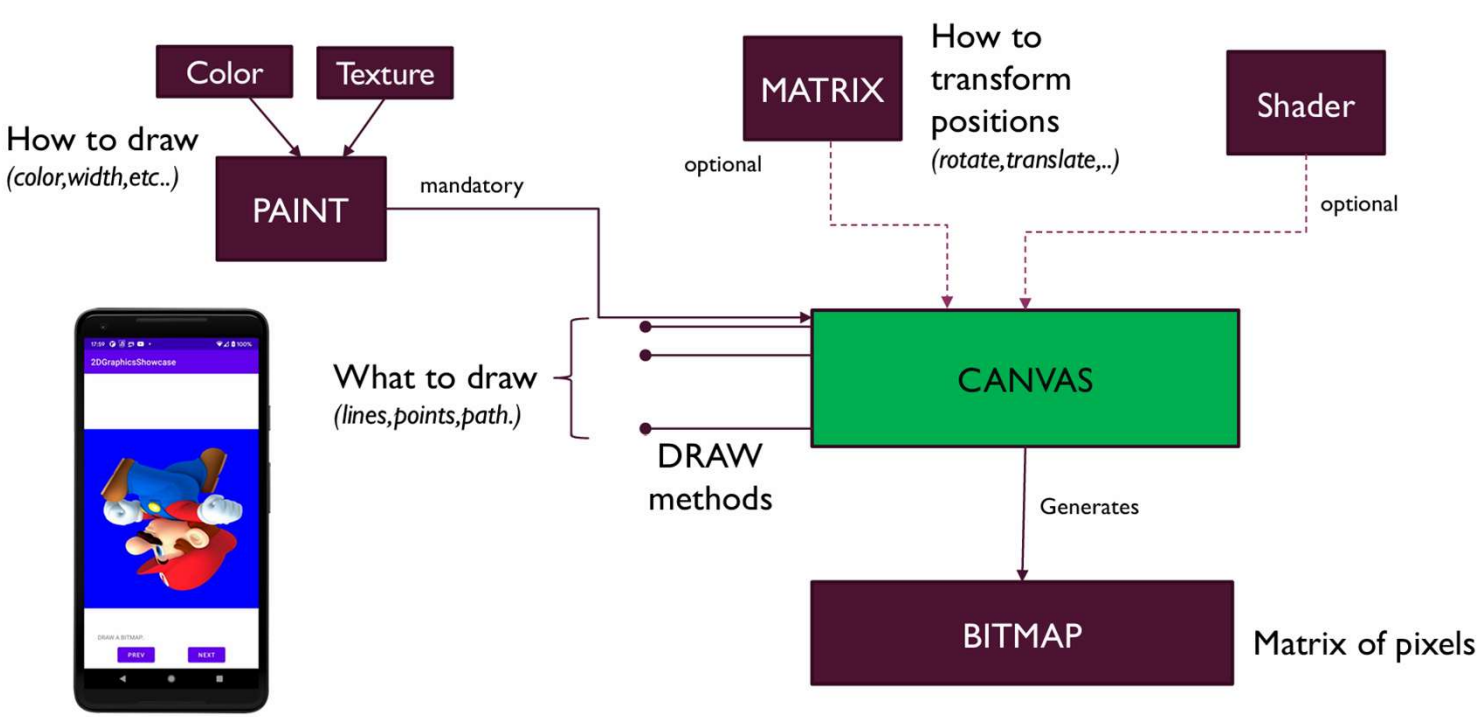


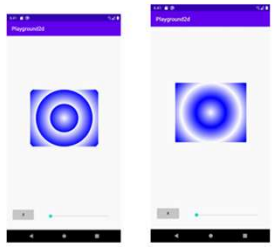


IMPLEMENTATION STRATEGIES

- Depending on the need, there are different alternatives
- Jetpack compose function (via modifier)
- Low-latency graphics library (androidx),
- Extending the View class: rendering occurs ‘on-screen’, which means that the rendering writes on the current back-buffer and the code runs on the UI thread
- Extending the SurfaceView class: : rendering occurs ‘off-screen’ and on a separate thread. The Choreographer class allows to register a callback method called at vsync boundaries
- OpenGL ES (best for 3D Graphis)

ANDROID.GRAPHICS.*





WHAT TO DO WITH 2D GRAPHICS

- While a camera produces "real" images, a 2D graphics library generates synthetic or artificial images.
- With a 2D library, images are created through drawing commands executed on a Canvas (e.g., drawing a line, a circle, etc.).
- The final output is a bitmap, that is, a matrix of pixels values.

WHAT TO DO WITH 2D GRAPHICS

- Drawing requires specifying the stroke width, expressed in pixels. The minimum drawable stroke is 1 px.
- To obtain consistent image sizes across devices, use device-independent pixels (dp).

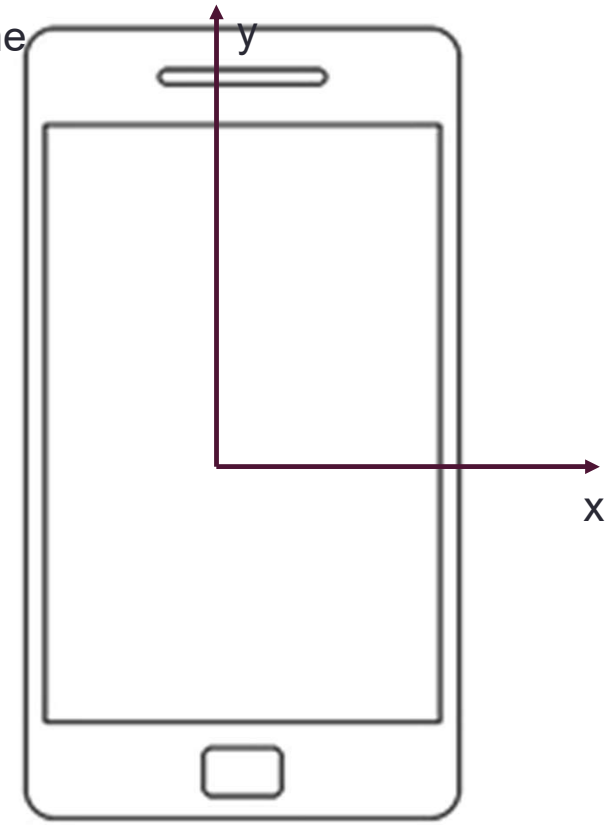
- Geometric primitives use floating-point coordinates
  - e.g., drawing a circle requires the center coordinates and the radius as floats.
- The coordinate system used for drawing corresponds to the screen reference frame.
- For simplicity, we will adopt a standard reference frame in our examples.
- Changing the coordinate basis is straightforward.

COORDINATE SYSTEMS USED FOR DRAWING

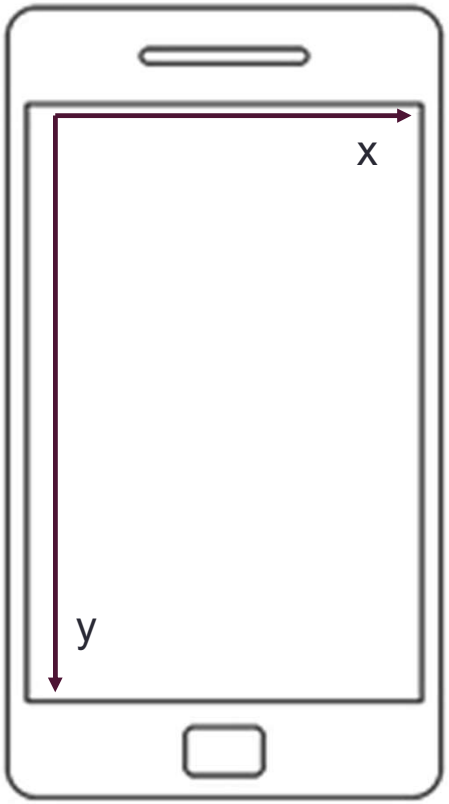
Device Reference Frame (DFR)

Can be normalize [-1,1]x[-1,1]

Right-Handed System (RHS)



Screen Reference Frame (SRF)



Left-Handed System LHS

CHANGE-OF-BASIS MATRIX (FROM DFR TO SRF)

$$ {}^{SRF}\mathrm{M}_{DRF} = \begin{bmatrix} X_{MAX}/2 & 0 & X_{MAX}/2 \\ 0 & -YMAX/2 & Y_{MAX}/2 \\ 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} X_{MAX}/2 & 0 & X_{MAX}/2 \\ 0 & -YMAX/2 & Y_{MAX}/2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} $$

For example

$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ is mapped to $\begin{bmatrix} X_{MAX}/2 \\ Y_{MAX}/2 \\ 1 \end{bmatrix}$

DRAWING PRIMITIVES

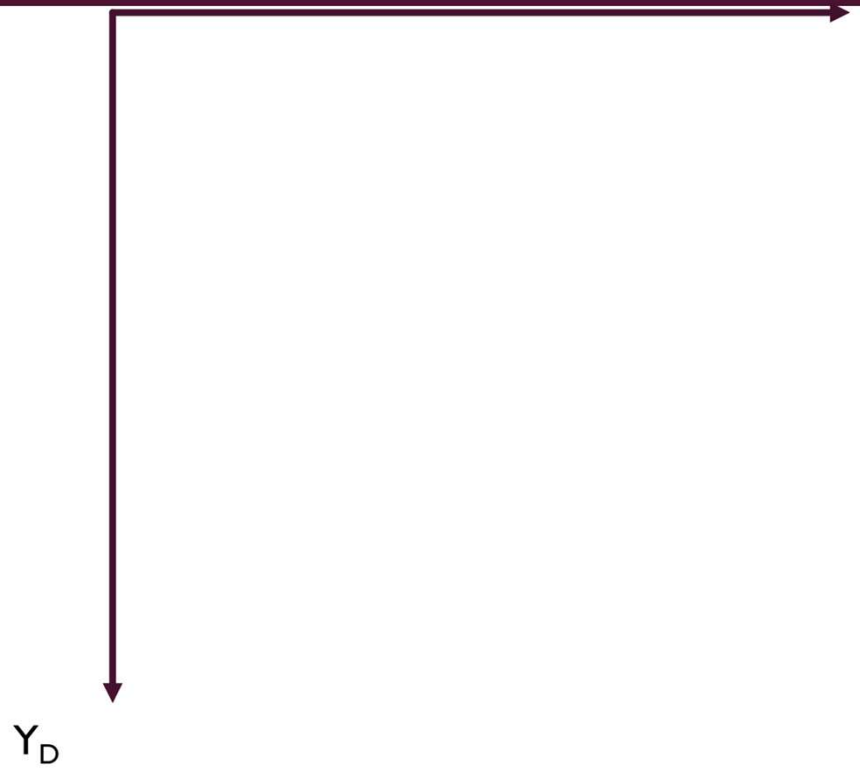
- Point (Points)
- Line (Lines)
- Path

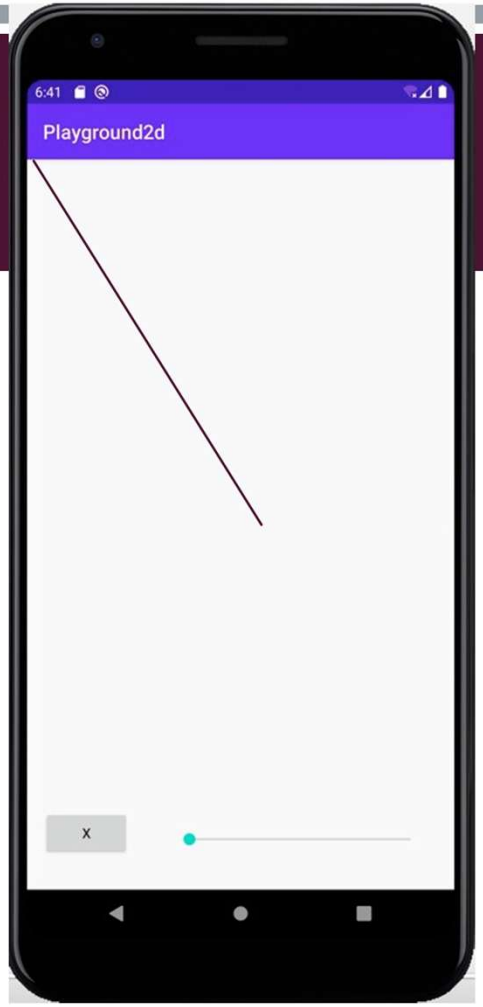
- Triangles
  - Connected triangle allows to approximate (almost) any shape

