WetInk Next

Исходный код P0-P7 - актуальная редакция

Фактическое состояние

P0-P6: базовый Android-проект, GL canvas, ввод, стабилизация для stamp-кисти, preview/commit, слои и tile-based Undo/Redo. Дополнительно создан текущий условный Compose UI: панель слоёв, Undo/Redo, размер и непрозрачность.

P7 Ribbon/G-Pen: добавлены GPU-рендер, disk-sweep триангуляция, AA-юбка, variable-width input и режим RIBBON. В текущем коде есть обработка короткой открытой траектории как круглого диска, UP-сэмпл при завершении штриха, диагностический лог решения о замыкании и pressure-filter для стилуса. P7 остаётся в работе: необходимо герметизировать стыки mesh и убрать редкие белые точки на длинных линиях.

План далее: завершить P7 - непрерывная topology/AA и проверка на планшете; затем P8 - texture/pencil/grain; P9 - marker/airbrush; P10 - selection.

app/src/main/java/com/wetinknext/app/EditorScreen.kt

```
package com.wetinknext.app

import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.heightIn
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.width
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material3.Checkbox
import androidx.compose.material3.Slider
import androidx.compose.material3.Text
import androidx.compose.material3.TextButton
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.unit.dp
import androidx.compose.ui.viewinterop.AndroidView
import com.wetinknext.engine.core.EditorUiState
import com.wetinknext.engine.core.LayerUiModel
import com.wetinknext.engine.core.PaintSurfaceView
import com.wetinknext.ui.theme.WetInkNextPalette

/** P7 shell: Compose reads immutable engine snapshots and posts commands to PaintSurfaceView. */
@Composable
fun EditorScreen() {
    val palette = WetInkNextPalette.default
    var uiState by remember { mutableStateOf(EditorUiState.empty) }
    var surface by remember { mutableStateOf<PaintSurfaceView?>(null) }

    Box(
        modifier = Modifier
            .fillMaxSize()
            .background(palette.appBg),
    ) {
        AndroidView(
            modifier = Modifier.fillMaxSize(),
            factory = { context ->
                PaintSurfaceView(context).also { view ->
                    view.onEditorStateChange = { uiState = it }
                    surface = view
                    view.requestState()
                }
            },
        )

        EditorTopBar(
            state = uiState,
            palette = palette,
            modifier = Modifier
                .align(Alignment.TopCenter)
                .padding(top = 12.dp),
            onUndo = { surface?.undo() },
            onRedo = { surface?.redo() },
        )

        BrushControls(
            state = uiState,
            palette = palette,
            modifier = Modifier
                .align(Alignment.BottomStart)
                .padding(12.dp),
            onBrushSizeCommit = { surface?.setBrushSize(it) },
            onBrushOpacityCommit = { surface?.setBrushOpacity(it) },
        )

        LayersPanel(
            state = uiState,
            palette = palette,
            modifier = Modifier
                .align(Alignment.CenterEnd)
                .padding(end = 12.dp)
                .width(300.dp),
            onSelectLayer = { surface?.setActiveLayer(it) },
```

```
            onAddLayer = { surface?.addLayer() },
            onDeleteLayer = { surface?.removeLayer(it) },
            onLayerVisible = { id, visible -> surface?.setLayerVisible(id, visible) },
            onLayerOpacity = { id, opacity -> surface?.setLayerOpacity(id, opacity) },
        )
    }
}

@Composable
private fun EditorTopBar(
    state: EditorUiState,
    palette: WetInkNextPalette,
    modifier: Modifier = Modifier,
    onUndo: () -> Unit,
    onRedo: () -> Unit,
) {
    Row(
        modifier = modifier
            .background(palette.panelBg.copy(alpha = 0.94f), RoundedCornerShape(24.dp))
            .border(1.dp, palette.panelStroke, RoundedCornerShape(24.dp))
            .padding(horizontal = 8.dp, vertical = 4.dp),
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement = Arrangement.spacedBy(4.dp),
    ) {
        TextButton(onClick = onUndo, enabled = state.canUndo) { Text("Undo") }
        TextButton(onClick = onRedo, enabled = state.canRedo) { Text("Redo") }
    }
}

@Composable
private fun BrushControls(
    state: EditorUiState,
    palette: WetInkNextPalette,
    modifier: Modifier = Modifier,
    onBrushSizeCommit: (Float) -> Unit,
    onBrushOpacityCommit: (Float) -> Unit,
) {
    Column(
        modifier = modifier.panel(palette).width(260.dp).padding(14.dp),
        verticalArrangement = Arrangement.spacedBy(10.dp),
    ) {
        Text(text = "■■■■■", color = palette.textPrimary)
        CommitSlider(
            label = "■■■■■■",
            value = state.brushSizePx,
            range = 2f..400f,
            palette = palette,
            format = { "${it.toInt()} px" },
            onCommit = onBrushSizeCommit,
        )
        CommitSlider(
            label = "■■■■■■■■■■■■■■",
            value = state.brushOpacity,
            range = 0f..1f,
            palette = palette,
            format = { "${(it * 100f).toInt()}%" },
            onCommit = onBrushOpacityCommit,
        )
    }
}

@Composable
private fun LayersPanel(
    state: EditorUiState,
    palette: WetInkNextPalette,
    modifier: Modifier = Modifier,
    onSelectLayer: (Long) -> Unit,
    onAddLayer: () -> Unit,
    onDeleteLayer: (Long) -> Unit,
    onLayerVisible: (Long, Boolean) -> Unit,
    onLayerOpacity: (Long, Float) -> Unit,
) {
    Column(
        modifier = modifier.panel(palette).padding(14.dp),
        verticalArrangement = Arrangement.spacedBy(10.dp),
    ) {
        Row(
            modifier = Modifier.fillMaxWidth(),
            verticalAlignment = Alignment.CenterVertically,
        ) {
            Text(text = "■■■■", color = palette.textPrimary, modifier = Modifier.weight(1f))
            TextButton(onClick = onAddLayer, enabled = state.ready) { Text("■■■■■■■■") }
        }

        LazyColumn(
            modifier = Modifier.heightIn(max = 380.dp),
            verticalArrangement = Arrangement.spacedBy(8.dp),
        ) {
            items(items = state.layers.asReversed(), key = LayerUiModel::id) { layer ->
                LayerRow(
```

```
                    layer = layer,
                    palette = palette,
                    onSelect = { onSelectLayer(layer.id) },
                    onDelete = { onDeleteLayer(layer.id) },
                    onVisible = { visible -> onLayerVisible(layer.id, visible) },
                    onOpacity = { opacity -> onLayerOpacity(layer.id, opacity) },
                )
            }
        }
    }
}

@Composable
private fun LayerRow(
    layer: LayerUiModel,
    palette: WetInkNextPalette,
    onSelect: () -> Unit,
    onDelete: () -> Unit,
    onVisible: (Boolean) -> Unit,
    onOpacity: (Float) -> Unit,
) {
    Column(
        modifier = Modifier
            .fillMaxWidth()
            .clip(RoundedCornerShape(14.dp))
            .background(if (layer.active) palette.accent.copy(alpha = 0.18f) else palette.panelBg.copy(alpha = 0.72f))
            .border(1.dp, if (layer.active) palette.accent else palette.panelStroke, RoundedCornerShape(14.dp))
            .clickable(onClick = onSelect)
            .padding(10.dp),
        verticalArrangement = Arrangement.spacedBy(8.dp),
    ) {
        Row(
            modifier = Modifier.fillMaxWidth(),
            verticalAlignment = Alignment.CenterVertically,
        ) {
            Checkbox(checked = layer.isVisible, onCheckedChange = onVisible)
            Text(text = layer.name, color = palette.textPrimary, modifier = Modifier.weight(1f))
            if (layer.canDelete) TextButton(onClick = onDelete) { Text("[redacted]") }
        }
        if (layer.active) {
            CommitSlider(
                label = "[redacted]",
                value = layer.opacity,
                range = 0f..1f,
                palette = palette,
                format = { "${(it * 100f).toInt()}%" },
                onCommit = onOpacity,
            )
        }
    }
}

@Composable
private fun CommitSlider(
    label: String,
    value: Float,
    range: ClosedFloatingPointRange<Float>,
    palette: WetInkNextPalette,
    format: (Float) -> String,
    onCommit: (Float) -> Unit,
) {
    var localValue by remember(value) { mutableStateOf(value) }
    Column(modifier = Modifier.fillMaxWidth(), verticalArrangement = Arrangement.spacedBy(4.dp)) {
        Row(modifier = Modifier.fillMaxWidth(), verticalAlignment = Alignment.CenterVertically) {
            Text(text = label, color = palette.textSecondary, modifier = Modifier.weight(1f))
            Text(text = format(localValue), color = palette.textPrimary)
        }
        Slider(
            value = localValue,
            onValueChange = { localValue = it },
            onValueChangeFinished = { onCommit(localValue) },
            valueRange = range,
        )
    }
}

private fun Modifier.panel(palette: WetInkNextPalette): Modifier =
    background(palette.panelBg.copy(alpha = 0.94f), RoundedCornerShape(24.dp))
        .border(1.dp, palette.panelStroke, RoundedCornerShape(24.dp))
```

app/src/main/java/com/wetinknext/app/MainActivity.kt

```
package com.wetinknext.app

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent { EditorScreen() }
    }
}
```

app/src/main/java/com/wetinknext/engine/brush/BrushSettings.kt

```
package com.wetinknext.engine.brush

enum class BrushRenderMode { STAMP, RIBBON, WET }
enum class BlendPolicy { NORMAL_BUILDUP, NON_BUILDUP }
enum class RibbonCap { ROUND, BUTT }
enum class RibbonJoin { MITER, ROUND, BEVEL }
data class RibbonSettings(
    val minWidthRatio: Float = 0.06f,
    val taperStartPx: Float = 8f,
    val taperEndPx: Float = 14f,
    val miterLimit: Float = 3f,
    val aaWidthPx: Float = 1f,
    val cap: RibbonCap = RibbonCap.ROUND,
    val join: RibbonJoin = RibbonJoin.ROUND,
    val minPointDistancePx: Float = 0.5f,
)
data class WetSettings(val wetness: Float = 0f, val spread: Float = 0f, val bleed: Float = 0f)
data class BrushSettings(
    val name: String = "Debug Stamp", val category: String = "Debug", val renderMode: BrushRenderMode =
BrushRenderMode.STAMP,
    val baseRadiusPx: Float = 14f, val opacity: Float = 1f, val flow: Float = 1f, val spacing: Float = 0.16f,
    val spacingUsesDiameter: Boolean = true, val hardness: Float = 1f, val colorArgb: Long = 0xFF000000L,
    val smoothing: Float = 0f, val streamline: Float = 0f, val antiAliasLevel: Int = 2, val useTempStrokeBuffer:
Boolean = false,
    val pressureToSize: Boolean = true, val pressureToOpacity: Boolean = true, val pressureGamma: Float = 1f,
    val minSizeRatio: Float = 0.05f, val velocityToSize: Float = 0f, val velocityToOpacity: Float = 0f,
    val tiltToSize: Float = 0f, val tiltToOpacity: Float = 0f, val tiltToRotation: Float = 0f,
    val rotationJitter: Float = 0f, val sizeJitter: Float = 0f, val opacityJitter: Float = 0f, val scatter: Float =
0f,
    val shapeAssetPath: String? = null, val grainAssetPath: String? = null, val grainScale: Float = 1f,
    val grainCanvasLocked: Boolean = true, val rgbToAlpha: Boolean = false, val textureContrast: Float = 1f,
    val textureDepth: Float = 1f, val pixelPen: Boolean = false, val squareStroke: Boolean = false,
    val noAntialias: Boolean = false, val blendPolicy: BlendPolicy = BlendPolicy.NORMAL_BUILDUP,
    val ribbon: RibbonSettings = RibbonSettings(), val wet: WetSettings = WetSettings(),
)
```

