Snappy Ruler Set - Complete Android Development Guide

Table of Contents

Project Overview

The Snappy Ruler Set is an Android drawing application that provides virtual geometry tools with intelligent snapping capabilities. Users can draw freehand and use precision tools like rulers, set squares, protractors, and compasses.

Key Features

- Freehand drawing with pen/finger
- Virtual ruler with rotation and snapping
- Set squares (45° and 30°-60° variants)
- Protractor with angle measurement
- Compass for circles and arcs
- Intelligent snapping system
- Undo/Redo functionality
- Export to PNG/JPEG

Technical Stack

- **Language**: Kotlin
- **UI Framework**: Jetpack Compose
- **Architecture**: MVVM with Repository pattern
- **Graphics**: Canvas API with custom drawing

- **State Management**: StateFlow and Compose State
- **Testing**: JUnit, Espresso, Compose Testing

Prerequisites

Development Environment

- Android Studio Hedgehog (2023.1.1) or later
- JDK 17 or higher
- Android SDK API 24+ (Android 7.0)
- Kotlin 1.9.0+

Hardware Requirements

- Minimum 8GB RAM (16GB recommended)
- 10GB free disk space
- Android device or emulator for testing

Knowledge Requirements

- Basic Kotlin programming
- Android development fundamentals
- Jetpack Compose basics
- Canvas drawing concepts

Project Setup

Step 1: Create New Android Project

1. Open Android Studio
2. Click "Create New Project"
3. Select "Empty Compose Activity"
4. Configure project:
    - Name: `SnapRulerSet`
    - Package: `com.yourname.snaprulerset`
    - Language: Kotlin
    - Minimum SDK: API 24
    - Build configuration: Kotlin DSL

Step 2: Update build.gradle.kts (Module: app)

```
kotlin
```

```
android {
    namespace = "com.yourname.snaprulerset"
    compileSdk = 34

    defaultConfig {
        applicationId = "com.yourname.snaprulerset"
        minSdk = 24
        targetSdk = 34
        versionCode = 1
        versionName = "1.0"

        testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"
        vectorDrawables {
            useSupportLibrary = true
        }
    }

    buildTypes {
        release {
            isMinifyEnabled = false
            proguardFiles(
                getDefaultProguardFile("proguard-android-optimize.txt"),
                "proguard-rules.pro"
            )
        }
    }

    compileOptions {
        sourceCompatibility = JavaVersion.VERSION_17
        targetCompatibility = JavaVersion.VERSION_17
    }

    kotlinOptions {
        jvmTarget = "17"
    }

    buildFeatures {
        compose = true
    }

    composeOptions {
        kotlinCompilerExtensionVersion = "1.5.8"
    }

    packaging {
        resources {
            excludes += "/META-INF/{AL2.0,LGPL2.1}"
        }
    }
}

dependencies {
    implementation("androidx.core:core-ktx:1.12.0")
    implementation("androidx.lifecycle:lifecycle-runtime-ktx:2.7.0")
    implementation("androidx.activity:activity-compose:1.8.2")
    implementation("androidx.compose.ui:ui:1.5.8")
    implementation("androidx.compose.ui:ui-tooling-preview:1.5.8")
    implementation("androidx.compose.material3:material3:1.1.2")

    // ViewModel
    implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.7.0")

    // Navigation
    implementation("androidx.navigation:navigation-compose:2.7.6")

    // Permissions
    implementation("com.google.accompanist:accompanist-permissions:0.32.0")

    // Testing
    testImplementation("junit:junit:4.13.2")
    testImplementation("org.mockito:mockito-core:5.8.0")
    androidTestImplementation("androidx.test.ext:junit:1.1.5")
    androidTestImplementation("androidx.test.espresso:espresso-core:3.5.1")
    androidTestImplementation("androidx.compose.ui:ui-test-junit4:1.5.8")

    debugImplementation("androidx.compose.ui:ui-tooling:1.5.8")
    debugImplementation("androidx.compose.ui:ui-test-manifest:1.5.8")
}
```

Step 3: Add Permissions (AndroidManifest.xml)

```
xml
```

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools">

  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
  <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />

  <application
    android:allowBackup="true"
    android:dataExtractionRules="@xml/data_extraction_rules"
    android:fullBackupContent="@xml/backup_rules"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/Theme.SnapRulerSet"
    tools:targetApi="31">