- Predefined shapes
  - Rect, Circle, Oval, Arc

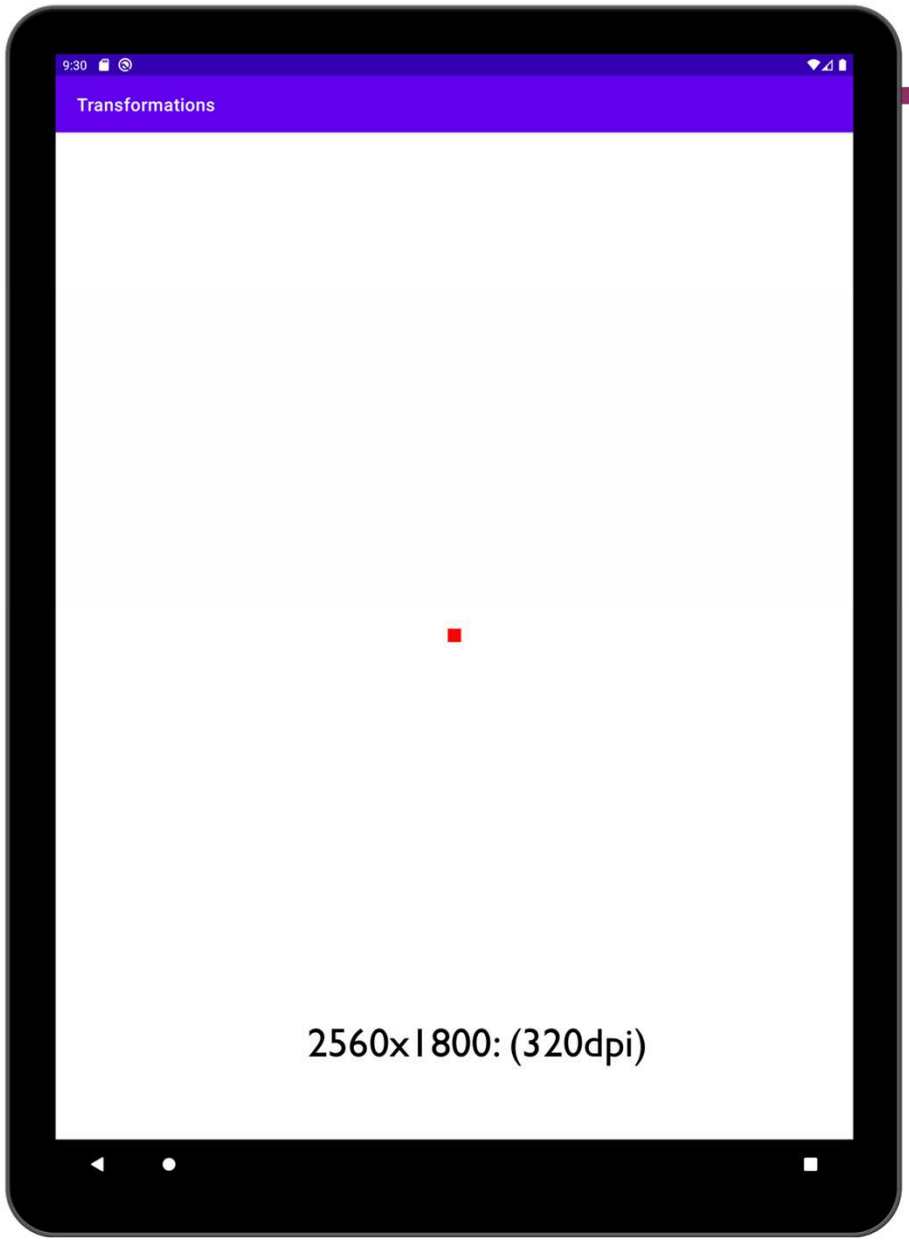
- Text

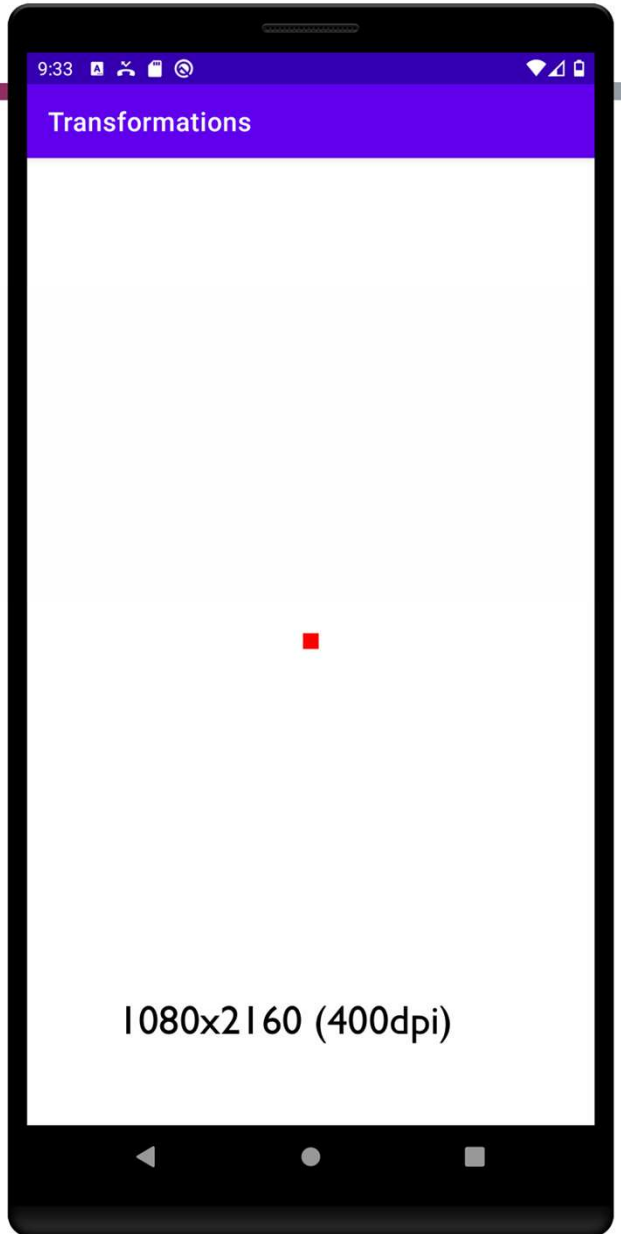
A SIMPLE EXAMPLE (DRAW A LINE)

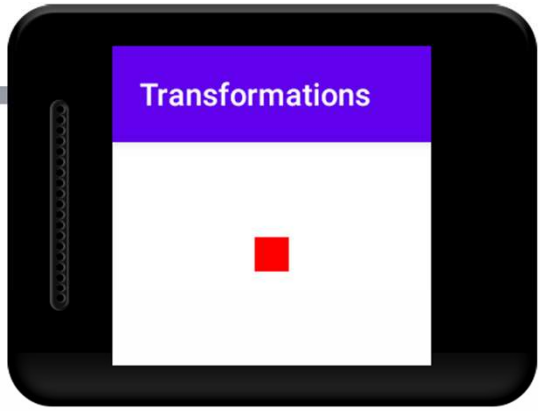





```
canvas.drawLine(0f,0f,canvas.width/2f,canvas.height/2f,mypaint)
```







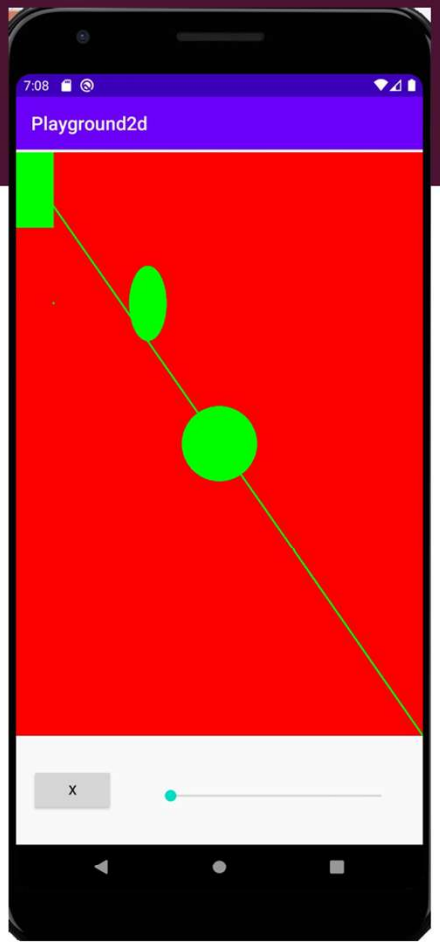
280x280 (240dpi)

```
val mPaint = Paint().apply {
style=Paint.Style.STROKE
color = Color.RED
strokeWidth=30f}
```

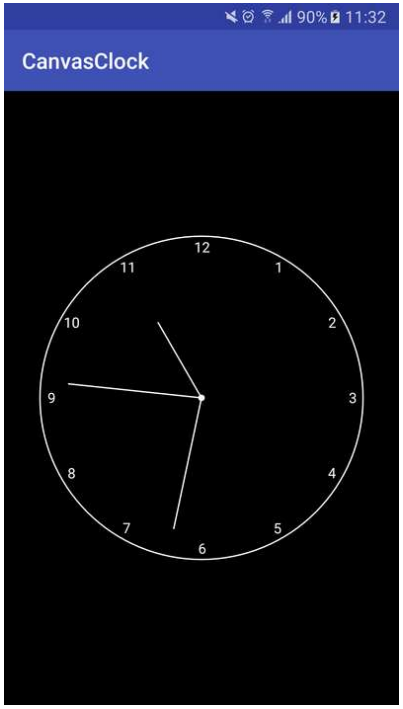
```
val cx = canvas.width/2f
val cy = canvas.height/2f
canvas.drawPoint(cx,cy,mPaint)
```

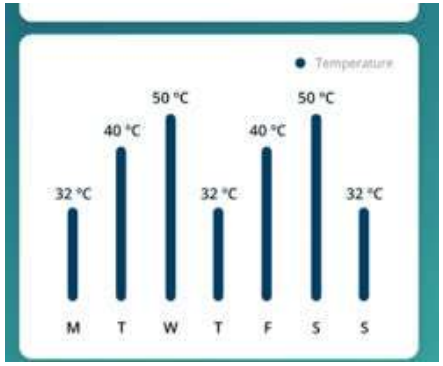
OTHER DRAWING EXAMPLES

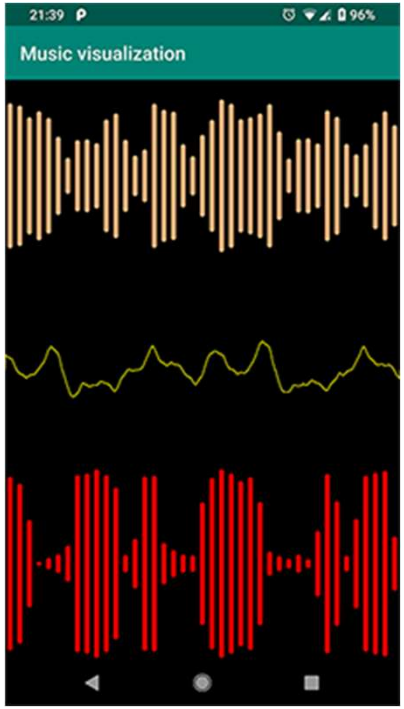
```
canvas?.drawCircle(w/2f,h/2f,100f,mypaint)
canvas?.drawOval(300f,300f,400f,500f,mypaint)
canvas?.drawRect(0f,0f,100f,200f,mypaint)
canvas?.drawLine(0f,0f,w.toFloat(),h.toFloat(),mypaint)
```