app/src/main/java/com/wetinknext/engine/brush/CapsuleEmitter.kt

```
package com.wetinknext.engine.brush

import com.wetinknext.engine.input.InputBatch
import kotlin.math.hypot
import kotlin.math.max
import kotlin.math.pow

/**
 * [redacted] InputSample [redacted] [redacted].
 *
 * [redacted]:
 *   A = x0, y0, radius0
 *   B = x1, y1, radius1
 *
 * [redacted] [redacted] [redacted] [redacted] [redacted] capsule.
 * [redacted] [redacted] [redacted] shader [redacted] round-cone.
 *
 * [redacted]:
 * - [redacted] [redacted] InputSample;
 * - [redacted] [redacted] List/FloatArray [redacted] MOVE;
 * - reusable state [redacted] GL/render pipeline;
 * - pointerId [redacted] [redacted] DOWN;
 * - [redacted] [redacted] [redacted] [redacted];
 * - opacity [redacted] [redacted] [redacted] [redacted], [redacted] [redacted] [redacted] blit-pass.
 */
class CapsuleEmitter(
    private var settings: BrushSettings,
) {
    private val stabilizer = Stabilizer()
    private val pressureFilter = PressureFilter()

    private var active = false
    private var pointerId = -1

    private var hasLastPoint = false
    private var lastX = 0f
    private var lastY = 0f
    private var lastRadius = 0f

    // [redacted] [redacted] [redacted] [redacted] P0, P1, P2, P3
    private var p0x = 0f; private var p0y = 0f; private var p0r = 0f
    private var p1x = 0f; private var p1y = 0f; private var p1r = 0f
    private var p2x = 0f; private var p2y = 0f; private var p2r = 0f
    private var p3x = 0f; private var p3y = 0f; private var p3r = 0f
    private var pointsInWindow = 0

    private var emittedSegments = 0

    var hasStroke: Boolean = false
        private set

    var totalLength: Float = 0f
        private set

    var minX: Float = 0f
        private set

    var minY: Float = 0f
        private set

    var maxX: Float = 0f
        private set

    var maxY: Float = 0f
        private set

    var hasBounds: Boolean = false
        private set

    fun updateSettings(value: BrushSettings) {
        settings = value
    }

    /**
     * [redacted] [redacted] [redacted] stroke.
     * overflowCount [redacted] [redacted] [redacted] [redacted].
     */
    fun reset() {
        active = false
        pointerId = -1

        hasLastPoint = false
        lastX = 0f
        lastY = 0f
        lastRadius = 0f

        pointsInWindow = 0
```

```
    emittedSegments = 0
    hasStroke = false
    totalLength = 0f

    minX = 0f
    minY = 0f
    maxX = 0f
    maxY = 0f
    hasBounds = false

    stabilizer.reset()
    pressureFilter.reset()
}

/**
 * ■■■■■■■ stroke.
 *
 * ■■■■■■■ sample ■■■■■■■■■■■■ ■■■ ■■■■■■■■■■■■ ■■■■■■■■.
 * ■ fragment shader ■■■■ ■■■■■■■■■■■■■ ■ ■■■■■■■ ■■■■■.
 */
fun begin(
    batch: InputBatch,
    out: CapsuleStrokeRenderer,
) {
    reset()
    out.beginStroke()

    if (batch.sampleCount <= 0) return

    val sample = batch.samples[0]

    active = true
    pointerId = sample.pointerId

    stabilizer.strength = (
        settings.smoothing * SMOOTHING_WEIGHT +
            settings.streamline * STREAMLINE_WEIGHT
        ).coerceIn(0f, 1f)

    stabilizer.process(
        timestampNanos = sample.timestampNanos,
        rawX = sample.canvasX,
        rawY = sample.canvasY,
    )

    val x = stabilizer.x
    val y = stabilizer.y
    val filteredPressure = pressureFilter.filter(sample.timestampNanos, sample.pressure)
    val radius = radiusForPressure(filteredPressure)

    lastX = x
    lastY = y
    lastRadius = radius
    hasLastPoint = true
    hasStroke = true

    // ■■■■■■■■■■■■■■ ■■■■■
    p0x = x; p0y = y; p0r = radius
    p1x = x; p1y = y; p1r = radius
    p2x = x; p2y = y; p2r = radius
    pointsInWindow = 1

    // ■■■■■■■■■■■■ ■■■■■■■■: ■■■■■■■■ dab/tap.
    out.addSegment(
        x0 = x,
        y0 = y,
        radius0 = radius,
        x1 = x,
        y1 = y,
        radius1 = radius,
    )
    emittedSegments++

    includeBounds(x, y, radius)
}

/**
 * ■■■■■■■■■■■■■■ MOVE batch.
 *
 * Historical samples ■■■■ ■■■■■■■ ■■■■■■■■■■■■ ■■■■■■■ batch
 * ■■■■■■■ ■■■■■■■■■■ sample, ■■■■ ■■■■■■■■■■■■■■ StrokeInputCapturer.
 */
fun append(
    batch: InputBatch,
    out: CapsuleStrokeRenderer,
) {
    if (!active || batch.sampleCount <= 0) return

    for (index in 0 until batch.sampleCount) {
        val sample = batch.samples[index]
```

```
        if (sample.pointerId != pointerId) {
            continue
        }

        stabilizer.process(
            timestampNanos = sample.timestampNanos,
            rawX = sample.canvasX,
            rawY = sample.canvasY,
        )

        val x = stabilizer.x
        val y = stabilizer.y
        val filteredPressure = pressureFilter.filter(sample.timestampNanos, sample.pressure)
        val radius = radiusForPressure(filteredPressure)

        emitPoint(
            x = x,
            y = y,
            radius = radius,
            out = out,
        )
    }
}

/**
 * ■■■■■■■■■■ stroke.
 *
 * ■■■■■■■■■■ ■■■■■■■■ sample ■■■■■■■ ■■■■■ ■■■■■■■■ ■■■■■■ append()
 * ■■ ■■■■■■■ finish().
 */
fun finish(
    out: CapsuleStrokeRenderer,
    cancel: Boolean,
) {
    if (!active) return

    if (cancel) {
        reset()
        out.clearStrokeData()
        return
    }

    // ■■■■■■■■■■■■■■■ ■■■■■■■■■■ ■■■■■■■■, ■■■■■■■■■■■ ■■■■■■■■■■■ ■■■■■■ ■■■■ P3
    if (pointsInWindow >= 2 && settings.smoothing > 1e-4f) {
        interpolate(p0x, p0y, p0r, p1x, p1y, p1r, p2x, p2y, p2r, p2x, p2y, p2r, out)
    }

    active = false
    pointerId = -1
}

private fun emitPoint(
    x: Float,
    y: Float,
    radius: Float,
    out: CapsuleStrokeRenderer,
) {
    val dx = x - lastX
    val dy = y - lastY
    val distance = hypot(dx, dy)

    if (distance < minimumPointDistance()) {
        return
    }

    if (settings.smoothing <= 1e-4f) {
        // Raw mode: ■■■■■■■ ■■■■■■■■■ ■■■■ ■■■■■■■■■ ■■ ■■■■■
        out.addSegment(lastX, lastY, lastRadius, x, y, radius)
        emittedSegments++
        totalLength += distance
        includeBounds(lastX, lastY, lastRadius)
        includeBounds(x, y, radius)
    } else {
        // Interpolated mode: Catmull-Rom ■■■■■■ ■■■■■
        when (pointsInWindow) {
            1 -> {
                p2x = x; p2y = y; p2r = radius
                pointsInWindow = 2
            }
            2 -> {
                p3x = x; p3y = y; p3r = radius
                interpolate(p0x, p0y, p0r, p1x, p1y, p1r, p2x, p2y, p2r, p3x, p3y, p3r, out)
                // ■■■■■■ ■■■■■
                p0x = p1x; p0y = p1y; p0r = p1r
                p1x = p2x; p1y = p2y; p1r = p2r
                p2x = p3x; p2y = p3y; p2r = p3r
            }
        }
    }
    hasStroke = true
```

```
    lastX = x
    lastY = y
    lastRadius = radius
}

/**
 * ########### ############## ######### P1-P2 # ################ P0 # P3 ### ############# ######.
 */
private fun interpolate(
    p0x: Float, p0y: Float, p0r: Float,
    p1x: Float, p1y: Float, p1r: Float,
    p2x: Float, p2y: Float, p2r: Float,
    p3x: Float, p3y: Float, p3r: Float,
    out: CapsuleStrokeRenderer
) {
    val dist = hypot(p2x - p1x, p2y - p1y)
    if (dist < 1e-4f) return

    // ########## ############ ###### ### ########### ####### ###### (Catmull-Rom -> Bezier)
    val c1x = p1x + (p2x - p0x) / 6f
    val c1y = p1y + (p2y - p0y) / 6f
    val c2x = p2x - (p3x - p1x) / 6f
    val c2y = p2y - (p3y - p1y) / 6f

    // ########### ########### ######: ######### ####### 0.25..0.5 ########
    val avgRadius = (p1r + p2r) * 0.5f
    val stepPx = max(1f, avgRadius * INTERPOLATION_STEP_RATIO)
    val steps = max(1, (dist / stepPx).toInt().coerceAtMost(MAX_INTERPOLATION_STEPS))

    var prevX = p1x
    var prevY = p1y
    var prevR = p1r

    for (i in 1..steps) {
        val t = i.toFloat() / steps
        val mt = 1f - t

        // ########### ###### ### ########
        val x = mt * mt * mt * p1x + 3f * mt * mt * t * c1x + 3f * mt * t * t * c2x + t * t * t * p2x
        val y = mt * mt * mt * p1y + 3f * mt * mt * t * c1y + 3f * mt * t * t * c2y + t * t * t * p2y

        // ######### ############## ########
        val r = p1r + (p2r - p1r) * t

        out.addSegment(prevX, prevY, prevR, x, y, r)
        emittedSegments++
        totalLength += hypot(x - prevX, y - prevY)
        includeBounds(x, y, r)

        prevX = x
        prevY = y
        prevR = r
    }
}

/**
 * ####### ######.
 *
 * baseRadiusPx # ######## #### ######### #########,
 * ######## UI-######## ###### ################## # EngineRenderer
 * ## ########## #####.
 */
private fun radiusForPressure(pressure: Float): Float {
    val normalized = pressure.coerceIn(0f, 1f)

    val shapedPressure =
        if (settings.pressureGamma > 0f) {
            normalized.pow(settings.pressureGamma)
        } else {
            normalized
        }

    val sizeFactor =
        if (settings.pressureToSize) {
            settings.minSizeRatio +
                (1f - settings.minSizeRatio) * shapedPressure
        } else {
            1f
        }

    return (
        settings.baseRadiusPx * sizeFactor
    ).coerceAtLeast(MIN_RADIUS_PX)
}

/**
 * ### ####### ## ###### ###### ######## ###, ### ### stamp-######:
 * segment shader ### ########## #### ######## ###### A # B.
 */
private fun minimumPointDistance(): Float {
```

```
        val configured = settings.ribbon.minPointDistancePx
            .coerceIn(MIN_POINT_DISTANCE_MIN, MAX_POINT_DISTANCE)

        val radiusBased = settings.baseRadiusPx * POINT_STEP_RADIUS_RATIO

        return max(
            configured,
            radiusBased.coerceAtMost(MAX_POINT_DISTANCE),
        )
    }

    private fun includeBounds(
        x: Float,
        y: Float,
        radius: Float,
    ) {
        val left = x - radius
        val top = y - radius
        val right = x + radius
        val bottom = y + radius

        if (!hasBounds) {
            minX = left
            minY = top
            maxX = right
            maxY = bottom
            hasBounds = true
            return
        }

        if (left < minX) minX = left
        if (top < minY) minY = top
        if (right > maxX) maxX = right
        if (bottom > maxY) maxY = bottom
    }

    companion object {
        private const val SMOOTHING_WEIGHT = 0.7f
        private const val STREAMLINE_WEIGHT = 0.3f

        private const val MIN_RADIUS_PX = 0.25f

        private const val MIN_POINT_DISTANCE_MIN = 0.05f
        private const val MAX_POINT_DISTANCE = 32f
        private const val POINT_STEP_RADIUS_RATIO = 0.08f

        private const val INTERPOLATION_STEP_RATIO = 0.4f
        private const val MAX_INTERPOLATION_STEPS = 64
    }
}
```