    <activity
      android:name=".MainActivity"
      android:exported="true"
      android:screenOrientation="portrait"
      android:theme="@style/Theme.SnapRulerSet">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Architecture Overview

MVVM Architecture Pattern

```
┌─────────────────┐    ┌─────────────────┐    ┌─────────────────┐
│   Composable    │───▶│   ViewModel     │───▶│   Repository    │
│   (View Layer)  │    │  (Logic Layer)  │    │  (Data Layer)   │
└─────────────────┘    └─────────────────┘    └─────────────────┘
         │                       │                       │
         ▼                       ▼                       ▼
┌─────────────────┐    ┌─────────────────┐    ┌─────────────────┐
│   UI State      │    │   Use Cases     │    │  Data Models    │
│  (StateFlow)    │    │  (Business)     │    │  (Entities)     │
└─────────────────┘    └─────────────────┘    └─────────────────┘
```

Key Components

1. **Drawing Engine**: Handles canvas operations and rendering
2. **Tool System**: Manages geometry tools (ruler, protractor, etc.)
3. **Snapping System**: Implements intelligent snapping logic
4. **Gesture Handler**: Processes touch inputs and gestures
5. **State Manager**: Manages drawing state and undo/redo

Core Components Implementation

1. Data Models

DrawingState.kt

kotlin

```kotlin
package com.yourname.snaprulerset.model

import androidx.compose.ui.geometry.Offset
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.Path

data class DrawingState(
    val paths: List<DrawingPath> = emptyList(),
    val tools: List<GeometryTool> = emptyList(),
    val selectedTool: ToolType? = null,
    val isDrawing: Boolean = false,
    val currentPath: Path = Path(),
    val canvasOffset: Offset = Offset.Zero,
    val canvasScale: Float = 1f,
    val gridVisible: Boolean = true,
    val gridSpacing: Float = 50f, // 5mm at 160dpi
    val snapEnabled: Boolean = true,
    val snapRadius: Float = 20f
)

data class DrawingPath(
    val path: Path,
    val color: Color,
    val strokeWidth: Float,
    val timestamp: Long = System.currentTimeMillis()
)

sealed class GeometryTool {
    abstract val id: String
    abstract val position: Offset
    abstract val rotation: Float

    data class Ruler(
        override val id: String = "ruler_${System.currentTimeMillis()}",
        override val position: Offset,
        override val rotation: Float = 0f,
        val length: Float = 300f // 30cm at 160dpi
    ) : GeometryTool()

    data class SetSquare(
        override val id: String = "setsquare_${System.currentTimeMillis()}",
        override val position: Offset,
        override val rotation: Float = 0f,
        val type: SetSquareType = SetSquareType.TRIANGLE_45,
        val size: Float = 200f
    ) : GeometryTool()
    data class Protractor(
        override val id: String = "protractor_${System.currentTimeMillis()}",
        override val position: Offset,
        override val rotation: Float = 0f,
        val radius: Float = 100f,
        val startAngle: Float = 0f,
        val sweepAngle: Float = 180f
    ) : GeometryTool()

    data class Compass(
        override val id: String = "compass_${System.currentTimeMillis()}",
        override val position: Offset,
        override val rotation: Float = 0f,
        val radius: Float = 50f,
        val center: Offset = position
    ) : GeometryTool()
}

enum class ToolType {
    PEN, RULER, SET_SQUARE_45, SET_SQUARE_30_60, PROTRACTOR, COMPASS
}

enum class SetSquareType {
    TRIANGLE_45, TRIANGLE_30_60
}

data class SnapPoint(
    val position: Offset,
    val type: SnapType,
    val strength: Float = 1f // 0-1, higher = stronger snap
)

enum class SnapType {
    GRID, ENDPOINT, MIDPOINT, INTERSECTION, ANGLE, CIRCLE_CENTER
}
```