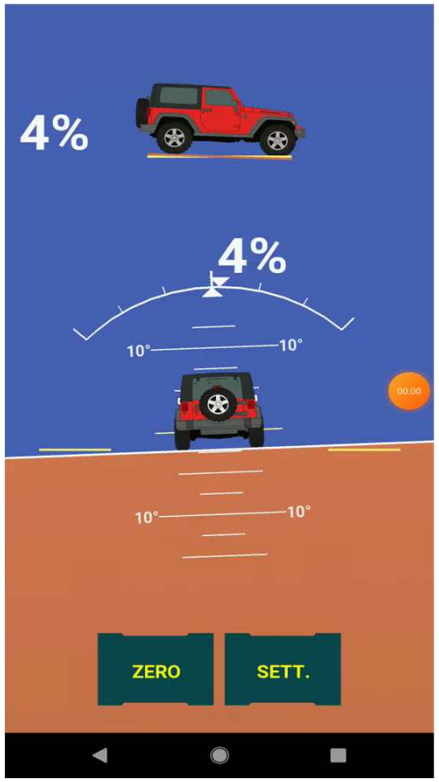
SOME EXAMPLES OF GRAPHICS USAGE

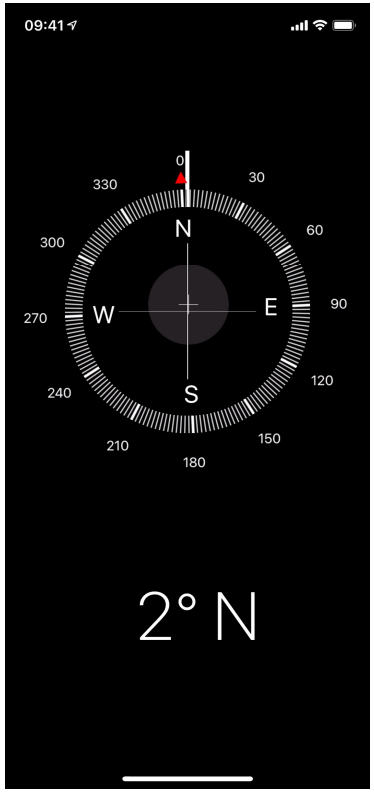


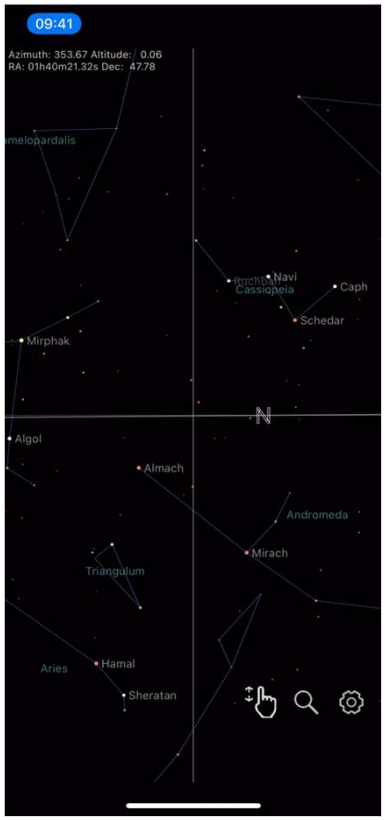




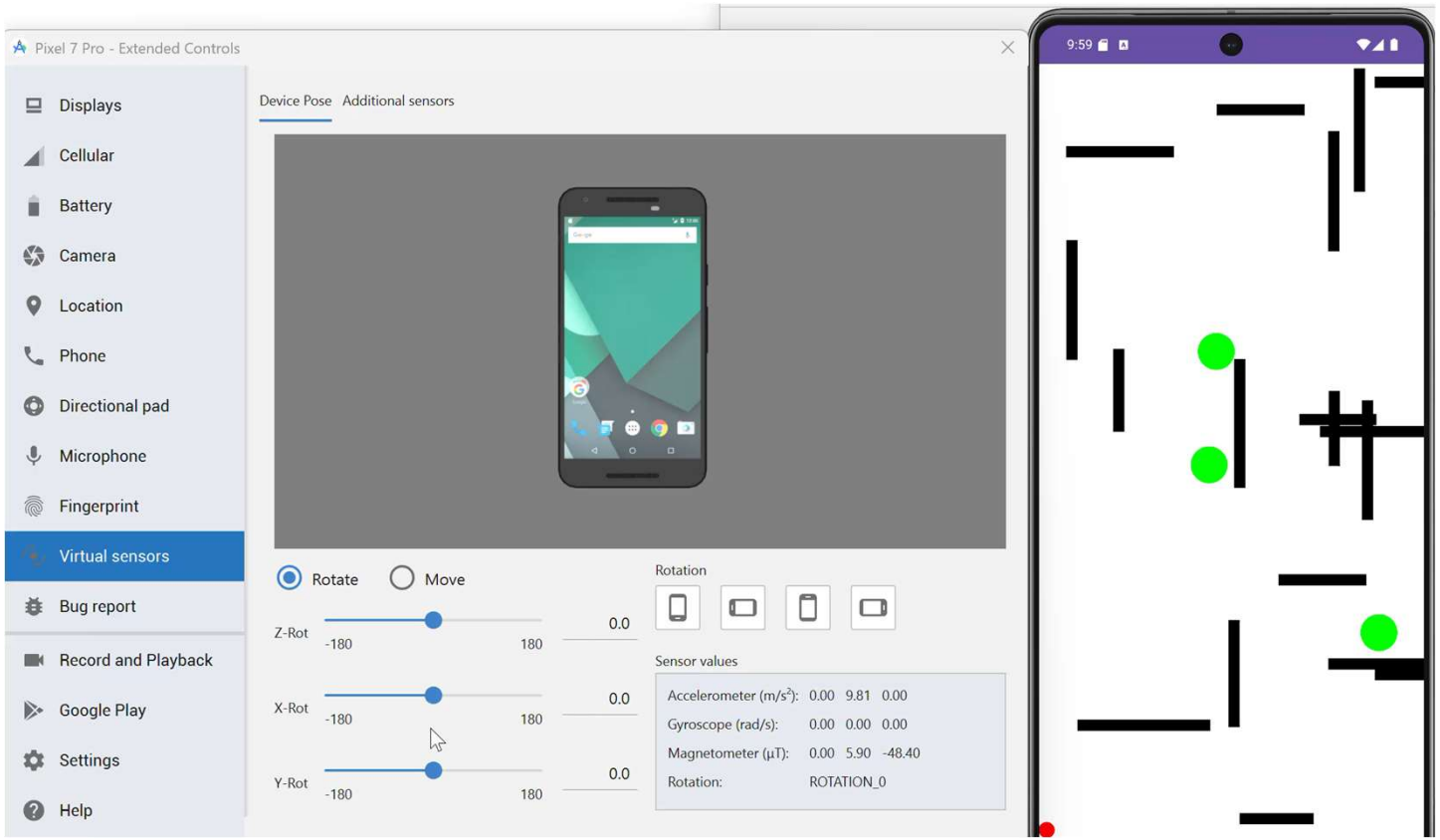
SOME EXAMPLES OF GRAPHICS USAGE







SOME EXAMPLES OF GRAPHICS USAGE

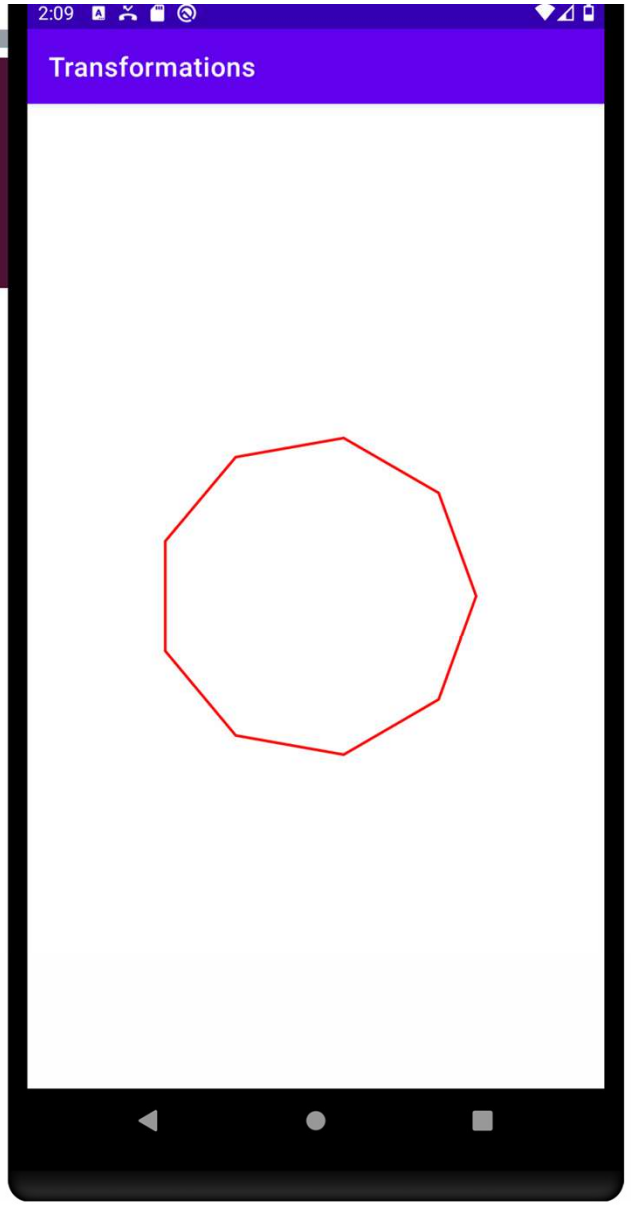


ALGORITHMICS IMAGES

- These are images generated purely by iterative computation
- Can be simple or quite complex like the Mandelbrot set

ALGORITHMICS IMAGES

```
val sides = 9
val rad = 300
val path = Path()
val angle = 2.0 * Math.PI / sides
path.moveTo(
        cx + (rad * Math.cos(0.0)).toFloat(),
        cy + (rad * Math.sin(0.0)).toFloat())
for (i in 1 until sides) {
    path.lineTo(
            cx + (rad * Math.cos(angle * i)).toFloat(),
            cy + (rad * Math.sin(angle * i)).toFloat())
}
path.close()
```

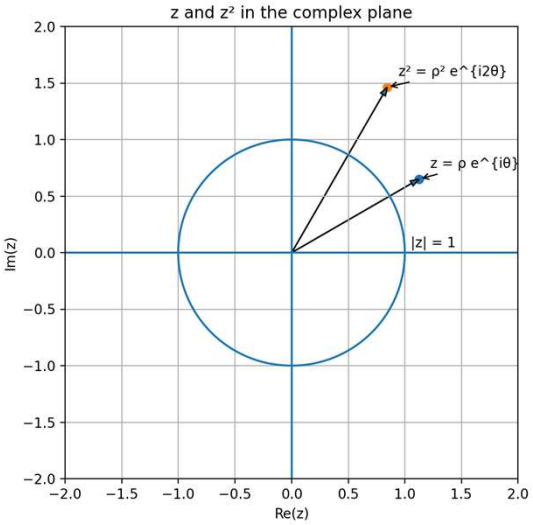


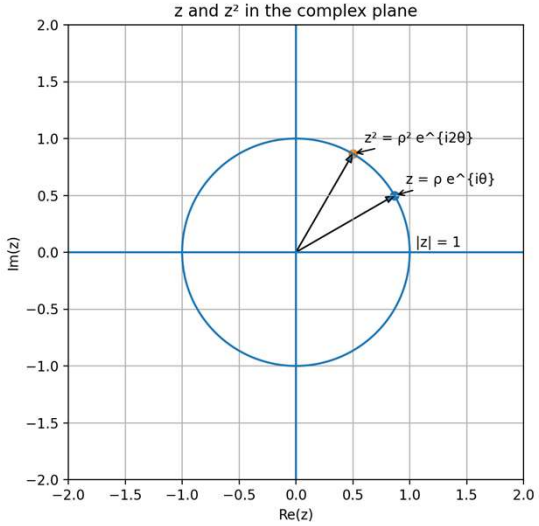
COMPLEX NUMBERS AND THE 2D SPACE

- A 2D point (x,y) can be interpreted as a complex number, **z** = x+iy
- In polar form, the same point is written as
- $\boldsymbol{z} = \rho e^{i\theta}$,
  - Where ρ= $\sqrt{x^2 + y^2}$and $\theta = \text{atan2}(y, x)$.
- Algebraic operations on complex numbers have a nice geometric interpretation.
  Roughly, additions correspond to translations, while multiplications correspond to rotations.

COMPLEX NUMBERS AND THE 2D SPACE

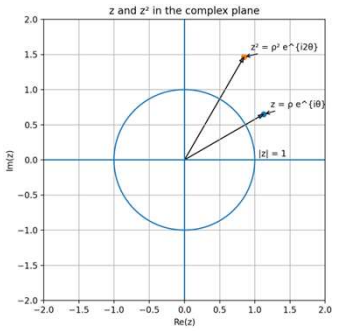
- A 2D point (x,y) can be interpreted as a complex number, $\mathbf{z} = x+iy$
- Some complex functions have a geometric interpretation
- For example, consider the function $f(\mathbf{z})=\mathbf{z}^2$
- In the polar for $\mathbf{z}=\rho e^{i\theta}$ and $\mathbf{z}^2=\rho^2 e^{i2\theta}$
- The function doubles the argument of $\mathbf{z}$
  - The argument of z is the angle of point z with the real axis
- squares its modulus

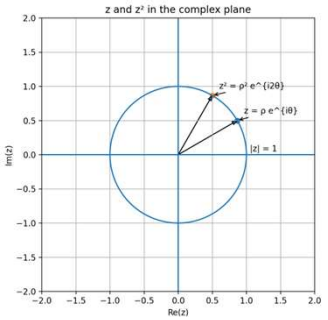


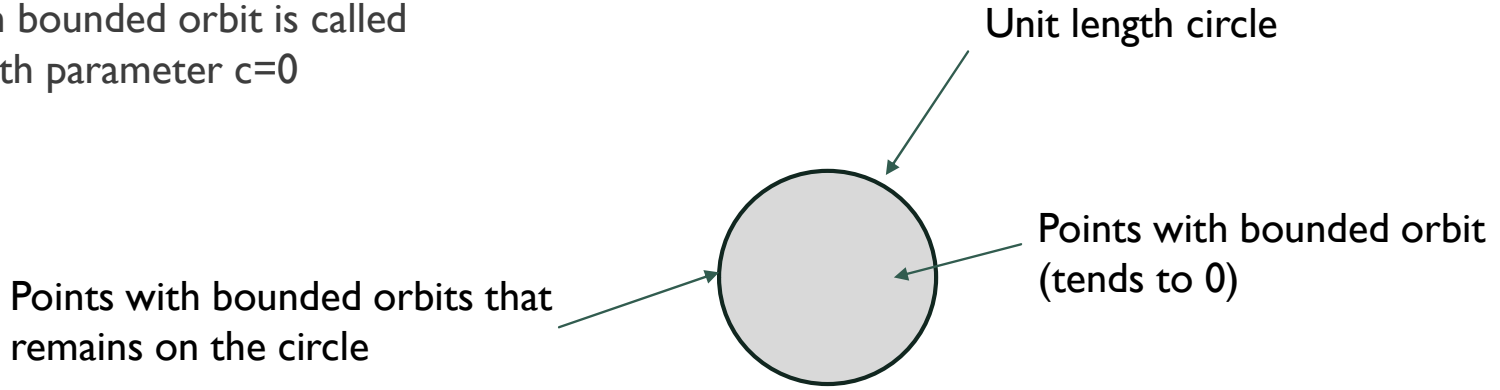


ORBITS

- What happens if the square function is applied iteratively?
- $\mathbf{z}, \mathbf{z}^2, \mathbf{z}^4, ..$
- The initial point **z** generates an orbit (or trajectory) in the 2D plane
- The orbit of **z**, may explode (if $|\mathbf{z}|>1$) or be bounded
- The set of points with bounded orbit is called the (filled) Julia Set with parameter c=0

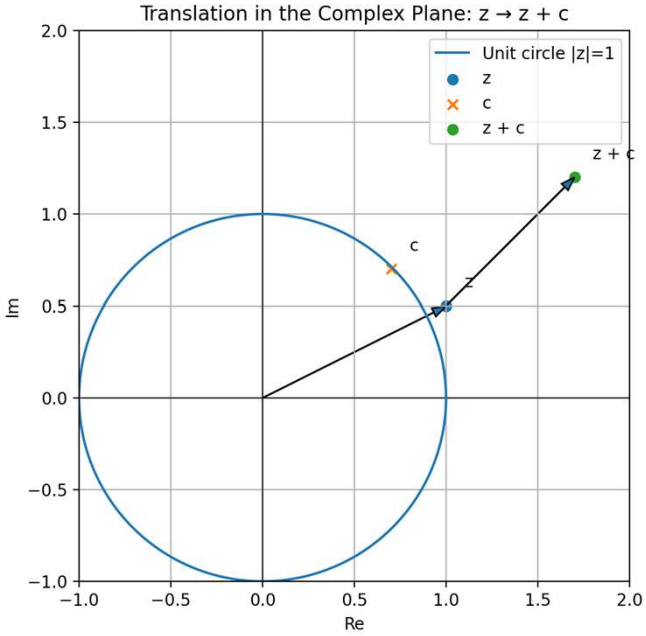






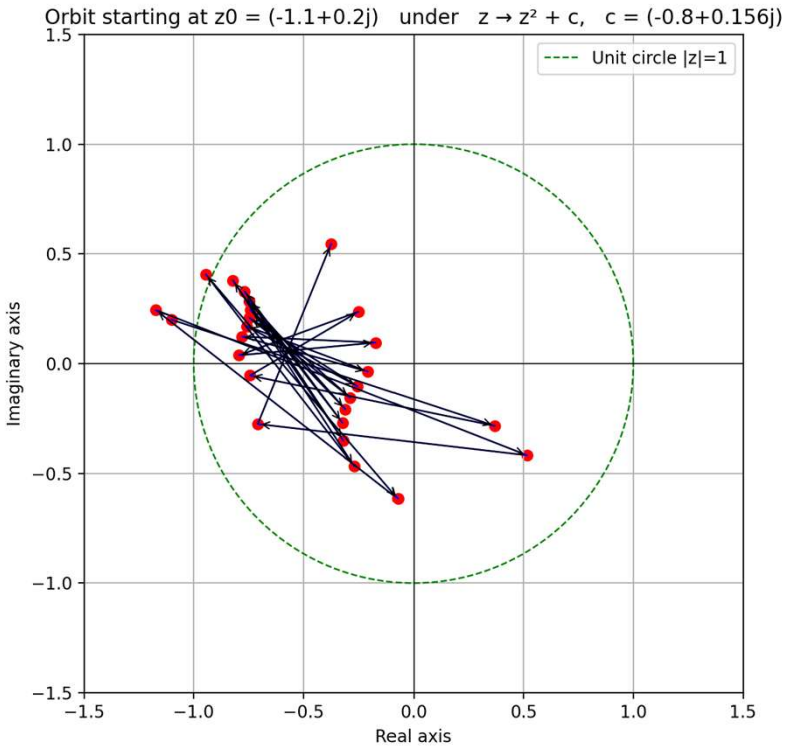
COMPLEX NUMBERS AND THE 2D SPACE

- The mapping $z \mapsto z + c$ shifts z by the vector c.