app/src/main/java/com/wetinknext/engine/brush/CapsuleStrokeRenderer.kt

```
package com.wetinknext.engine.brush

import android.opengl.GLES30
import com.wetinknext.engine.gl.GlProgram
import com.wetinknext.engine.gl.RenderTarget
import com.wetinknext.engine.gl.ShaderLib
import java.nio.ByteBuffer
import java.nio.ByteOrder

/**
 * GPU-renderer [redacted].
 *
 * [redacted] instance:
 *   x0, y0, radius0,
 *   x1, y1, radius1
 *
 * [redacted] AABB, [redacted]
 * [redacted] capsuleFragment [redacted] signed distance.
 *
 * [redacted]:
 * - target [redacted];
 * - [redacted] GL_MAX;
 * - [redacted] strokeTarget;
 * - [redacted] shader: premultiplied linear RGBA.
 */
class CapsuleStrokeRenderer(
    private val maxSegments: Int = DEFAULT_MAX_SEGMENTS,
) {
    private var program: GlProgram? = null

    private var vaoId = 0
    private var cornerVboId = 0
    private var instanceVboId = 0

    private var uCanvasToClip = -1
    private var uColorLinear = -1

    /**
     * [redacted]:
     * x0, y0, r0, x1, y1, r1
     */
    private val instanceData =
        ByteBuffer
            .allocateDirect(
                maxSegments * FLOATS_PER_SEGMENT * Float.SIZE_BYTES,
            )
            .order(ByteOrder.nativeOrder())
            .asFloatBuffer()

    var segmentCount: Int = 0
        private set

    private var uploadedSegmentCount = 0

    /**
     * [redacted] stroke.
     * [redacted].
     */
    var overflowCount: Long = 0L
        private set

    val isEmpty: Boolean
        get() = segmentCount == 0

    val hasPendingSegments: Boolean
        get() = segmentCount > uploadedSegmentCount

    fun create() {
        checkOnGlThreadOnly()

        release()

        program = GlProgram(
            vertexSource = ShaderLib.capsuleVertex,
            fragmentSource = ShaderLib.capsuleFragment,
        )

        val currentProgram = checkNotNull(program)
        currentProgram.use()

        uCanvasToClip = GLES30.glGetUniformLocation(
            currentProgram.id,
            "uCanvasToClip",
        )
        uColorLinear = GLES30.glGetUniformLocation(
```

```
    currentProgram.id,
    "uColorLinear",
)

check(uCanvasToClip >= 0) {
    "Capsule shader uniform uCanvasToClip is missing"
}
check(uColorLinear >= 0) {
    "Capsule shader uniform uColorLinear is missing"
}

val corners = floatArrayOf(
    -1f, -1f,
     1f, -1f,
    -1f,  1f,
     1f,  1f,
)

val cornerData =
    ByteBuffer
        .allocateDirect(corners.size * Float.SIZE_BYTES)
        .order(ByteOrder.nativeOrder())
        .asFloatBuffer()

cornerData.put(corners)
cornerData.position(0)

val ids = IntArray(1)

GLES30.glGenVertexArrays(1, ids, 0)
vaoId = ids[0]

GLES30.glGenBuffers(1, ids, 0)
cornerVboId = ids[0]

GLES30.glGenBuffers(1, ids, 0)
instanceVboId = ids[0]

check(vaoId != 0) { "Failed to create capsule VAO" }
check(cornerVboId != 0) { "Failed to create capsule corner VBO" }
check(instanceVboId != 0) { "Failed to create capsule instance VBO" }

GLES30.glBindVertexArray(vaoId)

// ■■■■ AABB-■■■■■.
GLES30.glBindBuffer(
    GLES30.GL_ARRAY_BUFFER,
    cornerVboId,
)
GLES30.glBufferData(
    GLES30.GL_ARRAY_BUFFER,
    corners.size * Float.SIZE_BYTES,
    cornerData,
    GLES30.GL_STATIC_DRAW,
)
GLES30.glEnableVertexAttribArray(ATTR_CORNER)
GLES30.glVertexAttribPointer(
    ATTR_CORNER,
    2,
    GLES30.GL_FLOAT,
    false,
    2 * Float.SIZE_BYTES,
    0,
)

// ■■■■■■ ■■■■■■■■■■.
GLES30.glBindBuffer(
    GLES30.GL_ARRAY_BUFFER,
    instanceVboId,
)
GLES30.glBufferData(
    GLES30.GL_ARRAY_BUFFER,
    maxSegments * STRIDE_BYTES,
    null,
    GLES30.GL_DYNAMIC_DRAW,
)

// A: x0, y0, radius0
GLES30.glEnableVertexAttribArray(ATTR_SEGMENT_A)
GLES30.glVertexAttribPointer(
    ATTR_SEGMENT_A,
    3,
    GLES30.GL_FLOAT,
    false,
    STRIDE_BYTES,
    0,
)
GLES30.glVertexAttribDivisor(
    ATTR_SEGMENT_A,
    1,
```

```
    )

    // B: x1, y1, radius1
    GLES30.glEnableVertexAttribArray(ATTR_SEGMENT_B)
    GLES30.glVertexAttribPointer(
        ATTR_SEGMENT_B,
        3,
        GLES30.GL_FLOAT,
        false,
        STRIDE_BYTES,
        3 * Float.SIZE_BYTES,
    )
    GLES30.glVertexAttribDivisor(
        ATTR_SEGMENT_B,
        1,
    )

    GLES30.glBindBuffer(
        GLES30.GL_ARRAY_BUFFER,
        0,
    )
    GLES30.glBindVertexArray(0)

    GLES30.glUseProgram(0)

    checkNoGlError("CapsuleStrokeRenderer.create")
}

/**
 * ********* ****** stroke.
 *
 * ******* target ******* ***** ******* *********:
 * strokeTarget.clear(0f, 0f, 0f, 0f)
 */
fun beginStroke() {
    segmentCount = 0
    uploadedSegmentCount = 0

    instanceData.clear()
    instanceData.limit(
        maxSegments * FLOATS_PER_SEGMENT,
    )
}

/**
 * ********** ***** ******** * CPU-buffer.
 *
 * ***** capacity **********, ******** ** ********* ******:
 * overflowCount **************.
 */
fun addSegment(
    x0: Float,
    y0: Float,
    radius0: Float,
    x1: Float,
    y1: Float,
    radius1: Float,
): Boolean {
    if (segmentCount >= maxSegments) {
        overflowCount++
        return false
    }

    val offset = segmentCount * FLOATS_PER_SEGMENT

    instanceData.put(offset, x0)
    instanceData.put(offset + 1, y0)
    instanceData.put(offset + 2, radius0.coerceAtLeast(0.01f))

    instanceData.put(offset + 3, x1)
    instanceData.put(offset + 4, y1)
    instanceData.put(offset + 5, radius1.coerceAtLeast(0.01f))

    segmentCount++
    return true
}

/**
 * ********** * ******* ******* ****** *********.
 *
 * ************** *** preview ** ****** MOVE.
 * ****** ******* ******** target ******* ***** *******.
 */
fun drawPending(
    target: RenderTarget,
    width: Int,
    height: Int,
    canvasToClip: FloatArray,
    colorLinear: FloatArray,
) {
```

```
    val first = uploadedSegmentCount
    val count = segmentCount - uploadedSegmentCount

    if (count <= 0) return

    val drawn = drawRange(
        target = target,
        width = width,
        height = height,
        canvasToClip = canvasToClip,
        colorLinear = colorLinear,
        firstSegment = first,
        count = count,
    )

    if (drawn) {
        uploadedSegmentCount = segmentCount
    }
}

/**
 * ********** ****** *** ************ *********.
 *
 * ************** *** commit, **** target *** ******.
 */
fun drawAll(
    target: RenderTarget,
    width: Int,
    height: Int,
    canvasToClip: FloatArray,
    colorLinear: FloatArray,
) {
    if (segmentCount <= 0) return

    drawRange(
        target = target,
        width = width,
        height = height,
        canvasToClip = canvasToClip,
        colorLinear = colorLinear,
        firstSegment = 0,
        count = segmentCount,
    )
}

/**
 * ********** **************** ********* * ********** [firstSegment, firstSegment + count).
 *
 * ******* *******: glVertexAttribPointer *********** ***** *********,
 * ******* *** offset * instanced-********** ********* *** ******** **********.
 */
private fun drawRange(
    target: RenderTarget,
    width: Int,
    height: Int,
    canvasToClip: FloatArray,
    colorLinear: FloatArray,
    firstSegment: Int,
    count: Int,
): Boolean {
    val currentProgram = program ?: return false

    require(canvasToClip.size >= 16) {
        "canvasToClip must contain at least 16 floats"
    }
    require(colorLinear.size >= 3) {
        "colorLinear must contain at least 3 floats"
    }
    require(firstSegment >= 0)
    require(count >= 0)
    require(firstSegment + count <= segmentCount)

    if (count == 0) return true

    target.bind()

    GLES30.glViewport(
        0,
        0,
        width,
        height,
    )

    GLES30.glDisable(GLES30.GL_SCISSOR_TEST)

    /*
     * Shader output:
     *   rgb = linearColor * coverage
     *   a   = coverage
     *
```

```
 * GL_MAX [redacted] [redacted] [redacted] [redacted].
 * [redacted] [redacted] [redacted] [redacted] [redacted] [redacted] [redacted] [redacted] [redacted] [redacted].
 */
GLES30.glEnable(GLES30.GL_BLEND)
GLES30.glBlendEquation(GLES30.GL_MAX)
GLES30.glBlendFunc(
    GLES30.GL_ONE,
    GLES30.GL_ONE,
)

currentProgram.use()

GLES30.glUniformMatrix4fv(
    uCanvasToClip,
    1,
    false,
    canvasToClip,
    0,
)

GLES30.glUniform3f(
    uColorLinear,
    colorLinear[0].coerceIn(0f, 1f),
    colorLinear[1].coerceIn(0f, 1f),
    colorLinear[2].coerceIn(0f, 1f),
)

/*
 * FloatBuffer [redacted] [redacted] [redacted] [redacted] [redacted].
 * glBufferSubData [redacted] remaining() [redacted].
 */
val sourceStart = firstSegment * FLOATS_PER_SEGMENT
val sourceEnd =
    sourceStart + count * FLOATS_PER_SEGMENT

instanceData.position(sourceStart)
instanceData.limit(sourceEnd)

GLES30.glBindVertexArray(vaoId)
GLES30.glBindBuffer(
    GLES30.GL_ARRAY_BUFFER,
    instanceVboId,
)

GLES30.glBufferSubData(
    GLES30.GL_ARRAY_BUFFER,
    firstSegment * STRIDE_BYTES,
    count * STRIDE_BYTES,
    instanceData,
)

/*
 * [redacted] ES 3.0 [redacted] glDrawArraysInstancedBaseInstance.
 * [redacted] offset [redacted] [redacted] [redacted].
 *
 * VAO [redacted] [redacted] [redacted] [redacted] [redacted] glVertexAttribPointer.
 */
GLES30.glVertexAttribPointer(
    ATTR_SEGMENT_A,
    3,
    GLES30.GL_FLOAT,
    false,
    STRIDE_BYTES,
    firstSegment * STRIDE_BYTES,
)

GLES30.glVertexAttribPointer(
    ATTR_SEGMENT_B,
    3,
    GLES30.GL_FLOAT,
    false,
    STRIDE_BYTES,
    firstSegment * STRIDE_BYTES + 3 * Float.SIZE_BYTES,
)

GLES30.glDrawArraysInstanced(
    GLES30.GL_TRIANGLE_STRIP,
    0,
    CORNER_VERTEX_COUNT,
    count,
)

GLES30.glBindBuffer(
    GLES30.GL_ARRAY_BUFFER,
    0,
)
GLES30.glBindVertexArray(0)

/*
 * [redacted] [redacted] [redacted]:
```

```
     * Compositor [redacted] FUNC_ADD.
     */
    GLES30.glBlendEquation(GLES30.GL_FUNC_ADD)
    GLES30.glBlendFunc(
        GLES30.GL_ONE,
        GLES30.GL_ONE_MINUS_SRC_ALPHA,
    )
    GLES30.glDisable(GLES30.GL_BLEND)

    GLES30.glBindFramebuffer(
        GLES30.GL_FRAMEBUFFER,
        0,
    )

    instanceData.clear()
    instanceData.limit(
        maxSegments * FLOATS_PER_SEGMENT,
    )

    checkNoGlError("CapsuleStrokeRenderer.drawRange")
    return true
}

/**
 * [redacted] GPU/CPU [redacted] renderer.
 * [redacted] overflow [redacted].
 */
fun clearStrokeData() {
    segmentCount = 0
    uploadedSegmentCount = 0
    instanceData.clear()
    instanceData.limit(
        maxSegments * FLOATS_PER_SEGMENT,
    )
}

/**
 * [redacted] GL-[redacted] [redacted] [redacted] [redacted] GL-[redacted].
 */
fun release() {
    program?.release()
    program = null

    if (instanceVboId != 0) {
        GLES30.glDeleteBuffers(
            1,
            intArrayOf(instanceVboId),
            0,
        )
    }

    if (cornerVboId != 0) {
        GLES30.glDeleteBuffers(
            1,
            intArrayOf(cornerVboId),
            0,
        )
    }

    if (vaoId != 0) {
        GLES30.glDeleteVertexArrays(
            1,
            intArrayOf(vaoId),
            0,
        )
    }

    vaoId = 0
    cornerVboId = 0
    instanceVboId = 0

    uCanvasToClip = -1
    uColorLinear = -1

    segmentCount = 0
    uploadedSegmentCount = 0

    instanceData.clear()
    instanceData.limit(
        maxSegments * FLOATS_PER_SEGMENT,
    )
}

private fun checkOnGlThreadOnly() {
    // [redacted] [redacted]-[redacted].
    // create/release [redacted] onSurfaceCreated/onDrawFrame.
}

private fun checkNoGlError(label: String) {
    if (!CHECK_GL_ERRORS) return
```

```
        val error = GLES30.glGetError()
        check(error == GLES30.GL_NO_ERROR) {
            "$label: GL error 0x${error.toString(16)}"
        }
    }

    companion object {
        private const val CHECK_GL_ERRORS = true

        private const val ATTR_CORNER = 0
        private const val ATTR_SEGMENT_A = 1
        private const val ATTR_SEGMENT_B = 2

        private const val CORNER_VERTEX_COUNT = 4

        const val FLOATS_PER_SEGMENT = 6
        const val STRIDE_BYTES =
            FLOATS_PER_SEGMENT * Float.SIZE_BYTES

        const val DEFAULT_MAX_SEGMENTS = 16_384
    }
}
```