2. Snapping System

SnapEngine.kt

kotlin

```kotlin
package com.yourname.snaprulerset.engine

import androidx.compose.ui.geometry.Offset
import com.yourname.snaprulerset.model.*
import kotlin.math.*

class SnapEngine {

    fun findSnapPoints(
        targetPoint: Offset,
        drawingState: DrawingState,
        snapRadius: Float
    ): List<SnapPoint> {
        if (!drawingState.snapEnabled) return emptyList()

        val snapPoints = mutableListOf<SnapPoint>()

        // Grid snapping
        if (drawingState.gridVisible) {
            snapPoints.addAll(findGridSnaps(targetPoint, drawingState.gridSpacing, snapRadius))
        }

        // Tool snapping
        drawingState.tools.forEach { tool ->
            snapPoints.addAll(findToolSnaps(targetPoint, tool, snapRadius))
        }

        // Path snapping
        drawingState.paths.forEach { drawingPath ->
            snapPoints.addAll(findPathSnaps(targetPoint, drawingPath, snapRadius))
        }

        return snapPoints.filter {
            distance(targetPoint, it.position) <= snapRadius
        }.sortedBy {
            distance(targetPoint, it.position)
        }
    }

    private fun findGridSnaps(
        targetPoint: Offset,
        gridSpacing: Float,
        snapRadius: Float
    ): List<SnapPoint> {
        val snapX = round(targetPoint.x / gridSpacing) * gridSpacing
        val snapY = round(targetPoint.y / gridSpacing) * gridSpacing
    val gridSnap = Offset(snapX, snapY)

    return if (distance(targetPoint, gridSnap) <= snapRadius) {
        listOf(SnapPoint(gridSnap, SnapType.GRID, 0.8f))
    } else emptyList()
}

private fun findToolSnaps(
    targetPoint: Offset,
    tool: GeometryTool,
    snapRadius: Float
): List<SnapPoint> {
    return when (tool) {
        is GeometryTool.Ruler -> findRulerSnaps(targetPoint, tool, snapRadius)
        is GeometryTool.SetSquare -> findSetSquareSnaps(targetPoint, tool, snapRadius)
        is GeometryTool.Protractor -> findProtractorSnaps(targetPoint, tool, snapRadius)
        is GeometryTool.Compass -> findCompassSnaps(targetPoint, tool, snapRadius)
    }
}

private fun findRulerSnaps(
    targetPoint: Offset,
    ruler: GeometryTool.Ruler,
    snapRadius: Float
): List<SnapPoint> {
    val snapPoints = mutableListOf<SnapPoint>()

    // Calculate ruler endpoints
    val halfLength = ruler.length / 2
    val cos = cos(ruler.rotation)
    val sin = sin(ruler.rotation)

    val start = Offset(
        ruler.position.x - halfLength * cos,
        ruler.position.y - halfLength * sin
    )
    val end = Offset(
        ruler.position.x + halfLength * cos,
        ruler.position.y + halfLength * sin
    )

    // Endpoint snapping
    if (distance(targetPoint, start) <= snapRadius) {
        snapPoints.add(SnapPoint(start, SnapType.ENDPOINT, 1f))
    }
    if (distance(targetPoint, end) <= snapRadius) {
        snapPoints.add(SnapPoint(end, SnapType.ENDPOINT, 1f))
    }

    // Midpoint snapping
    val midpoint = Offset(
        (start.x + end.x) / 2,
        (start.y + end.y) / 2
    )
    if (distance(targetPoint, midpoint) <= snapRadius) {
        snapPoints.add(SnapPoint(midpoint, SnapType.MIDPOINT, 0.9f))
    }

    return snapPoints
}

private fun findSetSquareSnaps(
    targetPoint: Offset,
    setSquare: GeometryTool.SetSquare,
    snapRadius: Float
): List<SnapPoint> {
    val snapPoints = mutableListOf<SnapPoint>()

    // Calculate triangle vertices based on type
    val vertices = when (setSquare.type) {
        SetSquareType.TRIANGLE_45 -> calculate45TriangleVertices(setSquare)
        SetSquareType.TRIANGLE_30_60 -> calculate30_60TriangleVertices(setSquare)
    }

    // Add vertex snaps
    vertices.forEach { vertex ->
        if (distance(targetPoint, vertex) <= snapRadius) {
            snapPoints.add(SnapPoint(vertex, SnapType.ENDPOINT, 1f))
        }
    }

    // Add edge midpoint snaps
    for (i in vertices.indices) {
        val nextIndex = (i + 1) % vertices.size
        val midpoint = Offset(