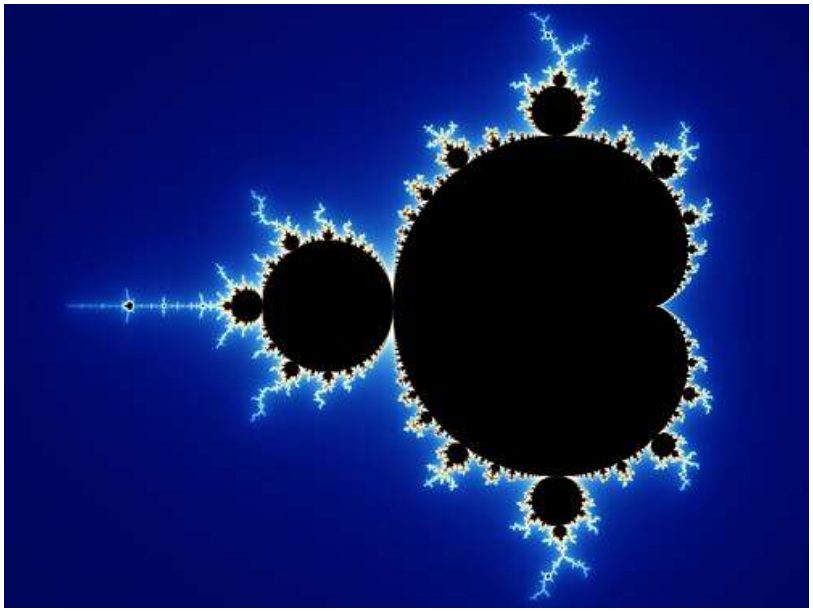
TRAJECTORY OF Z^2+C

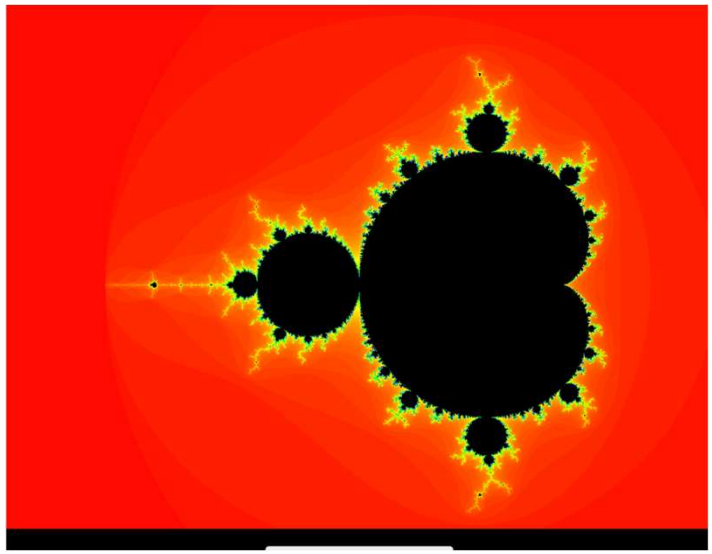
- The orbit of the point **z** under the quadratic mapping $\mathbf{z}^2+\mathbf{c}$ depends on **c**
- Because c is a translation, even points outside the unit circle can have a bounded orbit
- Julia set associated with *c* : set the points whose orbits remain bounded
- Mandelbrot set : set of points **c** (the translation amount) for which the orbit of **z**=0 is bounded



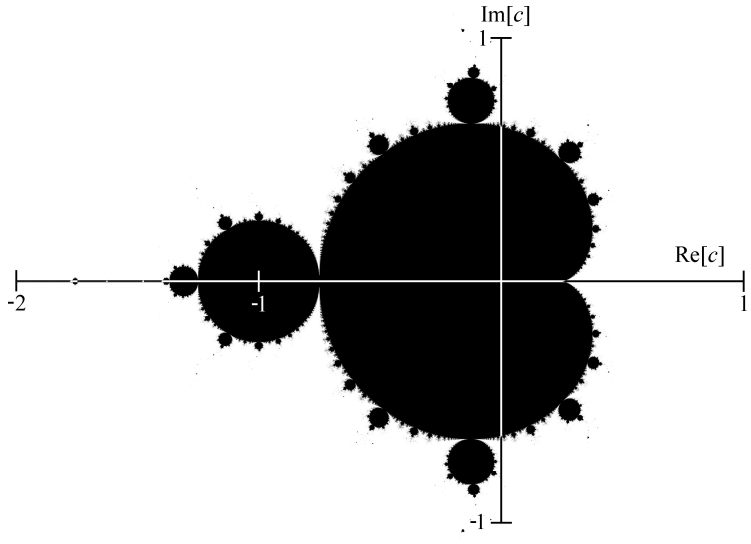
ALGORITHMICS IMAGES

- Mandelbrot set





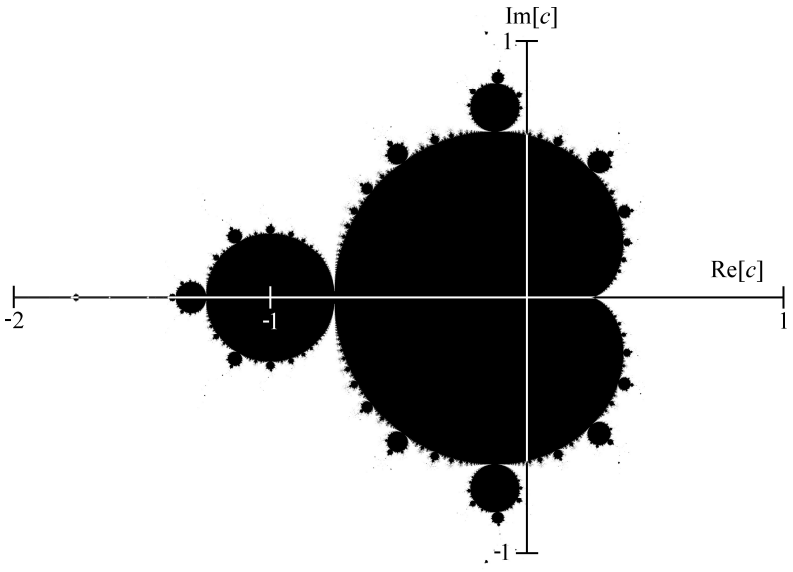
THE MANDELBROT SET



- The Mandelbrot Set is the set of all complex numbers **c** for which orbit of $z_0 = 0$ is bounded
- Numerically: If in a sequence of N numbers $|z_n|$ is never higher than 2 then c is in the set

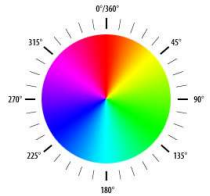
EXAMPLE : THE MANDELBROT SET

- To visualize the Mandelbrot set, we define a binary image I_{MS}.
- We consider the coordinates of a pixel at position (x,y) as the components of a complex number $c = x + iy$.
- Then the binary Mandelbrot set image I_{MS} is defined as
- $I_{MS}((x,y)) = \begin{cases} 0, & \text{if c=x+jy belongs to the Mandelbrot set,} \\ 1, & \text{otherwise.} \end{cases}$
- This image tells which values c provides bounded orbits for 0

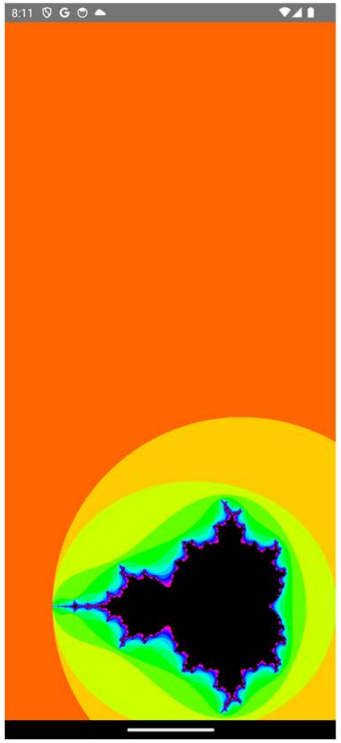


COLORED MANDELBROT SET

```
for each (x, y) in [HxW]
        c = x+iy
        z = 0
        iter = 0
        while iter < maxIter:
                z = z*z + c
                if |z| > 2:    break
                iter += 1
        IMG.DrawPixel(x,y)=colorFor(iter)
Return IMG
```



HSV, H=360 x iter/maxIter



maxIter=15

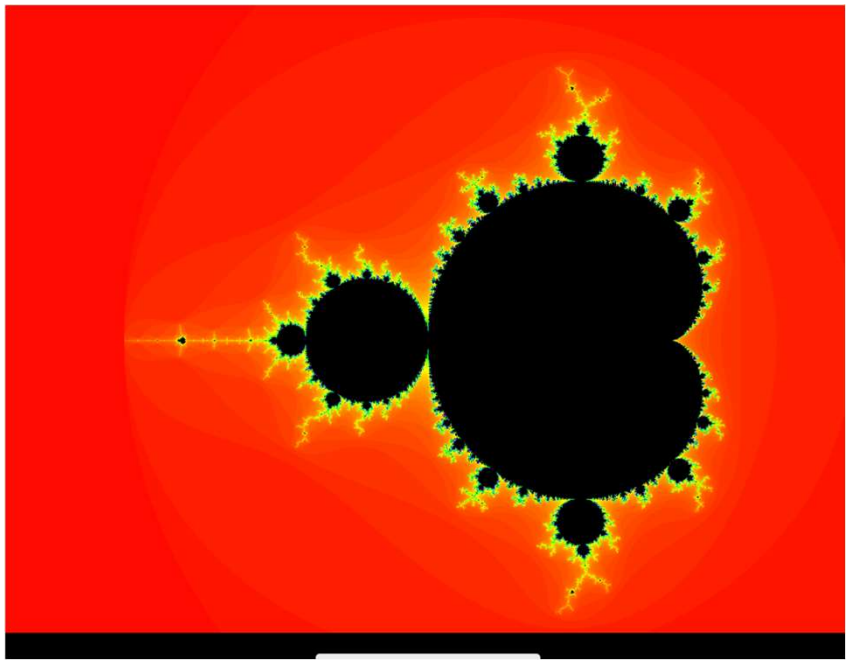
In the colored version of the Mandelbrot set, each point outside the set is assigned a color according to how fast its orbit escapes to infinity (|z|>2), escape-time coloring

For example, setting N=15 iterations

- Red= escape after 1 iteration
- Orange = escape after 3 iterations
- Green = 7 iterations
- Black = not escape (in the set)

COLORED MANDELBROT SET

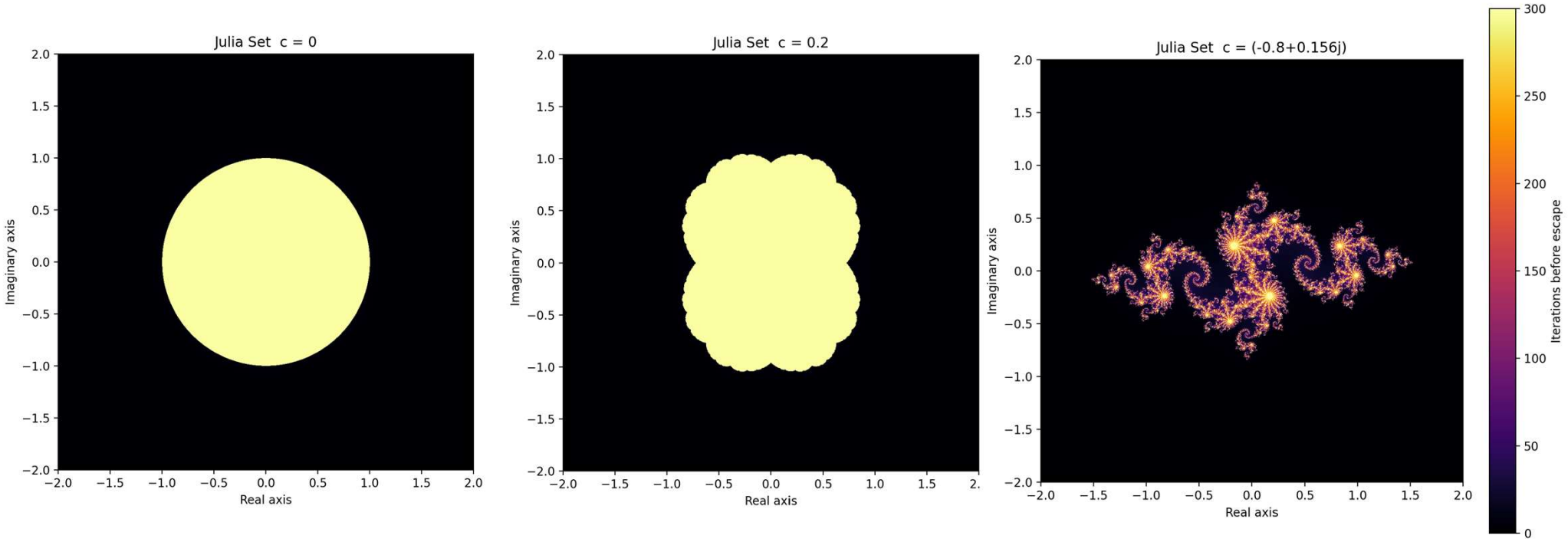
- With 150 iterations the images shows more details