app/src/main/java/com/wetinknext/engine/brush/ClosureDebug.kt

```
package com.wetinknext.engine.brush

import android.util.Log
import com.wetinknext.BuildConfig

/** One debug log per finished stroke; never used from the render hot path. */
object ClosureDebug {
    const val TAG = "RibbonClosure"
    fun publish(distance: Float, threshold: Float, closed: Boolean, samples: Int) {
        if (!BuildConfig.DEBUG) return
        runCatching {
            Log.d(TAG, "dist=%.2f thr=%.2f closed=%s n=%d".format(distance, threshold, closed, samples))
        }
    }
}
```

app/src/main/java/com/wetinknext/engine/brush/ColorSpaces.kt

```
package com.wetinknext.engine.brush

import kotlin.math.pow

object ColorSpaces {
    fun srgbToLinear(value: Float): Float { val v=value.coerceIn(0f,1f); return if(v<=.04045f)v/12.92f else ((v+.
055f)/1.055f).pow(2.4f) }
    fun linearToSrgb(value: Float): Float { val v=value.coerceIn(0f,1f); return if(v<=.0031308f)v*12.92f else 1.
055f*v.pow(1f/2.4f)-.055f }
    fun srgb8ToLinear(argb: Long,out:FloatArray){require(out.size>=3);out[0]=srgbToLinear(((argb shr 16)and 0xFF)/
255f);out[1]=srgbToLinear(((argb shr 8)and 0xFF)/255f);out[2]=srgbToLinear((argb and 0xFF)/255f)}
}
```

app/src/main/java/com/wetinknext/engine/brush/DabBuffer.kt

```
package com.wetinknext.engine.brush

import java.nio.ByteBuffer
import java.nio.ByteOrder

class DabBuffer(val capacity: Int = DEFAULT_CAPACITY) {
    init { require(capacity > 0) }
    val floats = ByteBuffer.allocateDirect(capacity * FLOATS_PER_DAB * Float.SIZE_BYTES).order(ByteOrder.nativeOrder(
)).asFloatBuffer()
    var count = 0; private set; var overflowCount = 0L; private set
    fun clear() { count = 0; overflowCount = 0L; floats.clear() }
    fun add(x: Float, y: Float, radius: Float, rotation: Float, alpha: Float): Boolean { if (count >= capacity) {
overflowCount++; return false }; floats.put(x); floats.put(y); floats.put(radius); floats.put(rotation); floats.put(
alpha); count++; return true }
    fun prepareForUpload() { floats.position(0); floats.limit(count * FLOATS_PER_DAB) }
    /** Restores the direct buffer for appending after a GL upload. */
    fun finishUpload() {
        floats.limit(capacity * FLOATS_PER_DAB)
        floats.position(count * FLOATS_PER_DAB)
    }
    companion object { const val FLOATS_PER_DAB = 5; const val DEFAULT_CAPACITY = 8192 }
}
```

app/src/main/java/com/wetinknext/engine/brush/DabRenderer.kt

```
package com.wetinknext.engine.brush

import android.opengl.GLES30
import com.wetinknext.engine.gl.GlProgram
import com.wetinknext.engine.gl.RenderTarget
import com.wetinknext.engine.gl.ShaderLib
import java.nio.ByteBuffer
import java.nio.ByteOrder

/** Instanced circular dab renderer. All colour values are premultiplied linear output. */
class DabRenderer(private val maxDabs: Int) {
    private var program: GlProgram? = null
    private var vaoId = 0
    private var quadBufferId = 0
    private var instanceBufferId = 0
    private var uCanvasToClip = -1
    private var uColorLinear = -1

    fun create() {
        release()
        program = GlProgram(ShaderLib.dabVertex, ShaderLib.dabFragment)
        val currentProgram = checkNotNull(program)
        currentProgram.use()
        uCanvasToClip = GLES30.glGetUniformLocation(currentProgram.id, "uCanvasToClip")
        uColorLinear = GLES30.glGetUniformLocation(currentProgram.id, "uColorLinear")
        check(uCanvasToClip >= 0 && uColorLinear >= 0) { "Dab shader uniforms missing" }

        val quad = floatArrayOf(-1f, -1f, 1f, -1f, -1f, 1f, 1f, 1f)
        val quadData = ByteBuffer.allocateDirect(quad.size * Float.SIZE_BYTES)
            .order(ByteOrder.nativeOrder())
            .asFloatBuffer()
        quadData.put(quad).position(0)

        val ids = IntArray(1)
        GLES30.glGenVertexArrays(1, ids, 0)
        vaoId = ids[0]
        GLES30.glGenBuffers(1, ids, 0)
        quadBufferId = ids[0]
        GLES30.glGenBuffers(1, ids, 0)
        instanceBufferId = ids[0]

        GLES30.glBindVertexArray(vaoId)
        GLES30.glBindBuffer(GLES30.GL_ARRAY_BUFFER, quadBufferId)
        GLES30.glBufferData(
            GLES30.GL_ARRAY_BUFFER,
            quad.size * Float.SIZE_BYTES,
            quadData,
            GLES30.GL_STATIC_DRAW,
        )
        GLES30.glEnableVertexAttribArray(0)
        GLES30.glVertexAttribPointer(0, 2, GLES30.GL_FLOAT, false, 0, 0)

        GLES30.glBindBuffer(GLES30.GL_ARRAY_BUFFER, instanceBufferId)
        GLES30.glBufferData(
            GLES30.GL_ARRAY_BUFFER,
            maxDabs * DabBuffer.FLOATS_PER_DAB * Float.SIZE_BYTES,
            null,
            GLES30.GL_DYNAMIC_DRAW,
        )
        GLES30.glEnableVertexAttribArray(1)
        GLES30.glVertexAttribPointer(1, 4, GLES30.GL_FLOAT, false, 20, 0)
        GLES30.glVertexAttribDivisor(1, 1)
        GLES30.glEnableVertexAttribArray(2)
        GLES30.glVertexAttribPointer(2, 1, GLES30.GL_FLOAT, false, 20, 16)
        GLES30.glVertexAttribDivisor(2, 1)
        GLES30.glBindBuffer(GLES30.GL_ARRAY_BUFFER, 0)
        GLES30.glBindVertexArray(0)
    }

    /** Draws all buffered dabs into a document-space RenderTarget. */
    fun drawInto(
        target: RenderTarget,
        width: Int,
        height: Int,
        canvasToFbo: FloatArray,
        dabs: DabBuffer,
        colorLinear: FloatArray,
    ) {
        if (dabs.count == 0) return
        val currentProgram = program ?: return
        require(colorLinear.size >= 3)

        target.bind()
        GLES30.glViewport(0, 0, width, height)
        GLES30.glEnable(GLES30.GL_BLEND)
        GLES30.glBlendEquation(GLES30.GL_FUNC_ADD)
        GLES30.glBlendFunc(GLES30.GL_ONE, GLES30.GL_ONE_MINUS_SRC_ALPHA)
```

```
        currentProgram.use()
        GLES30.glUniformMatrix4fv(uCanvasToClip, 1, false, canvasToFbo, 0)
        GLES30.glUniform3f(uColorLinear, colorLinear[0], colorLinear[1], colorLinear[2])
        dabs.prepareForUpload()
        GLES30.glBindVertexArray(vaoId)
        GLES30.glBindBuffer(GLES30.GL_ARRAY_BUFFER, instanceBufferId)
        GLES30.glBufferSubData(
            GLES30.GL_ARRAY_BUFFER,
            0,
            dabs.count * DabBuffer.FLOATS_PER_DAB * Float.SIZE_BYTES,
            dabs.floats,
        )
        dabs.finishUpload()
        GLES30.glDrawArraysInstanced(GLES30.GL_TRIANGLE_STRIP, 0, 4, dabs.count)
        GLES30.glBindBuffer(GLES30.GL_ARRAY_BUFFER, 0)
        GLES30.glBindVertexArray(0)
        GLES30.glDisable(GLES30.GL_BLEND)
        GLES30.glBindFramebuffer(GLES30.GL_FRAMEBUFFER, 0)
    }

    fun release() {
        program?.release()
        program = null
        if (instanceBufferId != 0) GLES30.glDeleteBuffers(1, intArrayOf(instanceBufferId), 0)
        if (quadBufferId != 0) GLES30.glDeleteBuffers(1, intArrayOf(quadBufferId), 0)
        if (vaoId != 0) GLES30.glDeleteVertexArrays(1, intArrayOf(vaoId), 0)
        vaoId = 0
        quadBufferId = 0
        instanceBufferId = 0
        uCanvasToClip = -1
        uColorLinear = -1
    }
}
```