            (vertices[i].x + vertices[nextIndex].x) / 2,
            (vertices[i].y + vertices[nextIndex].y) / 2
        )
        if (distance(targetPoint, midpoint) <= snapRadius) {
            snapPoints.add(SnapPoint(midpoint, SnapType.MIDPOINT, 0.9f))
        }
    }

    return snapPoints
```

```
}

private fun findProtractorSnaps(
    targetPoint: Offset,
    protractor: GeometryTool.Protractor,
    snapRadius: Float
): List<SnapPoint> {
    val snapPoints = mutableListOf<SnapPoint>()

    // Center snap
    if (distance(targetPoint, protractor.position) <= snapRadius) {
        snapPoints.add(SnapPoint(protractor.position, SnapType.CIRCLE_CENTER, 1f))
    }

    // Angle snaps (every 15 degrees)
    for (angle in 0..360 step 15) {
        val radians = Math.toRadians(angle.toDouble()).toFloat()
        val snapPoint = Offset(
            protractor.position.x + protractor.radius * cos(radians),
            protractor.position.y + protractor.radius * sin(radians)
        )

        if (distance(targetPoint, snapPoint) <= snapRadius) {
            val strength = if (angle % 30 == 0) 1f else 0.7f
            snapPoints.add(SnapPoint(snapPoint, SnapType.ANGLE, strength))
        }
    }

    return snapPoints
}

private fun findCompassSnaps(
    targetPoint: Offset,
    compass: GeometryTool.Compass,
    snapRadius: Float
): List<SnapPoint> {
    val snapPoints = mutableListOf<SnapPoint>()

    // Center snap
    if (distance(targetPoint, compass.center) <= snapRadius) {
        snapPoints.add(SnapPoint(compass.center, SnapType.CIRCLE_CENTER, 1f))
    }

    // Circle circumference snap
    val distanceFromCenter = distance(targetPoint, compass.center)
    if (abs(distanceFromCenter - compass.radius) <= snapRadius) {
        val angle = atan2(
```

```kotlin
            targetPoint.y - compass.center.y,
            targetPoint.x - compass.center.x
        )
        val snapPoint = Offset(
            compass.center.x + compass.radius * cos(angle),
            compass.center.y + compass.radius * sin(angle)
        )
        snapPoints.add(SnapPoint(snapPoint, SnapType.INTERSECTION, 0.8f))
    }

    return snapPoints
}

private fun findPathSnaps(
    targetPoint: Offset,
    drawingPath: DrawingPath,
    snapRadius: Float
): List<SnapPoint> {
    // For now, return empty - would need path analysis for endpoints/intersections
    return emptyList()
}

private fun calculate45TriangleVertices(setSquare: GeometryTool.SetSquare): List<Offset> {
    val size = setSquare.size
    val cos = cos(setSquare.rotation)
    val sin = sin(setSquare.rotation)

    // Right triangle with 45° angles
    val vertices = listOf(
        Offset(0f, 0f),
        Offset(size, 0f),
        Offset(size, size)
    )

    return vertices.map { vertex ->
        Offset(
            setSquare.position.x + vertex.x * cos - vertex.y * sin,
            setSquare.position.y + vertex.x * sin + vertex.y * cos
        )
    }
}

private fun calculate30_60TriangleVertices(setSquare: GeometryTool.SetSquare): List<Offset> {
    val size = setSquare.size
    val cos = cos(setSquare.rotation)
    val sin = sin(setSquare.rotation)
```

```
        // 30-60-90 triangle
        val height = size * sqrt(3f) / 2f
        val vertices = listOf(
            Offset(0f, 0f),
            Offset(size, 0f),
            Offset(size / 2f, height)
        )

        return vertices.map { vertex ->
            Offset(
                setSquare.position.x + vertex.x * cos - vertex.y * sin,
                setSquare.position.y + vertex.x * sin + vertex.y * cos
            )
        }
    }

    private fun distance(p1: Offset, p2: Offset): Float {
        return sqrt((p1.x - p2.x).pow(2) + (p1.y - p2.y).pow(2))
    }

    fun snapToCommonAngles(angle: Float): Float {
        val commonAngles = listOf(0f, 30f, 45f, 60f, 90f, 120f, 135f, 150f, 180f)
        val normalizedAngle = angle % 360f

        return commonAngles.minByOrNull { abs(normalizedAngle - it) } ?: normalizedAngle
    }
}
```