JULIA SETS

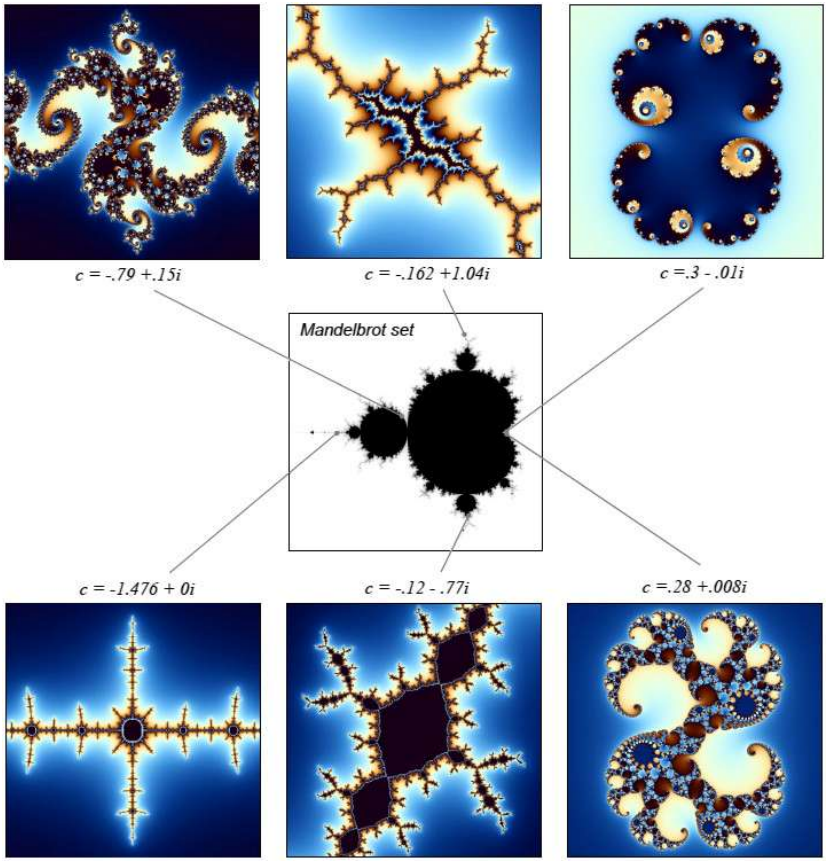
- Julie set uses the same idea, but c is fixed (parameter of the set)
- A pixel belongs to the set (it is black) if the orbit that starts from that pixel doesn't explode (remains bounded)

SOME EXAMPLES OF JULIA SET



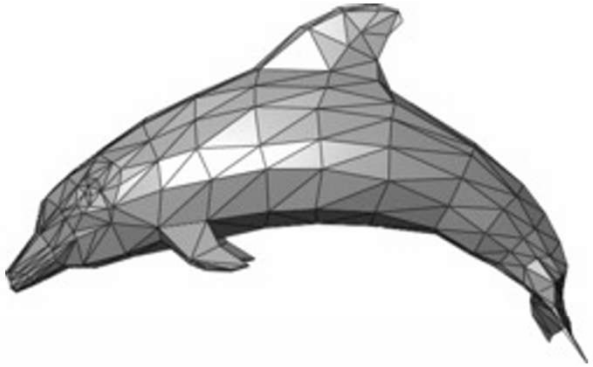
JULIA SETS AND MANDELBROT SET

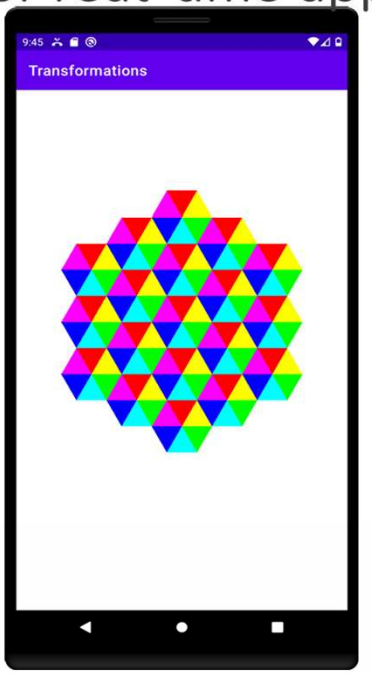
- Any point in the Mandelbrot Set generates a connected Julia set
- Widley used in graphics (psychedelic effects, texture, background in games, etc..)

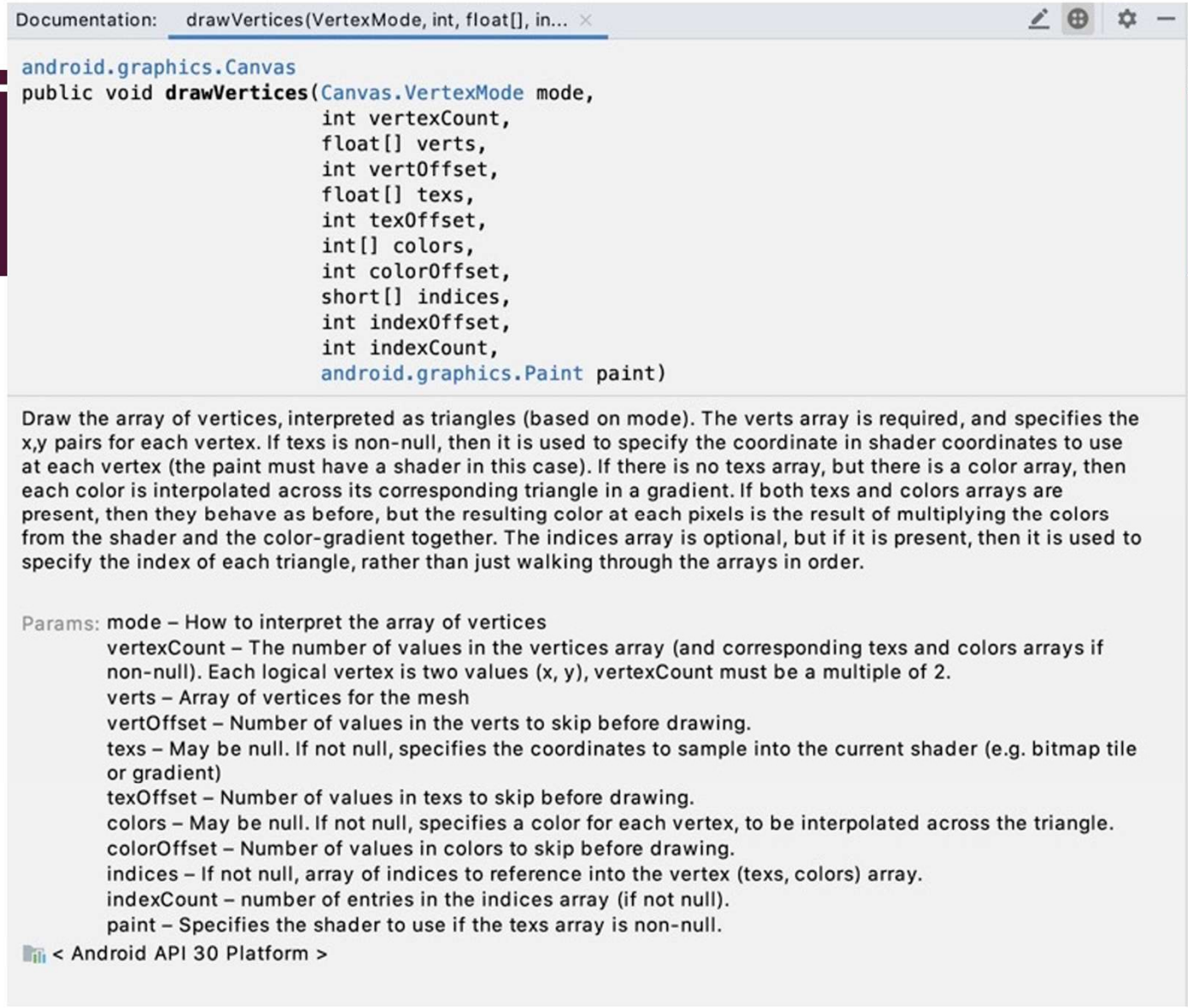


DRAWING ARBITRARY SHAPES

- Triangle is the basic shape used to create a **mesh** that approximates an arbitrary shape
- Triangles allow efficient rendering, especially needed for real-time apps





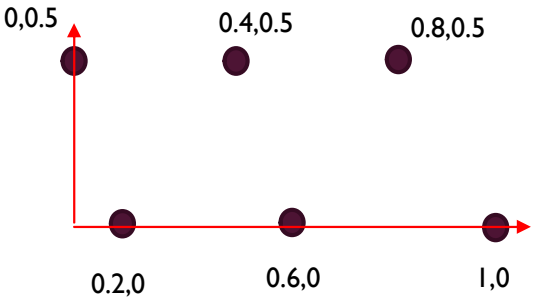

Documentation: drawVertices(VertexMode, int, float[], in...
android.graphics.Canvas
public void drawVertices(Canvas.VertexMode mode,
int vertexCount,
float[] verts,
int vertOffset,
float[] texs,
int texOffset,
int[] colors,
int colorOffset,
short[] indices,
int indexOffset,
int indexCount,
android.graphics.Paint 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.
Params: mode – How to interpret the array of vertices
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.
verts – Array of vertices for the mesh
vertOffset – Number of values in the verts to skip before drawing.
texs – May be null. If not null, specifies the coordinates to sample into the current shader (e.g. bitmap tile or gradient)
texOffset – Number of values in texs to skip before drawing.
colors – May be null. If not null, specifies a color for each vertex, to be interpolated across the triangle.
colorOffset – Number of values in colors to skip before drawing.
indices – If not null, array of indices to reference into the vertex (texs, colors) array.
indexCount – number of entries in the indices array (if not null).
paint – Specifies the shader to use if the texs array is non-null.
< Android API 30 Platform >

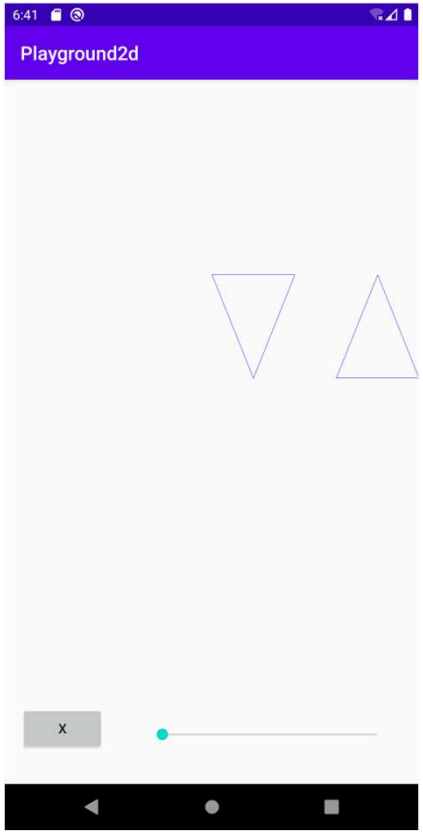
DRAWING MODE OF TRIANGLES

- A triangle is defined by three vertexes
- Given a **stream** of vertexes there are three different modes to draw triangles from these vertexes:
- TRIANGLE
- STRIP
- FAN

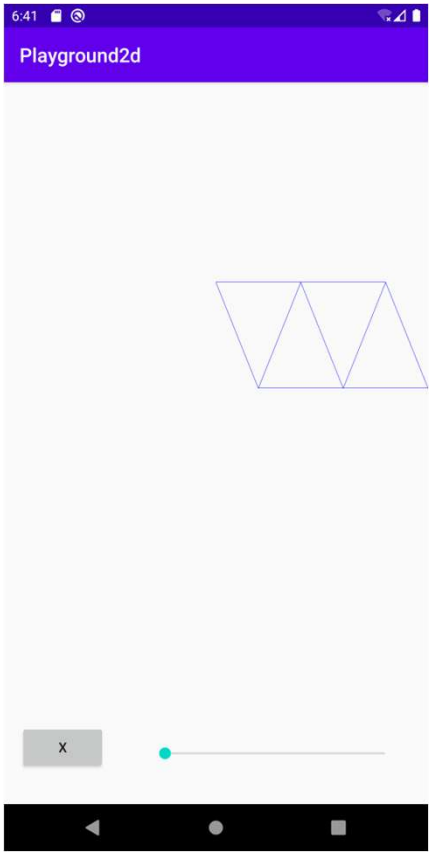
MODES: TRIANGLE, STRIP, FAN

- Triangle: Any 3 vertexes define a triangle
- Strip: Set of 'continuous' triangle
- Fan: Set of triangles sharing one vertex