app/src/main/java/com/wetinknext/engine/brush/OneEuroFilter.kt

```
package com.wetinknext.engine.brush

import kotlin.math.PI
import kotlin.math.sqrt

class OneEuroFilter(private var minCutoff: Float = 1.6f, private var beta: Float = 0.25f, private var dCutoff: Float
= 1f) {
    private var initialized = false; private var lastTimestampNanos = 0L
    private var x = 0f; private var y = 0f; private var dx = 0f; private var dy = 0f
    fun reset() { initialized = false; lastTimestampNanos = 0L; x = 0f; y = 0f; dx = 0f; dy = 0f }
    fun filter(timestampNanos: Long, rawX: Float, rawY: Float, out: FloatArray) {
        require(out.size >= 2)
        if (!initialized) { initialized = true; lastTimestampNanos = timestampNanos; x = rawX; y = rawY; out[0] = x;
out[1] = y; return }
        var dt = (timestampNanos - lastTimestampNanos) / 1_000_000_000f
        if (dt < MIN_DT) dt = MIN_DT
        lastTimestampNanos = timestampNanos
        val rate = 1f / dt; val alphaD = alpha(rate, dCutoff)
        dx = alphaD * ((rawX - x) * rate) + (1f - alphaD) * dx; dy = alphaD * ((rawY - y) * rate) + (1f - alphaD) * dy
        val alpha = alpha(rate, (minCutoff + beta * sqrt(dx * dx + dy * dy)).coerceAtLeast(0.01f))
        x = alpha * rawX + (1f - alpha) * x; y = alpha * rawY + (1f - alpha) * y; out[0] = x; out[1] = y
    }
    private fun alpha(rate: Float, cutoff: Float): Float { val tau = 1f / (2f * PI.toFloat() * cutoff); return 1f / (
1f + tau * rate) }
    private companion object { const val MIN_DT = 1f / 1000f }
}
```

app/src/main/java/com/wetinknext/engine/brush/PressureFilter.kt

```
package com.wetinknext.engine.brush

import kotlin.math.exp

/**
 * Time-based stylus pressure smoothing.
 * A zero pressure sample while the pointer is still in contact is treated as a
 * device glitch; Android sends the real zero value on UP after the final point.
 */
class PressureFilter {
    private var initialized = false
    private var lastTimestampNanos = 0L
    private var value = 0f

    fun reset() { initialized = false; lastTimestampNanos = 0L; value = 0f }

    fun filter(timestampNanos: Long, rawPressure: Float): Float {
        val raw = rawPressure.coerceIn(0f, 1f)
        if (!initialized) { initialized = true; lastTimestampNanos = timestampNanos; value = raw; return value }
        val dt = ((timestampNanos - lastTimestampNanos) / NANOS_PER_SECOND).coerceIn(MIN_DT, MAX_DT)
        lastTimestampNanos = timestampNanos
        val stableRaw = if (raw <= ZERO_GLITCH_THRESHOLD && value > ZERO_GLITCH_THRESHOLD) value else raw
        val tau = if (stableRaw >= value) RISE_TAU_SECONDS else FALL_TAU_SECONDS
        val alpha = 1f - exp((-dt / tau).toDouble()).toFloat()
        value += (stableRaw - value) * alpha
        return value.coerceIn(0f, 1f)
    }

    private companion object {
        const val NANOS_PER_SECOND = 1_000_000_000f
        const val MIN_DT = 1f / 1000f
        const val MAX_DT = 1f / 20f
        const val RISE_TAU_SECONDS = .018f
        const val FALL_TAU_SECONDS = .055f
        const val ZERO_GLITCH_THRESHOLD = .01f
    }
}
```

app/src/main/java/com/wetinknext/engine/brush/RibbonEmitter.kt

```
package com.wetinknext.engine.brush

import com.wetinknext.engine.input.InputBatch
import kotlin.math.hypot
import kotlin.math.max
import kotlin.math.min
import kotlin.math.pow

class RibbonEmitter(private var settings: BrushSettings) {
    private var active = false; private var pointerId = -1; private var lastX = 0f; private var lastY = 0f; private
var length = 0f
    private val stabilizer = Stabilizer()
    private val pressureFilter = PressureFilter()
    private val samples = ArrayList<RibbonSample>(); private val arcs = ArrayList<Float>()
    val hasStroke get() = samples.isNotEmpty()
    var closedLoop = false
        private set
    var lastClosureDistance = -1f
        private set
    var lastClosureThreshold = -1f
        private set
    private var firstX = 0f; private var firstY = 0f; private var firstHalfWidth = 0f
    private var maxHalfWidth = 0f

    /** [redacted]: [redacted], [redacted] [redacted]. */
    val isDot: Boolean get() = samples.isNotEmpty() && length <= maxHalfWidth * DOT_SPAN_RATIO

    fun updateSettings(value: BrushSettings) { settings = value }
    fun reset() { active = false; pointerId = -1; lastX = 0f; lastY = 0f; length = 0f; closedLoop = false;
lastClosureDistance = -1f; lastClosureThreshold = -1f; firstX = 0f; firstY = 0f; firstHalfWidth = 0f; maxHalfWidth =
0f; stabilizer.reset(); pressureFilter.reset(); samples.clear(); arcs.clear() }
    fun begin(batch: InputBatch) {
        reset(); if (batch.isEmpty()) return
        stabilizer.strength = (settings.smoothing * .7f + settings.streamline * .3f).coerceIn(0f, 1f)
        val s = batch.samples[0]; active = true; pointerId = s.pointerId
        stabilizer.process(s.timestampNanos, s.canvasX, s.canvasY)
        lastX = stabilizer.x; lastY = stabilizer.y; firstHalfWidth = width(pressureFilter.filter(s.timestampNanos, s.
pressure)); maxHalfWidth = firstHalfWidth; firstX = lastX; firstY = lastY; samples += RibbonSample(lastX, lastY,
firstHalfWidth); arcs += 0f
    }
    fun append(batch: InputBatch) { if (!active) return; for (i in 0 until batch.sampleCount) { val s = batch.samples[
i]; if (s.pointerId == pointerId) { stabilizer.process(s.timestampNanos, s.canvasX, s.canvasY); add(stabilizer.x,
stabilizer.y, pressureFilter.filter(s.timestampNanos, s.pressure)) } } }
    fun finish(cancel: Boolean) {
        if (!active) return
        closedLoop = false
        lastClosureDistance = -1f
        lastClosureThreshold = -1f

        val longEnoughForLoop = length >= maxHalfWidth * MIN_LOOP_LENGTH_TO_WIDTH
        if (!cancel && samples.size >= MIN_LOOP_SAMPLES && longEnoughForLoop) {
            val last = samples.last()
            val distance = hypot(last.x - firstX, last.y - firstY)
            val threshold = max(2f * stepPx(), min(MAX_CLOSURE_PX, length * CLOSURE_LENGTH_RATIO))
            lastClosureDistance = distance
            lastClosureThreshold = threshold
            if (distance <= threshold) {
                if (distance <= stepPx() && samples.size > MIN_LOOP_SAMPLES) {
                    val dropped = samples.removeAt(samples.lastIndex)
                    arcs.removeAt(arcs.lastIndex)
                    val previous = samples.last()
                    length -= hypot(dropped.x - previous.x, dropped.y - previous.y)
                }
                closedLoop = samples.size >= 3
            }
            ClosureDebug.publish(distance, threshold, closedLoop, samples.size)
        }
        active = false
    }
    fun samples(): List<RibbonSample> = samples
    fun arcLengths(): FloatArray = arcs.toFloatArray()
    fun totalLength(): Float = length
    private fun add(x: Float, y: Float, pressure: Float) {
        val d = hypot(x - lastX, y - lastY)
        if (d < stepPx()) return
        val w = width(pressure)
        if (w > maxHalfWidth) maxHalfWidth = w
        length += d
        samples += RibbonSample(x, y, w)
        arcs += length
        lastX = x
        lastY = y
    }

    /** [redacted] 132 px [redacted] 1.25 px = [redacted] [redacted] [redacted] [redacted] [redacted] [redacted]. */
    private fun stepPx(): Float {
        val base = settings.ribbon.minPointDistancePx.coerceIn(.05f, 32f)
```

```
        return max(base, settings.baseRadiusPx * STEP_TO_RADIUS)
    }
    private fun width(pressure: Float): Float { val shaped = pressure.coerceIn(0f, 1f).pow(settings.pressureGamma.
coerceAtLeast(.01f)); val factor = if (settings.pressureToSize) settings.minSizeRatio + (1f - settings.minSizeRatio)
* shaped else 1f; return (settings.baseRadiusPx * factor).coerceAtLeast(.25f) }

    private companion object {
        const val STEP_TO_RADIUS = .12f
        const val DOT_SPAN_RATIO = .75f
        const val MIN_LOOP_SAMPLES = 10
        const val MIN_LOOP_LENGTH_TO_WIDTH = 6f
        const val MAX_CLOSURE_PX = 24f
        const val CLOSURE_LENGTH_RATIO = .15f
    }
}
```