3. Drawing Engine

DrawingEngine.kt

kotlin

```kotlin
package com.yourname.snaprulerset.engine

import androidx.compose.ui.geometry.Offset
import androidx.compose.ui.graphics.*
import androidx.compose.ui.graphics.drawscope.DrawScope
import androidx.compose.ui.graphics.drawscope.Stroke
import androidx.compose.ui.graphics.drawscope.rotate
import androidx.compose.ui.graphics.drawscope.translate
import com.yourname.snaprulerset.model.*
import kotlin.math.*

class DrawingEngine {

    private val paint = Paint().apply {
        isAntiAlias = true
    }

    fun drawCanvas(
        drawScope: DrawScope,
        drawingState: DrawingState,
        snapPoints: List<SnapPoint>
    ) {
        with(drawScope) {
            // Draw grid
            if (drawingState.gridVisible) {
                drawGrid(drawingState.gridSpacing)
            }

            // Draw existing paths
            drawingState.paths.forEach { drawingPath ->
                drawPath(
                    path = drawingPath.path,
                    color = drawingPath.color,
                    style = Stroke(width = drawingPath.strokeWidth)
                )
            }

            // Draw current path being drawn
            if (drawingState.isDrawing) {
                drawPath(
                    path = drawingState.currentPath,
                    color = Color.Black,
                    style = Stroke(width = 4f)
                )
            }
        // Draw tools
        drawingState.tools.forEach { tool ->
            drawTool(tool, drawingState.selectedTool?.name == tool::class.simpleName)
        }

        // Draw snap indicators
        snapPoints.forEach { snapPoint ->
            drawSnapIndicator(snapPoint)
        }
    }
}

private fun DrawScope.drawGrid(spacing: Float) {
    val width = size.width
    val height = size.height

    val gridPaint = Paint().apply {
        color = Color.Gray.copy(alpha = 0.3f)
        strokeWidth = 1f
    }

    // Vertical lines
    var x = 0f
    while (x <= width) {
        drawLine(
            color = Color.Gray.copy(alpha = 0.3f),
            start = Offset(x, 0f),
            end = Offset(x, height),
            strokeWidth = 1f
        )
        x += spacing
    }

    // Horizontal lines
    var y = 0f
    while (y <= height) {
        drawLine(
            color = Color.Gray.copy(alpha = 0.3f),
            start = Offset(0f, y),
            end = Offset(width, y),
            strokeWidth = 1f
        )
        y += spacing
    }
}

private fun DrawScope.drawTool(tool: GeometryTool, isSelected: Boolean) {
    val color = if (isSelected) Color.Blue else Color.Black
    val strokeWidth = if (isSelected) 3f else 2f

    when (tool) {
        is GeometryTool.Ruler -> drawRuler(tool, color, strokeWidth)
        is GeometryTool.SetSquare -> drawSetSquare(tool, color, strokeWidth)
        is GeometryTool.Protractor -> drawProtractor(tool, color, strokeWidth)
        is GeometryTool.Compass -> drawCompass(tool, color, strokeWidth)
    }
}

private fun DrawScope.drawRuler(
    ruler: GeometryTool.Ruler,
    color: Color,
    strokeWidth: Float
) {
    val halfLength = ruler.length / 2
    val cos = cos(ruler.rotation)
    val sin = sin(ruler.rotation)

    val start = Offset(
        ruler.position.x - halfLength * cos,
        ruler.position.y - halfLength * sin
    )
    val end = Offset(
        ruler.position.x + halfLength * cos,
        ruler.position.y + halfLength * sin
    )

    // Main ruler line
    drawLine(
        color = color,
        start = start,
        end = end,
        strokeWidth = strokeWidth
    )

    // Measurement marks
    val markCount = 20 // Every 1.5cm
    for (i in 0..markCount) {
        val t = i.toFloat() / markCount
        val markPos = Offset(
            start.x + (end.x - start.x) * t,
            start.y + (end.y - start.y) * t
        )

        val markHeight = if (i % 5 == 0) 15f else 8f
        val perpX = -sin * markHeight / 2
        val perpY = cos * markHeight / 2

        drawLine(
            color = color,