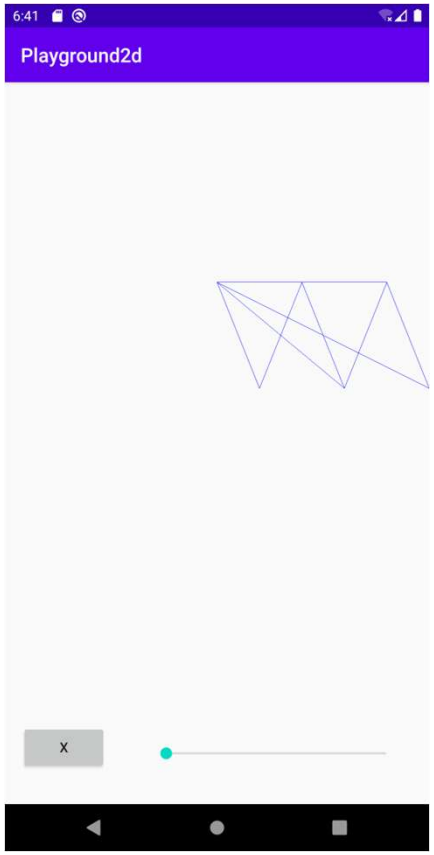




Triangle

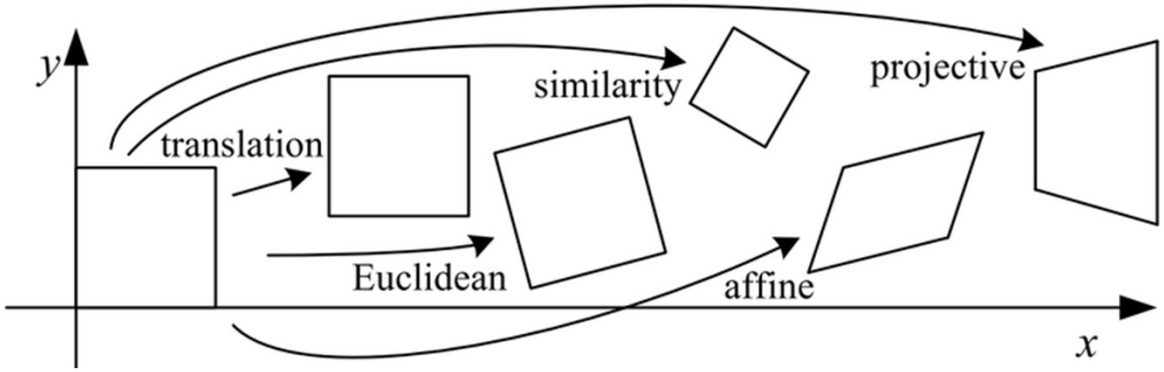


Strip

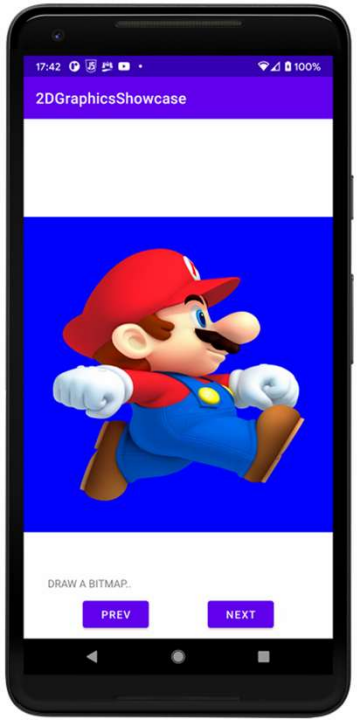


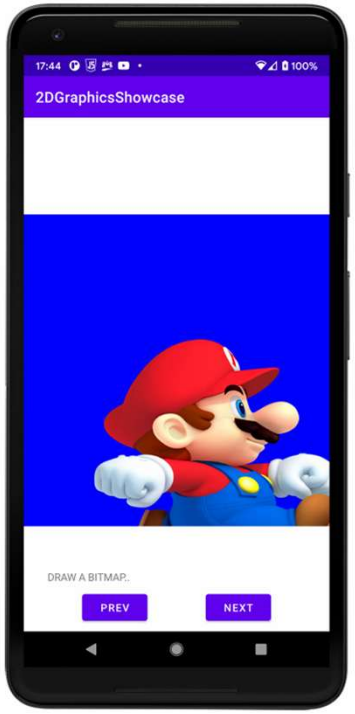
Fan

HOW TO MOVE A 2D OBJECT?

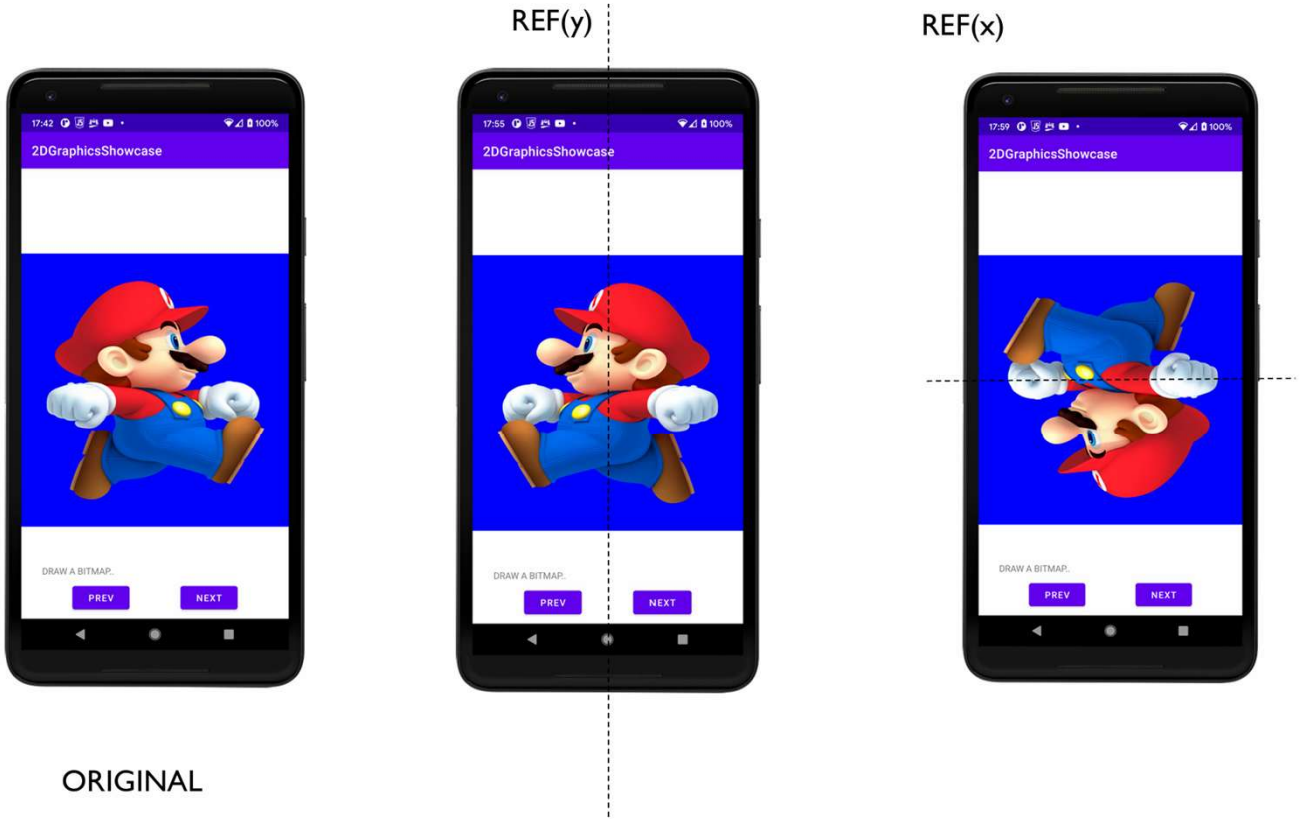


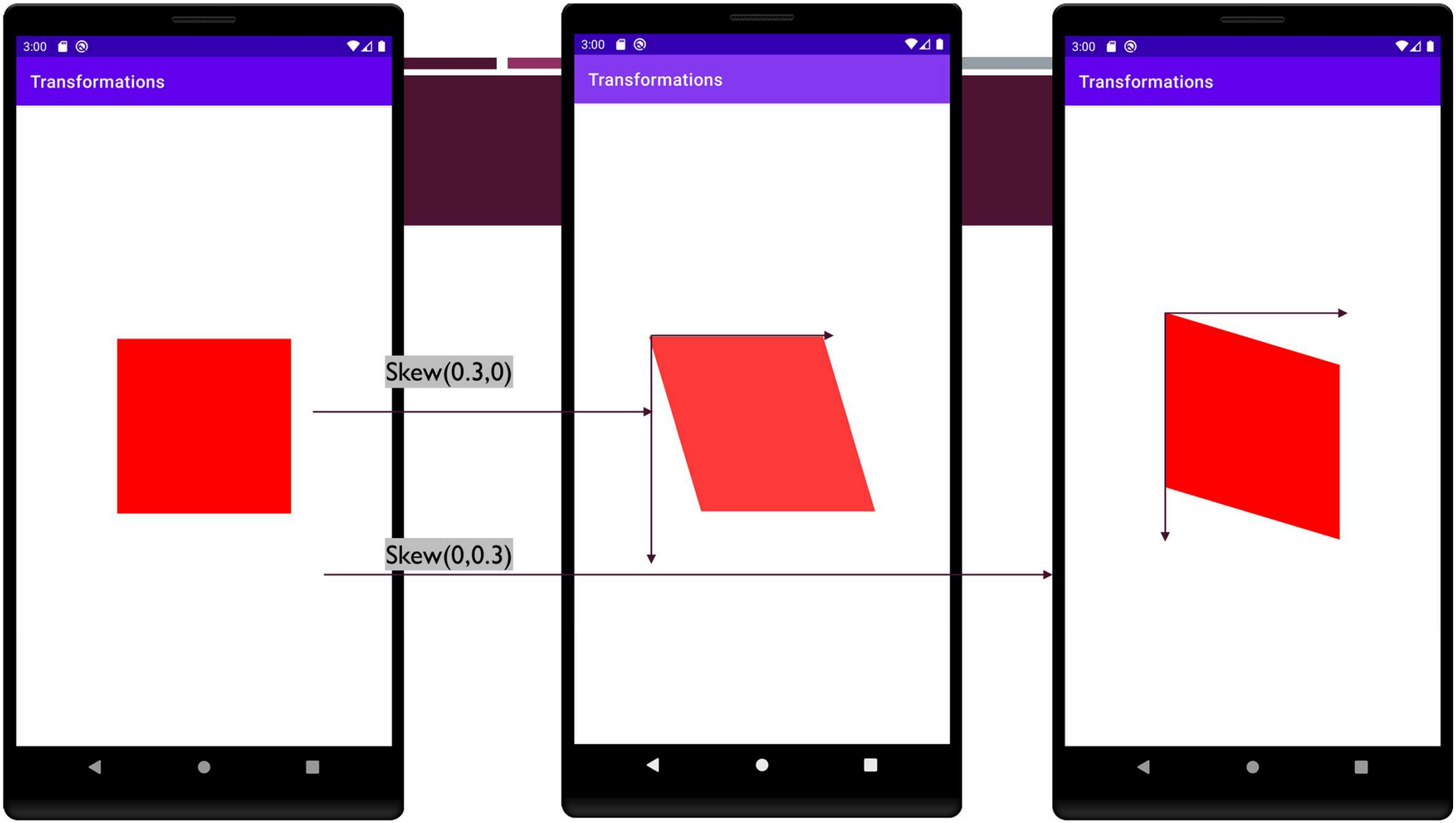
EXAMPLE: TRANSLATION





180 ROT = 2 REFLECTIONS




3:00
Transformations
Skew(0.3,0)
Skew(0,0.3)
3:00
Transformations
3:00
Transformations

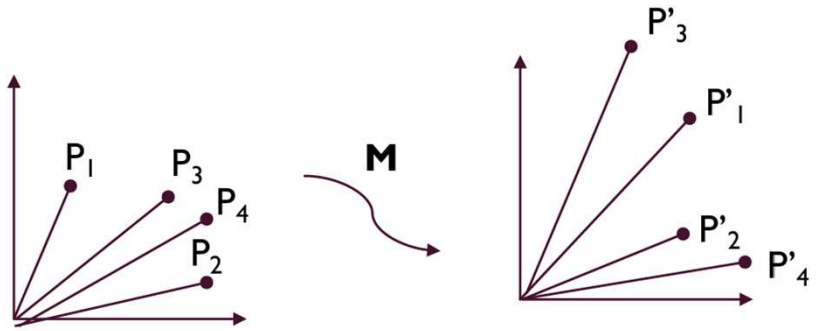
PROJECTIVE TRASFORMATIONS

$$\underbrace{\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & 1 \end{pmatrix}}_{M} \underbrace{\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}}_{P} = \underbrace{\begin{pmatrix} a_{11}x + a_{12}y + a_{13} \\ a_{21}x + a_{22}y + a_{23} \\ a_{31}x + a_{32}y + 1 \end{pmatrix}}_{P'}$$

$$x' = \frac{a_{11}x + a_{12}y + a_{13}}{a_{31}x + a_{32}y + 1}, \qquad y' = \frac{a_{21}x + a_{22}y + a_{23}}{a_{31}x + a_{32}y + 1}$$

- The most general form of 2D transformation that preserve lines is the projective transformation that is defined using a 3x3 matrix
- Points have the last component = 1
- After the transformation is applied, the coordinates of the point P' are normalized

FIND THE MATRIX FROM POINTS, GENERAL CASE [PROJECTIVE]



- A perspective matrix **M** can be computed knowing how 4 points are mapped by **M**

EXAMPLE OF A PROJECTIVE TRANSFORMATION

PROJECTIVE TRANSFORMATIONS

- Projective transformations have the last row with not zeros
- Without loss of generativity, the last element is one
- The are 8 independent coefficients to chose
- The transformation preserves straight lines, but not parallelism among lines

$$\mathbf{M}= \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & 1 \end{pmatrix}$$