app/src/main/java/com/wetinknext/engine/brush/RibbonGeometry.kt

```
package com.wetinknext.engine.brush

import kotlin.math.atan2
import kotlin.math.cos
import kotlin.math.hypot
import kotlin.math.sin

data class RVec2(val x: Float, val y: Float)
data class RibbonSample(val x: Float, val y: Float, val halfWidth: Float)

/** Axis and widths only. Segment sweep and joins are triangulated separately. */
data class RibbonOutline(
    val centers: List<RVec2>,
    val widths: FloatArray,
    val startCap: List<RVec2>,
    val endCap: List<RVec2>,
    val closed: Boolean = false,
) { val isEmpty: Boolean get() = centers.isEmpty() }

object RibbonGeometry {
    private const val CAP_SEGMENTS = 48
    private const val DOT_SPAN_RATIO = 0.75f

    fun build(samples: List<RibbonSample>, cap: RibbonCap, join: RibbonJoin, miterLimit: Float, aaWidthPx: Float = 0f,
 closed: Boolean = false): RibbonOutline {
        val clean = dedupe(samples)
        if (clean.isEmpty()) return RibbonOutline(emptyList(), FloatArray(0), emptyList(), emptyList())

        val isClosed = closed && clean.size >= 3
        if (clean.size == 1) {
            val point = clean.first(); val width = point.halfWidth.coerceAtLeast(0f)
            return RibbonOutline(listOf(RVec2(point.x, point.y)), floatArrayOf(width), fullCircle(point.x, point.y,
width), emptyList())
        }

        // 2-5 ■■■■■■■ ■■■■■■ ■■■■■■■■■■■■■ ■■■■■■■■: sweep ■■■■ ■■■■■■/■■■■■■■■■■■■. ■■■■■■■ ■■■■.
        var widest = 0f
        for (s in clean) if (s.halfWidth > widest) widest = s.halfWidth
        if (widest > 0f && pathLength(clean) <= widest * DOT_SPAN_RATIO) {
            val p = clean.last()
            return RibbonOutline(
                listOf(RVec2(p.x, p.y)),
                floatArrayOf(widest),
                fullCircle(p.x, p.y, widest),
                emptyList(),
            )
        }

        val centers = ArrayList<RVec2>(clean.size); val widths = FloatArray(clean.size)
        for (i in clean.indices) { centers += RVec2(clean[i].x, clean[i].y); widths[i] = clean[i].halfWidth.
coerceAtLeast(0f) }
        val start = if (!isClosed && cap == RibbonCap.ROUND) endArc(clean.first(), segmentDirection(clean, 0), widths.
first(), true) else emptyList()
        val end = if (!isClosed && cap == RibbonCap.ROUND) endArc(clean.last(), segmentDirection(clean, clean.
lastIndex - 1), widths.last(), false) else emptyList()
        return RibbonOutline(centers, widths, start, end, isClosed)
    }

    private fun dedupe(samples: List<RibbonSample>): List<RibbonSample> {
        if (samples.isEmpty()) return samples
        val result = ArrayList<RibbonSample>(samples.size); result += samples.first()
        for (i in 1 until samples.size) if (hypot(samples[i].x - result.last().x, samples[i].y - result.last().y) >=
1e-4f) result += samples[i]
        return result
    }
    private fun pathLength(samples: List<RibbonSample>): Float {
        var total = 0f
        for (i in 1 until samples.size) total += hypot(samples[i].x - samples[i - 1].x, samples[i].y - samples[i - 1].
y)
        return total
    }
    private fun segmentDirection(samples: List<RibbonSample>, index: Int): RVec2 { val dx = samples[index + 1].x -
samples[index].x; val dy = samples[index + 1].y - samples[index].y; val length = hypot(dx, dy).coerceAtLeast(1e-5f);
return RVec2(dx / length, dy / length) }
    private fun endArc(sample: RibbonSample, direction: RVec2, radius: Float, tail: Boolean): List<RVec2> {
        if (radius <= 0f) return emptyList()
        val base = if (tail) atan2(-direction.x, direction.y) else atan2(direction.x, -direction.y)
        val sweep = -kotlin.math.PI.toFloat()
        return List(CAP_SEGMENTS + 1) { i ->
            val angle = base + sweep * i / CAP_SEGMENTS
            RVec2(sample.x + radius * cos(angle), sample.y + radius * sin(angle))
        }
    }
    private fun fullCircle(x: Float, y: Float, radius: Float) = if (radius <= 0f) emptyList() else List(CAP_SEGMENTS
+ 1) { i -> val angle = 2f * kotlin.math.PI.toFloat() * i / CAP_SEGMENTS; RVec2(x + radius * cos(angle), y + radius *
sin(angle)) }
}
```

app/src/main/java/com/wetinknext/engine/brush/RibbonRenderer.kt

```
package com.wetinknext.engine.brush

import android.opengl.GLES30
import com.wetinknext.engine.gl.GlProgram
import com.wetinknext.engine.gl.RenderTarget
import com.wetinknext.engine.gl.ShaderLib
import java.nio.ByteBuffer
import java.nio.ByteOrder

class RibbonRenderer(private val maxVertices: Int = DEFAULT_MAX_VERTICES) {
    private var program: GlProgram? = null; private var vao = 0; private var vbo = 0; private var ebo = 0
    private var uMatrix = -1; private var uColor = -1; private var uFlow = -1; private var uAa = -1; private var
uNoAa = -1
    private val vertexData = ByteBuffer.allocateDirect(maxVertices * RibbonShader.VERTEX_STRIDE_BYTES).order(
ByteOrder.nativeOrder()).asFloatBuffer()
    private val indexData = ByteBuffer.allocateDirect(maxVertices * 3 * Int.SIZE_BYTES).order(ByteOrder.nativeOrder(
)).asIntBuffer()
    fun create() {
        release(); program = GlProgram(ShaderLib.ribbonVertex, ShaderLib.ribbonFragment); val p = checkNotNull(
program); p.use()
        uMatrix = GLES30.glGetUniformLocation(p.id, "uCanvasToClip"); uColor = GLES30.glGetUniformLocation(p.id,
"uColorLinear"); uFlow = GLES30.glGetUniformLocation(p.id, "uFlow"); uAa = GLES30.glGetUniformLocation(p.id,
"uAntiAliasLevel"); uNoAa = GLES30.glGetUniformLocation(p.id, "uNoAntialias")
        val ids = IntArray(1); GLES30.glGenVertexArrays(1, ids, 0); vao = ids[0]; GLES30.glGenBuffers(1, ids, 0); vbo
= ids[0]; GLES30.glGenBuffers(1, ids, 0); ebo = ids[0]
        GLES30.glBindVertexArray(vao); GLES30.glBindBuffer(GLES30.GL_ARRAY_BUFFER, vbo); GLES30.glBufferData(GLES30.
GL_ARRAY_BUFFER, maxVertices * RibbonShader.VERTEX_STRIDE_BYTES, null, GLES30.GL_DYNAMIC_DRAW)
        GLES30.glBindBuffer(GLES30.GL_ELEMENT_ARRAY_BUFFER, ebo); GLES30.glBufferData(GLES30.GL_ELEMENT_ARRAY_BUFFER,
maxVertices * 3 * Int.SIZE_BYTES, null, GLES30.GL_DYNAMIC_DRAW)
        GLES30.glEnableVertexAttribArray(0); GLES30.glVertexAttribPointer(0, 2, GLES30.GL_FLOAT, false, RibbonShader.
VERTEX_STRIDE_BYTES, 0)
        GLES30.glEnableVertexAttribArray(1); GLES30.glVertexAttribPointer(1, 1, GLES30.GL_FLOAT, false, RibbonShader.
VERTEX_STRIDE_BYTES, 8)
        GLES30.glEnableVertexAttribArray(2); GLES30.glVertexAttribPointer(2, 1, GLES30.GL_FLOAT, false, RibbonShader.
VERTEX_STRIDE_BYTES, 12); GLES30.glBindVertexArray(0)
    }
    var droppedMeshes = 0L
        private set

    fun draw(target: RenderTarget, width: Int, height: Int, canvasToFbo: FloatArray, mesh: RibbonMesh, color:
FloatArray, flow: Float, antiAliasLevel: Int, noAntialias: Boolean, nonBuildup: Boolean = false) {
        val p = program ?: return
        if (mesh.isEmpty) return
        if (mesh.vertexCount > maxVertices || mesh.indices.size > maxVertices * 3) { droppedMeshes++; return }
        vertexData.clear(); for (i in 0 until mesh.vertexCount) { vertexData.put(mesh.vertices[i*2]); vertexData.put(
mesh.vertices[i*2+1]); vertexData.put(mesh.coverage[i]); vertexData.put(mesh.alpha[i]) }; vertexData.flip()
        indexData.clear(); mesh.indices.forEach { indexData.put(it) }; indexData.flip()
        target.bind(); GLES30.glViewport(0, 0, width, height); GLES30.glEnable(GLES30.GL_BLEND)
        if (nonBuildup) {
            // ■■■■ ■■■■■■ ■■■■■■■■■■■, rgb = color*a => GL_MAX ■■ RGBA = ■■■■■■■■■■■■ ■■■■■■■■■.
            // ■■■■■■, ■■■■ ■■■■■■■■ ■ AA-■■■■ ■■■■■■■■■■ ■■■■■■■■■■■■■■■ ■■■■ ■■ ■■■■■■.
            GLES30.glBlendEquation(GLES30.GL_MAX); GLES30.glBlendFunc(GLES30.GL_ONE, GLES30.GL_ONE)
        } else {
            GLES30.glBlendEquation(GLES30.GL_FUNC_ADD); GLES30.glBlendFunc(GLES30.GL_ONE, GLES30.
GL_ONE_MINUS_SRC_ALPHA)
        }
        p.use()
        GLES30.glUniformMatrix4fv(uMatrix,1,false,canvasToFbo,0); GLES30.glUniform3f(uColor,color[0],color[1],color[
2]); GLES30.glUniform1f(uFlow,flow.coerceIn(0f,1f)); GLES30.glUniform1i(uAa,antiAliasLevel.coerceIn(0,3)); GLES30.
glUniform1i(uNoAa,if(noAntialias)1 else 0)
        GLES30.glBindVertexArray(vao); GLES30.glBindBuffer(GLES30.GL_ARRAY_BUFFER,vbo); GLES30.glBufferSubData(GLES30.
GL_ARRAY_BUFFER,0,mesh.vertexCount*RibbonShader.VERTEX_STRIDE_BYTES,vertexData); GLES30.glBindBuffer(GLES30.
GL_ELEMENT_ARRAY_BUFFER,ebo); GLES30.glBufferSubData(GLES30.GL_ELEMENT_ARRAY_BUFFER,0,mesh.indices.size*Int.
SIZE_BYTES,indexData); GLES30.glDrawElements(GLES30.GL_TRIANGLES,mesh.indices.size,GLES30.GL_UNSIGNED_INT,0); GLES30.
glBindVertexArray(0); GLES30.glBlendEquation(GLES30.GL_FUNC_ADD); GLES30.glDisable(GLES30.GL_BLEND); GLES30.
glBindFramebuffer(GLES30.GL_FRAMEBUFFER,0)
    }
    fun release(){ program?.release();program=null;if(vbo!=0)GLES30.glDeleteBuffers(1,intArrayOf(vbo),0);if(
ebo!=0)GLES30.glDeleteBuffers(1,intArrayOf(ebo),0);if(vao!=0)GLES30.glDeleteVertexArrays(1,intArrayOf(vao),0);vao=0;
vbo=0;ebo=0 }
    companion object { const val DEFAULT_MAX_VERTICES = 65_536 }
}
```