            start = Offset(markPos.x - perpX, markPos.y - perpY),
            end = Offset(markPos.x + perpX, markPos.y + perpY),
            strokeWidth = 1f
        )
    }
}

private fun DrawScope.drawSetSquare(
    setSquare: GeometryTool.SetSquare,
    color: Color,
    strokeWidth: Float
) {
    val vertices = when (setSquare.type) {
        SetSquareType.TRIANGLE_45 -> calculate45TriangleVertices(setSquare)
        SetSquareType.TRIANGLE_30_60 -> calculate30_60TriangleVertices(setSquare)
    }

    val path = Path().apply {
        moveTo(vertices[0].x, vertices[0].y)
        vertices.forEach { vertex ->
            lineTo(vertex.x, vertex.y)
        }
        close()
    }

    drawPath(
        path = path,
        color = color.copy(alpha = 0.3f)
    )

    drawPath(
        path = path,
        color = color,
        style = Stroke(width = strokeWidth)
    )
}

private fun DrawScope.drawProtractor(
    protractor: GeometryTool.Protractor,
    color: Color,
    strokeWidth: Float
) {
    // Draw main arc
    drawArc(
        color = color,
        startAngle = protractor.startAngle,
        sweepAngle = protractor.sweepAngle,
        useCenter = false,
        topLeft = Offset(
            protractor.position.x - protractor.radius,
            protractor.position.y - protractor.radius
        ),
        size = androidx.compose.ui.geometry.Size(
            protractor.radius * 2,
            protractor.radius * 2
        ),
        style = Stroke(width = strokeWidth)
    )

    // Draw angle marks
    for (angle in 0..180 step 10) {
        val radians = Math.toRadians(angle.toDouble()).toFloat()
        val innerRadius = protractor.radius - 10f
        val outerRadius = protractor.radius

        val startPoint = Offset(
            protractor.position.x + innerRadius * cos(radians),
            protractor.position.y + innerRadius * sin(radians)
        )
        val endPoint = Offset(
            protractor.position.x + outerRadius * cos(radians),
            protractor.position.y + outerRadius * sin(radians)
        )

        drawLine(
            color = color,
            start = startPoint,
            end = endPoint,
            strokeWidth = 1f
        )
    }

    // Draw center point
    drawCircle(
        color = color,
        radius = 3f,
        center = protractor.position
    )
}
```

```
private fun DrawScope.drawCompass(
    compass: GeometryTool.Compass,
    color: Color,
    strokeWidth: Float
) {
    // Draw compass circle
    drawCircle(
        color = color,
        radius = compass.radius,
        center = compass.center,
        style = Stroke(width = strokeWidth)
    )

    // Draw center point
    drawCircle(
        color = color,
        radius = 3f,
        center = compass.center
    )

    // Draw radius line
    drawLine(
        color = color,
        start = compass.center,
        end = Offset(
            compass.center.x + compass.radius,
            compass.center.y
        ),
        strokeWidth = 1f
    )
}

private fun DrawScope.drawSnapIndicator(snapPoint: SnapPoint) {
    val color = when (snapPoint.type) {
        SnapType.GRID -> Color.Blue
        SnapType.ENDPOINT -> Color.Red
        SnapType.MIDPOINT -> Color.Green
        SnapType.INTERSECTION -> Color.Yellow
        SnapType.ANGLE -> Color.Magenta
        SnapType.CIRCLE_CENTER -> Color.Cyan
    }

    val alpha = (0.5f + snapPoint.strength * 0.5f)

    // Draw snap indicator circle
    drawCircle(
        color = color.copy(alpha = alpha),
        radius = 8f,
        center = snapPoint.position,
        style = Stroke(width = 2f)
    )

    // Draw inner dot
    drawCircle(
        color = color.copy(alpha = alpha),
        radius = 2f,
        center = snapPoint.position
    )
}

private fun calculate45TriangleVertices(setSquare: GeometryTool.SetSquare): List<Offset> {
    val size = setSquare.size
    val cos = cos(setSquare.rotation)
    val sin = sin(setSquare.rotation)

    val vertices = listOf(
        Offset(0f, 0f),
        Offset(size, 0f),
        Offset(size, size)
    )

    return vertices.map
```