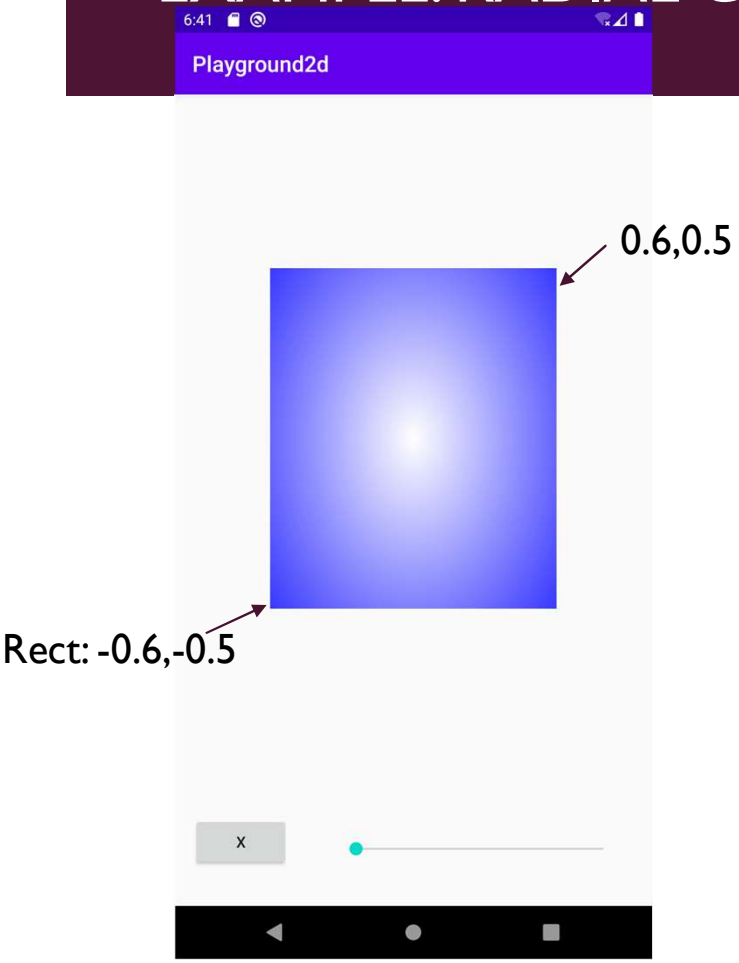
Space: 1999

SHADER

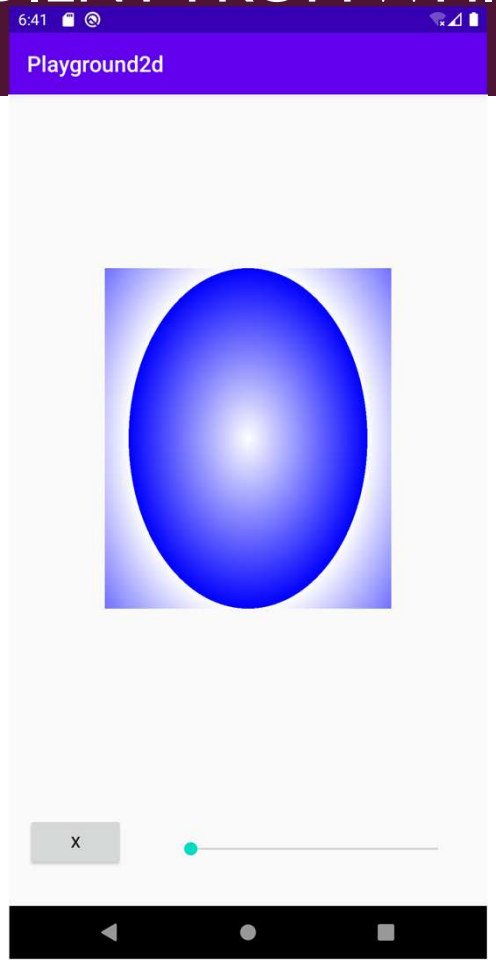
- Define the colours used to fill a shape
- More flexible than interpolation
- Define how colours change from one to another
  - Radial, Gradient
- Use a Local Transformation Matrix to modify the fill pattern, before applying shading

```
private var S = floatArrayOf(-0.6f,-0.5f, 0.6f,0.5f)
```

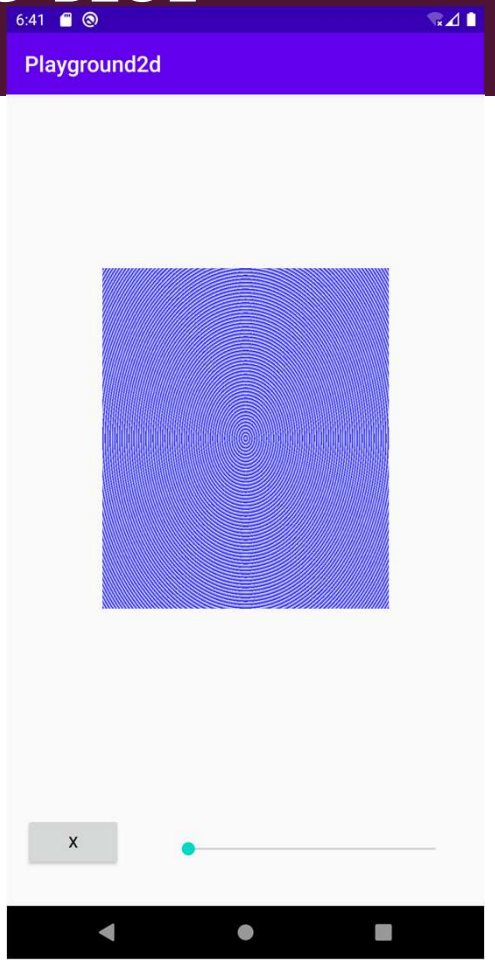
EXAMPLE: RADIAL GRADIENT FROM WHITE TO BLUE



centre=0,radius=1



centre=0,radius=0.5

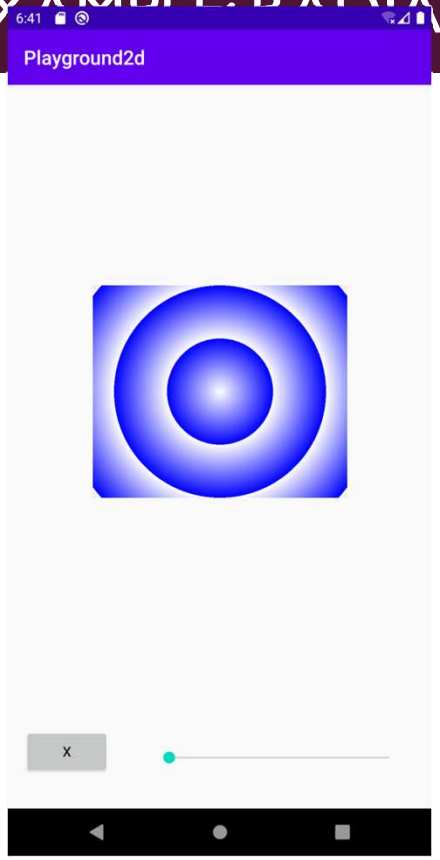


centre=0,radius=0.01

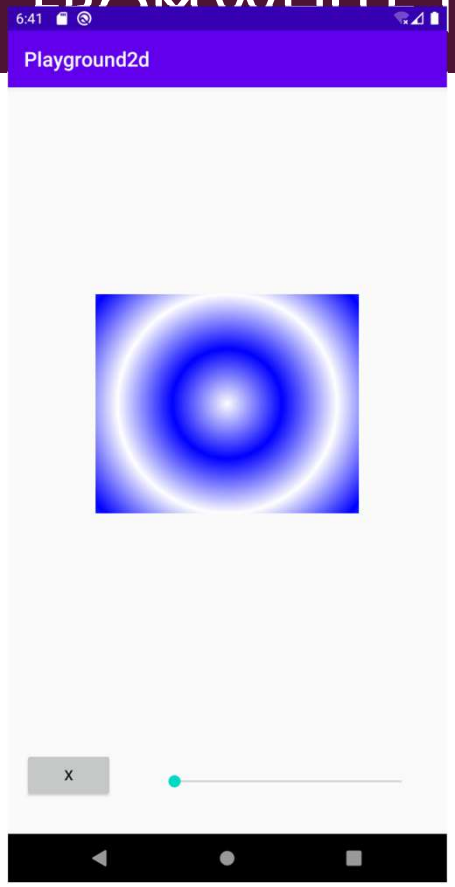
RadialGradient(0f,0f,0.25f,Color.*WHITE*,Color.*BLUE*,Shader.TileMode.**MIRROR**)

Edge rectangle = 1
Radius shader = 0.25

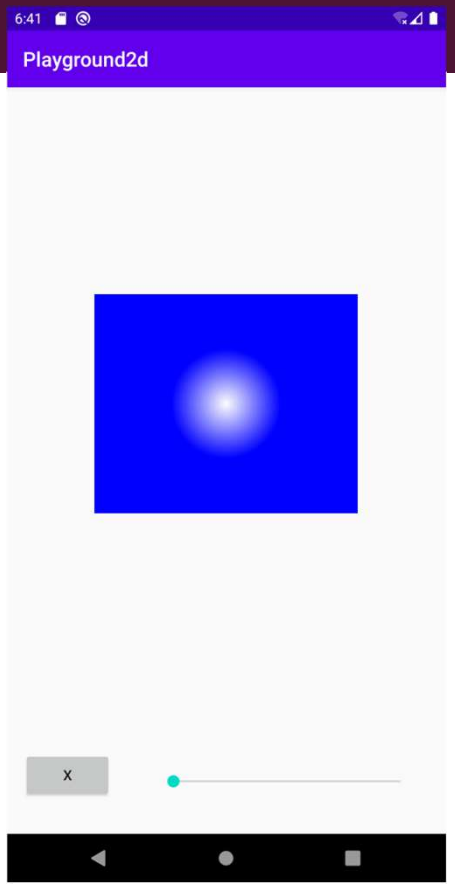
EXAMPLE: RADIAL GRADIENT FROM WHITE TO BLUE



Repeat



Mirror



Clamp

FILLING THE TRIANGLE

```
val vertex = floatArrayOf(
        200f,200f,
        300f,1000f,
        1000f,1000f)
```

Colour along an edge

$C_{02}(\alpha)=\alpha C_0+(1-\alpha)C_2$

gradient interpolation

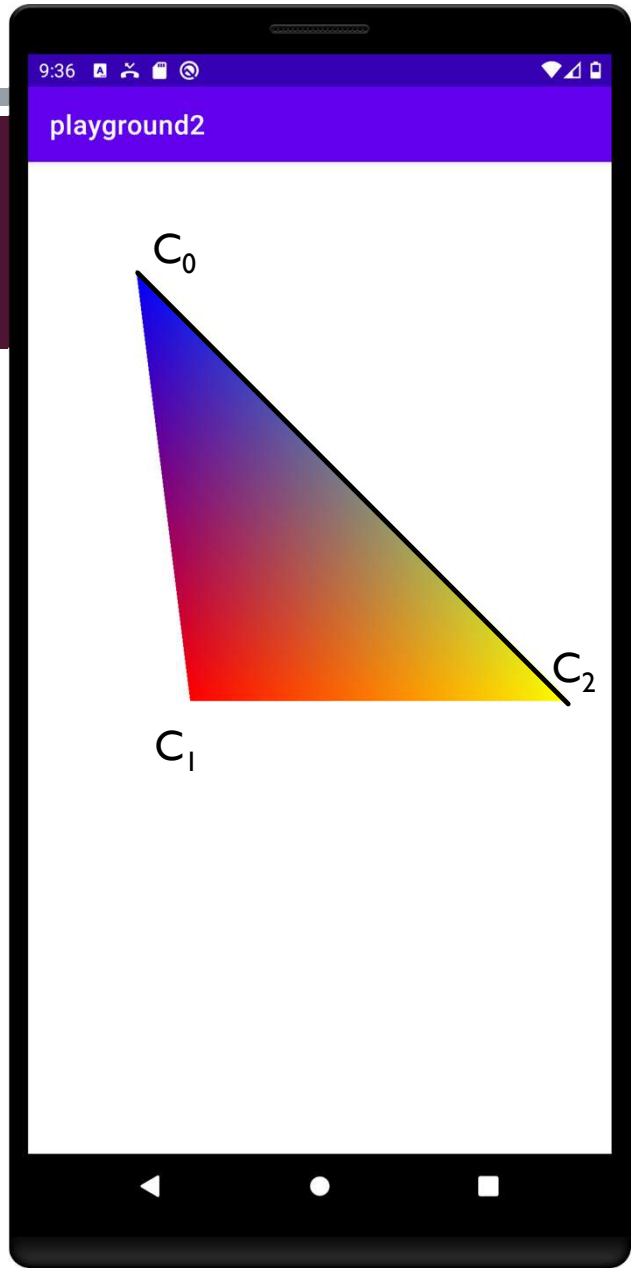
```
val colors = intArrayOf(
        Color.BLUE,
        Color.RED,
        Color.YELLOW)
```