app/src/main/java/com/wetinknext/engine/brush/RibbonShader.kt

```
package com.wetinknext.engine.brush

/** Vertex format consumed by [RibbonRenderer]. Shader sources live in ShaderLib. */
object RibbonShader { const val FLOATS_PER_VERTEX = 4; const val VERTEX_STRIDE_BYTES = FLOATS_PER_VERTEX * Float.
SIZE_BYTES }
```

app/src/main/java/com/wetinknext/engine/brush/RibbonTriangulation.kt

```
package com.wetinknext.engine.brush

import kotlin.math.abs
import kotlin.math.atan2
import kotlin.math.ceil
import kotlin.math.cos
import kotlin.math.hypot
import kotlin.math.sin

data class RibbonMesh(val vertices: FloatArray, val indices: IntArray, val coverage: FloatArray = FloatArray(vertices.
size / 2) { 1f }, val alpha: FloatArray = FloatArray(vertices.size / 2) { 1f }) { val vertexCount get() = vertices.
size / 2; val triangleCount get() = indices.size / 3; val isEmpty get() = indices.isEmpty() }

/** Disk sweep: quads follow segment normals, outer corners use a continuous fan. */
object RibbonTriangulation {
    fun build(outline: RibbonOutline, aaWidthPx: Float = 0f, arcLengths: FloatArray = FloatArray(0), totalLen: Float
= 0f, taperStartPx: Float = 0f, taperEndPx: Float = 0f, taperOpacity: Boolean = false, join: RibbonJoin = RibbonJoin.
ROUND, miterLimit: Float = 3f): RibbonMesh {
        if (outline.isEmpty) return RibbonMesh(FloatArray(0), IntArray(0))
        val centers = outline.centers; val widths = outline.widths; val n = centers.size
        val vertices = ArrayList<Float>(); val indices = ArrayList<Int>(); val coverage = ArrayList<Float>(); val
alpha = ArrayList<Float>()
        fun add(x: Float, y: Float, cov: Float = 1f, arc: Float = -1f): Int { val id = vertices.size / 2; vertices +=
x; vertices += y; coverage += cov; alpha += if (taperOpacity && arc >= 0f && totalLen > 0f) taper(arc, totalLen,
taperStartPx, taperEndPx) else 1f; return id }
        fun triangle(a: Int, b: Int, c: Int) { indices += a; indices += b; indices += c }
        fun arcAt(index: Int) = arcLengths.getOrElse(index) { -1f }
        if (n == 1) { fan(centers[0], outline.startCap, aaWidthPx, widths[0], arcAt(0), ::add, ::triangle); return
RibbonMesh(vertices.toFloatArray(), indices.toIntArray(), coverage.toFloatArray(), alpha.toFloatArray()) }
        val segmentCount = if (outline.closed) n else n - 1
        val directions = Array(segmentCount) { i -> val j = (i + 1) % n; val dx = centers[j].x - centers[i].x; val dy
= centers[j].y - centers[i].y; val length = hypot(dx, dy).coerceAtLeast(1e-5f); RVec2(dx / length, dy / length) }
        for (i in 0 until segmentCount) {
            val j = (i + 1) % n
            val normal = RVec2(-directions[i].y, directions[i].x); val a = add(centers[i].x + normal.x * widths[i],
centers[i].y + normal.y * widths[i], 1f, arcAt(i)); val b = add(centers[i].x - normal.x * widths[i], centers[i].y -
normal.y * widths[i], 1f, arcAt(i)); val d = add(centers[j].x + normal.x * widths[j], centers[j].y + normal.y *
widths[j], 1f, arcAt(j)); val e = add(centers[j].x - normal.x * widths[j], centers[j].y - normal.y * widths[j], 1f,
arcAt(j)); triangle(a,b,d); triangle(b,e,d)
            if (aaWidthPx > 0f) { val a2 = add(centers[i].x + normal.x * (widths[i]+aaWidthPx), centers[i].y + normal.
y * (widths[i]+aaWidthPx), 0f, arcAt(i)); val d2 = add(centers[j].x + normal.x * (widths[j]+aaWidthPx), centers[j].y
+ normal.y * (widths[j]+aaWidthPx), 0f, arcAt(j)); val b2 = add(centers[i].x - normal.x * (widths[i]+aaWidthPx),
centers[i].y - normal.y * (widths[i]+aaWidthPx), 0f, arcAt(i)); val e2 = add(centers[j].x - normal.x * (widths[
j]+aaWidthPx), centers[j].y - normal.y * (widths[j]+aaWidthPx), 0f, arcAt(j)); triangle(a,a2,d2); triangle(a,d2,d);
triangle(b,e2,b2); triangle(b,e,e2) }
        }
        if (join != RibbonJoin.MITER) for (i in 0 until n) {
            if (!outline.closed && (i == 0 || i == n - 1)) continue
            val previous = directions[(i - 1 + segmentCount) % segmentCount]; val next = directions[i % segmentCount];
 val cross = previous.x * next.y - previous.y * next.x; if (abs(cross) < 1e-5f) continue
            val pn = RVec2(-previous.y, previous.x); val nn = RVec2(-next.y, next.x); var mx = pn.x + nn.x; var my =
pn.y + nn.y; val ml = hypot(mx,my); if (ml < 1e-5f) continue; mx /= ml; my /= ml
            val width = widths[i]; val outerSign = if (cross < 0f) 1f else -1f; val outerA = RVec2(centers[i].x +
outerSign * pn.x * width, centers[i].y + outerSign * pn.y * width); val outerB = RVec2(centers[i].x + outerSign * nn.
x * width, centers[i].y + outerSign * nn.y * width); fan(centers[i], arcPoints(centers[i], outerA, outerB, width),
aaWidthPx, width, arcAt(i), ::add, ::triangle)
            val limit = miterLimit.coerceAtLeast(1f)
            val cosHalf = (mx * pn.x + my * pn.y).coerceAtLeast(1f / limit)
            val scale = (1f / cosHalf).coerceAtMost(limit)
            // ■■■■■■■■■■■ ■■■■■ ■■■■■■■■ ■■■■■■■■■■ ■■■■■■■■■ = self-intersection ■ ■■■■■■■■■■ ■■ ■■■■■■■■ ■■■■■■.
            val pi2 = (i - 1 + n) % n
            val ni2 = (i + 1) % n
            val prevLen = hypot(centers[i].x - centers[pi2].x, centers[i].y - centers[pi2].y)
            val nextLen = hypot(centers[ni2].x - centers[i].x, centers[ni2].y - centers[i].y)
            if (width * scale <= kotlin.math.min(prevLen, nextLen) * .9f) {
                val innerA = RVec2(centers[i].x - outerSign * pn.x * width, centers[i].y - outerSign * pn.y * width)
                val innerB = RVec2(centers[i].x - outerSign * nn.x * width, centers[i].y - outerSign * nn.y * width)
                val innerMiter = RVec2(centers[i].x - outerSign * mx * width * scale, centers[i].y - outerSign * my *
width * scale)
                triangle(add(innerA.x, innerA.y, 1f, arcAt(i)), add(innerMiter.x, innerMiter.y, 1f, arcAt(i)), add(
innerB.x, innerB.y, 1f, arcAt(i)))
            }
        }
        if (!outline.closed && outline.startCap.isNotEmpty()) fan(centers[0], outline.startCap, aaWidthPx, widths[0],
arcAt(0), ::add, ::triangle)
        if (!outline.closed && outline.endCap.isNotEmpty()) fan(centers.last(), outline.endCap, aaWidthPx, widths.
last(), arcAt(n-1), ::add, ::triangle)
        return RibbonMesh(vertices.toFloatArray(), indices.toIntArray(), coverage.toFloatArray(), alpha.toFloatArray(
))
    }
    private fun fan(center: RVec2, points: List<RVec2>, aa: Float, width: Float, arc: Float, add: (Float,Float,Float,
Float)->Int, triangle: (Int,Int,Int)->Unit) { if(points.size<2)return; val c=add(center.x,center.y,1f,arc); val
inner=IntArray(points.size){add(points[it].x,points[it].y,1f,arc)}; val outer=if(aa>0f) IntArray(points.size){i->val
dx=points[i].x-center.x;val dy=points[i].y-center.y;val l=hypot(dx,dy).coerceAtLeast(1e-5f);add(center.x+dx/l*(
width+aa),center.y+dy/l*(width+aa),0f,arc)} else null; for(i in 0 until points.lastIndex){triangle(c,inner[i],inner[
```

```
i+1]);if(outer!=null){triangle(inner[i],outer[i],outer[i+1]);triangle(inner[i],outer[i+1],inner[i+1])}} }
    private fun arcPoints(center:RVec2,a:RVec2,b:RVec2,width:Float):List<RVec2>{val start=atan2(a.y-center.y,a.x-
center.x);var delta=atan2(b.y-center.y,b.x-center.x)-start;while(delta>kotlin.math.PI.toFloat())delta-=2f*kotlin.math.
PI.toFloat();while(delta< -kotlin.math.PI.toFloat())delta+=2f*kotlin.math.PI.toFloat();val steps=ceil(abs(delta)/.
35f).toInt().coerceIn(1,12);return List(steps+1){i->val angle=start+delta*i/steps;RVec2(center.x+cos(angle)*width,
center.y+sin(angle)*width)}}
    private fun taper(arc:Float,total:Float,start:Float,end:Float):Float{val a=if(start<=0f)1f else smooth(arc/start);
val b=if(end<=0f)1f else smooth((total-arc)/end);return(a*b).coerceAtLeast(.01f)}
    private fun smooth(value:Float):Float{val x=value.coerceIn(0f,1f);return x*x*(3f-2f*x)}
}
```