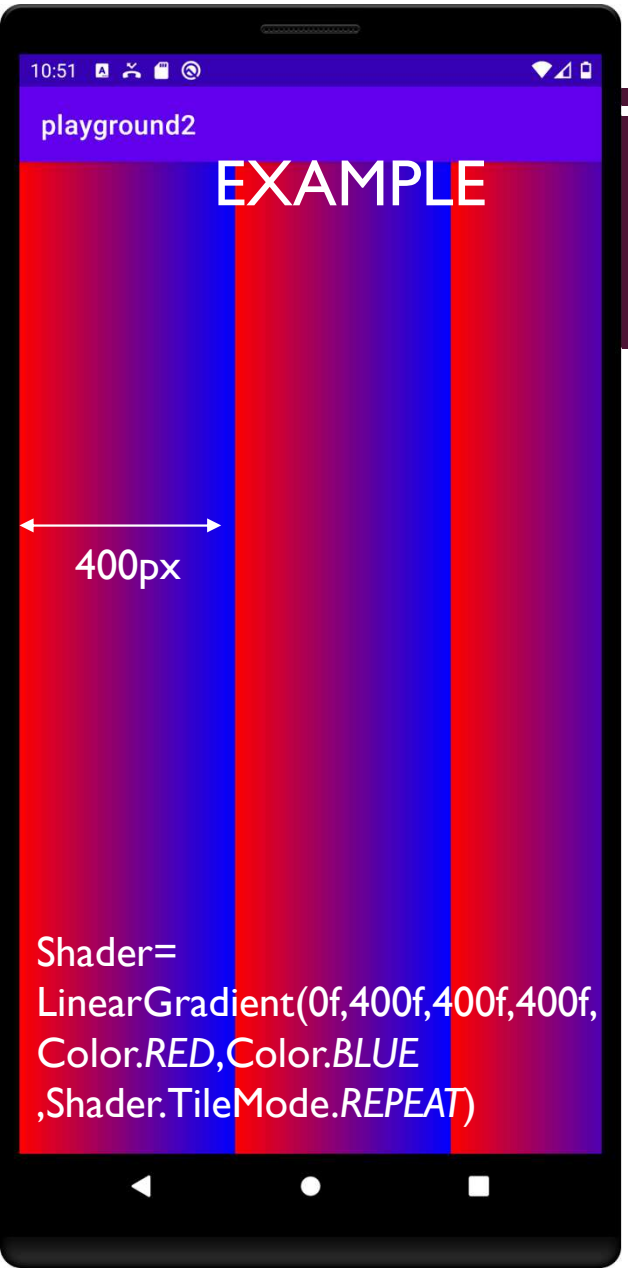
Colour of a inner pixel

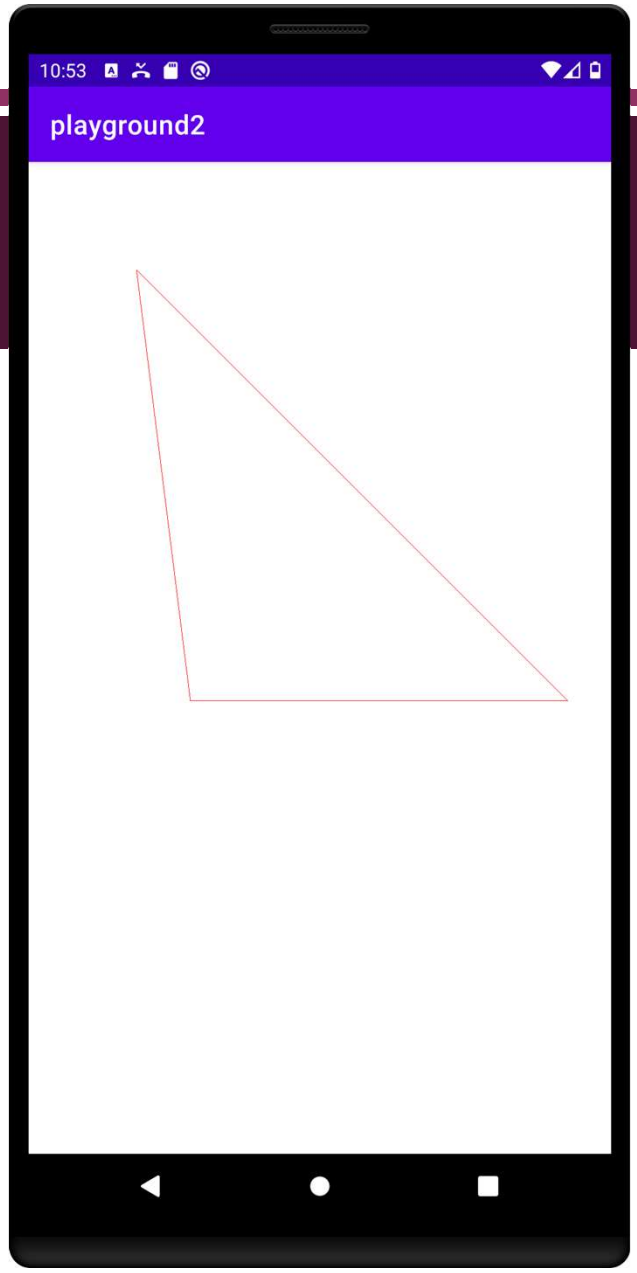
$C_4(\beta)=\beta C_1+(1-\beta)C_{02}$

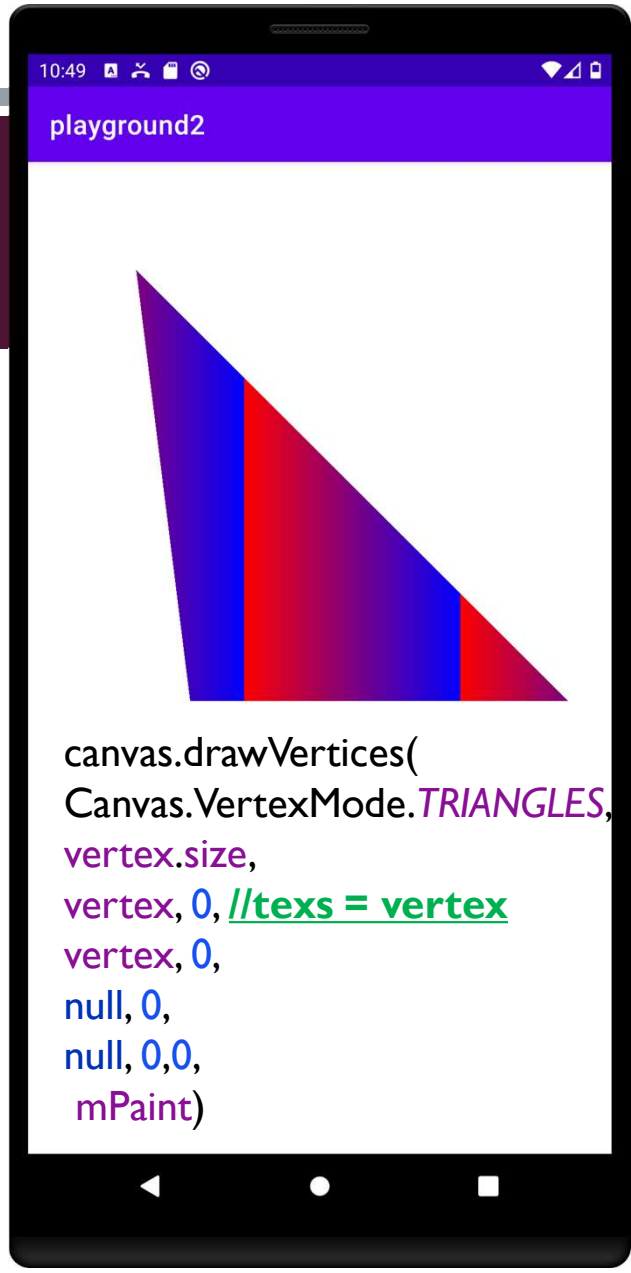
```
canvas.drawVertices(Canvas.VertexMode.TRIANGLES,vertex.size,vertex,
        0,
        null, 0,
        colors, 0,
        null, 0,0,
        mPaint)
```

No texture




10:51
playground2
EXAMPLE
400px
Shader=
LinearGradient(0f,400f,400f,400f,
Color.RED,Color.BLUE
,Shader.TileMode.REPEAT)

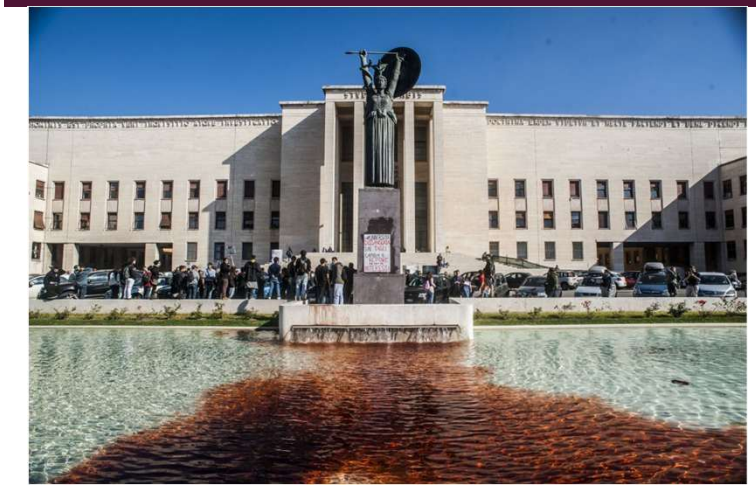

10:53
playground2


10:49
playground2
canvas.drawVertices(
Canvas.VertexMode.TRIANGLES,
vertex.size,
vertex, 0, //texs = vertex
vertex, 0,
null, 0,
null, 0,0,
mPaint)

BITMAP TEXTURE

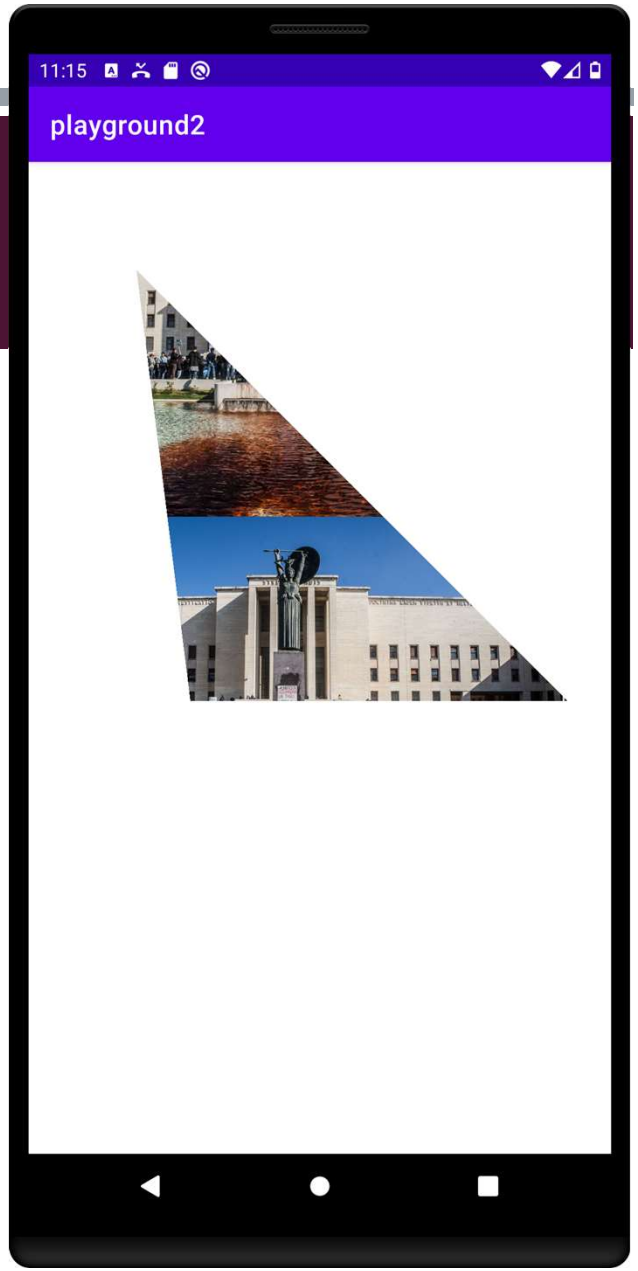
- Bitmap image as textures mimics a surface, a material, a pattern or even a picture
- Provide more sense of reality

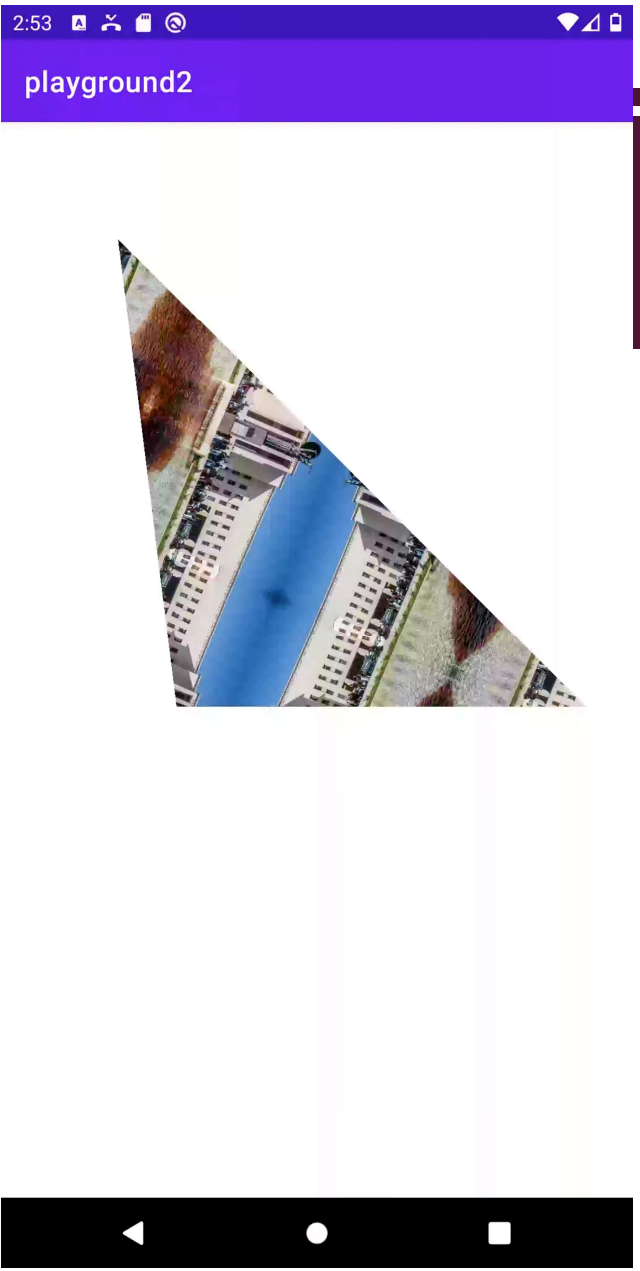
EXAMPLE



```
var sapienza =
BitmapFactory.decodeStream(getContext().assets.open("sapienza.jpg"))
```

```
BitmapShader(sapienza,Shader.TileMode.REPEAT,Shader.TileMode.REPEAT)
```





EXAMPLE

COMPUTATION COST AND COMPOSITION

- To move a point according to matrix M1 and M2, one can
- Apply M1 to the point p to obtain p'
  - 15 operations: 3 x (3M+2A)
- Apply M2 to p' to obtain p''
- Or compute M=M2M1 and then M'p
  - M requires to compute 9 elements: 9 x (3M+2A) = 45
  - Even if the cost of 2 is higher, ii becomes convenient if more than one point must be multiplied

- The order of multiplication is important
- In 2D, two rotations can be swapped (not true in 3D),