app/src/main/java/com/wetinknext/engine/brush/Stabilizer.kt

```
package com.wetinknext.engine.brush

import kotlin.math.sqrt

class Stabilizer(private val filter: OneEuroFilter = OneEuroFilter()) {
    var strength = 0f; var x = 0f; private set; var y = 0f; private set; var velocity = 0f; private set
    private var hasLast = false; private var lastX = 0f; private var lastY = 0f; private var lastT = 0L; private val
filtered = FloatArray(2)
    fun reset() { filter.reset(); x = 0f; y = 0f; velocity = 0f; hasLast = false; lastX = 0f; lastY = 0f; lastT = 0L }
    fun process(timestampNanos: Long, rawX: Float, rawY: Float) {
        if (strength <= 0f) { x = rawX; y = rawY } else { filter.filter(timestampNanos, rawX, rawY, filtered); x =
filtered[0]; y = filtered[1] }
        if (hasLast) { var dt = (timestampNanos - lastT) / 1_000_000_000f; if (dt < 1f / 1000f) dt = 1f / 1000f; val
d = sqrt((x-lastX)*(x-lastX)+(y-lastY)*(y-lastY)); velocity += 0.2f * (d / dt - velocity) }
        hasLast = true; lastX = x; lastY = y; lastT = timestampNanos
    }
}
```

app/src/main/java/com/wetinknext/engine/brush/StampEmitter.kt

```
package com.wetinknext.engine.brush

import com.wetinknext.engine.input.InputBatch
import kotlin.math.pow
import kotlin.math.sqrt

class StampEmitter(initialSettings: BrushSettings) {
    var settings: BrushSettings = initialSettings
        private set

    fun updateSettings(newSettings: BrushSettings) {
        settings = newSettings
    }

    private val stabilizer = Stabilizer(); private var active = false; private var pointerId = -1; private var
hasLast = false
    private var lastX = 0f; private var lastY = 0f; private var lastPressure = 0f; private var carried = 0f; private
var spacingPx = 1f
    fun reset() { active = false; pointerId = -1; hasLast = false; carried = 0f; stabilizer.reset() }
    fun begin(batch: InputBatch, out: DabBuffer) { reset(); if (batch.isEmpty()) return; stabilizer.strength = (
settings.smoothing*.7f + settings.streamline*.3f).coerceIn(0f,1f); spacingPx = (if(settings.spacingUsesDiameter)
settings.baseRadiusPx*2f*settings.spacing else settings.spacing).coerceIn(.75f,256f); val s=batch.samples[0];
active=true; pointerId=s.pointerId; stabilizer.process(s.timestampNanos,s.canvasX,s.canvasY); lastX=stabilizer.x;
lastY=stabilizer.y;lastPressure=s.pressure;hasLast=true; addDab(lastX,lastY,lastPressure,out) }
    fun append(batch: InputBatch, out: DabBuffer) { if(!active)return; for(i in 0 until batch.sampleCount){val
s=batch.samples[i];if(s.pointerId==pointerId){stabilizer.process(s.timestampNanos,s.canvasX,s.canvasY);addPoint(
stabilizer.x,stabilizer.y,s.pressure,out)}} }
    fun finish(out: DabBuffer, cancel: Boolean) { if(active && !cancel && carried > .001f) addDab(lastX,lastY,
lastPressure,out); reset() }
    private fun addPoint(x:Float,y:Float,p:Float,out:DabBuffer){ val dx=x-lastX;val dy=y-lastY;val dist=sqrt(
dx*dx+dy*dy);if(!hasLast||dist<1e-5f){lastX=x;lastY=y;lastPressure=p;hasLast=true;return};var travelled=0f;var
remaining=dist;while(carried+remaining>=spacingPx){val step=spacingPx-carried;travelled+=step;remaining-=step;val
t=travelled/dist;addDab(lastX+dx*t,lastY+dy*t,lastPressure+(p-lastPressure)*t,out);carried=0f};carried+=remaining;
lastX=x;lastY=y;lastPressure=p }
    private fun addDab(x:Float,y:Float,p:Float,out:DabBuffer){val pressure=p.coerceIn(0f,1f).let{if(settings.
pressureGamma>0f)it.pow(settings.pressureGamma)else it};val size=if(settings.pressureToSize)settings.minSizeRatio+(1f-
settings.minSizeRatio)*pressure else 1f;val alpha=if(settings.pressureToOpacity)pressure else 1f;out.add(x,y,(
settings.baseRadiusPx*size).coerceAtLeast(.25f),0f,(settings.opacity*alpha).coerceIn(0f,1f))}
}
```

app/src/main/java/com/wetinknext/engine/canvas/BlendMode.kt

```
package com.wetinknext.engine.canvas

/** Stored per layer now; shader implementations beyond NORMAL are scheduled after P6. */
enum class BlendMode(val id: Int) {
    NORMAL(0),
    MULTIPLY(1),
    SCREEN(2),
    OVERLAY(3),
    DARKEN(4),
    LIGHTEN(5),
    ADD(6),
}
```

app/src/main/java/com/wetinknext/engine/canvas/Compositor.kt

```
package com.wetinknext.engine.canvas

import android.opengl.GLES30
import com.wetinknext.engine.gl.CanvasGeometry
import com.wetinknext.engine.gl.GlCheck
import com.wetinknext.engine.gl.GlProgram
import com.wetinknext.engine.gl.ShaderLib

/** Presents visible layers bottom-to-top, inserting the active preview at its layer position. */
class Compositor {
    private var program: GlProgram? = null
    private var uCanvasToClip = -1
    private var uCanvasSize = -1
    private var uLayerTex = -1
    private var uStrokeTex = -1
    private var uStrokeActive = -1
    private var uOpacity = -1
    private var uStrokeOpacity = -1

    fun create() {
        release()
        program = GlProgram(ShaderLib.compositorVertex, ShaderLib.compositorFragment)
        val currentProgram = checkNotNull(program)
        currentProgram.use()
        uCanvasToClip = GLES30.glGetUniformLocation(currentProgram.id, "uCanvasToClip")
        uCanvasSize = GLES30.glGetUniformLocation(currentProgram.id, "uCanvasSize")
        uLayerTex = GLES30.glGetUniformLocation(currentProgram.id, "uLayerTex")
        uStrokeTex = GLES30.glGetUniformLocation(currentProgram.id, "uStrokeTex")
        uStrokeActive = GLES30.glGetUniformLocation(currentProgram.id, "uStrokeActive")
        uOpacity = GLES30.glGetUniformLocation(currentProgram.id, "uOpacity")
        uStrokeOpacity = GLES30.glGetUniformLocation(currentProgram.id, "uStrokeOpacity")
        check(uCanvasToClip >= 0 && uCanvasSize >= 0 && uLayerTex >= 0 &&
            uStrokeTex >= 0 && uStrokeActive >= 0 && uOpacity >= 0 && uStrokeOpacity >= 0) {
            "Compositor uniforms missing: canvasToClip=$uCanvasToClip, " +
                "canvasSize=$uCanvasSize, layerTex=$uLayerTex, strokeTex=$uStrokeTex, " +
                "strokeActive=$uStrokeActive, opacity=$uOpacity, strokeOpacity=$uStrokeOpacity"
        }
        GlCheck.noError("Compositor create")
    }

    fun render(
        geometry: CanvasGeometry,
        layers: LayerStack,
        activeLayerId: Long,
        strokeTextureId: Int,
        canvasToClip: FloatArray,
        strokeOpacity: Float,
    ) {
        val currentProgram = program ?: return
        currentProgram.use()
        GLES30.glUniformMatrix4fv(uCanvasToClip, 1, false, canvasToClip, 0)
        GLES30.glUniform2f(uCanvasSize, layers.canvasWidth.toFloat(), layers.canvasHeight.toFloat())
        GLES30.glEnable(GLES30.GL_BLEND)
        GLES30.glBlendFunc(GLES30.GL_ONE, GLES30.GL_ONE_MINUS_SRC_ALPHA)

        for (layer in layers.allLayers()) {
            if (!layer.created || !layer.isVisible) continue
            val hasStroke = layer.id == activeLayerId && strokeTextureId != 0

            GLES30.glActiveTexture(GLES30.GL_TEXTURE0)
            GLES30.glBindTexture(GLES30.GL_TEXTURE_2D, layer.target.textureId)
            GLES30.glUniform1i(uLayerTex, 0)
            if (hasStroke) {
                GLES30.glActiveTexture(GLES30.GL_TEXTURE1)
                GLES30.glBindTexture(GLES30.GL_TEXTURE_2D, strokeTextureId)
                GLES30.glUniform1i(uStrokeTex, 1)
            }
            GLES30.glUniform1i(uStrokeActive, if (hasStroke) 1 else 0)
            GLES30.glUniform1f(uOpacity, layer.opacity.coerceIn(0f, 1f))
            GLES30.glUniform1f(uStrokeOpacity, strokeOpacity.coerceIn(0f, 1f))
            geometry.draw()
        }

        GLES30.glDisable(GLES30.GL_BLEND)
        GLES30.glActiveTexture(GLES30.GL_TEXTURE1)
        GLES30.glBindTexture(GLES30.GL_TEXTURE_2D, 0)
        GLES30.glActiveTexture(GLES30.GL_TEXTURE0)
        GLES30.glBindTexture(GLES30.GL_TEXTURE_2D, 0)
    }

    fun release() {
        program?.release()
        program = null
        uCanvasToClip = -1
        uCanvasSize = -1
        uLayerTex = -1
        uStrokeTex = -1
```

```
        uStrokeActive = -1
        uOpacity = -1
        uStrokeOpacity = -1
    }
}
```

app/src/main/java/com/wetinknext/engine/canvas/LayerStack.kt

```
package com.wetinknext.engine.canvas

import com.wetinknext.engine.core.Camera
import com.wetinknext.engine.gl.GlCaps

/**
 * Render-thread-owned ordered collection of document layers.
 * Index 0 is the bottom-most layer and the final item is the top-most layer.
 */
class LayerStack {
    val camera = Camera()

    private val layers = mutableListOf<PaintLayer>()
    private var nextId = 1L

    var activeLayerId: Long = NO_LAYER
        private set
    var canvasWidth: Int = 0
        private set
    var canvasHeight: Int = 0
        private set

    val count: Int get() = layers.size

    fun allLayers(): List<PaintLayer> = layers

    fun activeLayer(): PaintLayer? = findLayerById(activeLayerId)

    fun findLayerById(id: Long): PaintLayer? = layers.firstOrNull { it.id == id }

    /** Creates the locked opaque white background and one transparent drawing layer. */
    fun create(caps: GlCaps, width: Int, height: Int) {
        require(width > 0 && height > 0)
        release()
        canvasWidth = width
        canvasHeight = height

        val background = newLayer("[redacted]", caps.supportsHalfFloatColorBuffer).also {
            it.isLocked = true
            it.target.clear(1f, 1f, 1f, 1f)
        }
        layers += background

        val drawing = newLayer("[redacted] 1", caps.supportsHalfFloatColorBuffer)
        layers += drawing
        activeLayerId = drawing.id
    }

    fun addLayer(name: String, insertAt: Int = layers.size): PaintLayer {
        check(canvasWidth > 0 && canvasHeight > 0) { "LayerStack has not been created" }
        val layer = newLayer(name, activeLayer()?.target?.usesHalfFloat == true)
        layers.add(insertAt.coerceIn(0, layers.size), layer)
        activeLayerId = layer.id
        return layer
    }

    /** A document must retain at least one layer; locked layers cannot be removed. */
    fun removeLayer(id: Long): PaintLayer? {
        val index = layers.indexOfFirst { it.id == id }
        if (index < 0 || layers.size <= 1) return null
        val layer = layers[index]
        if (layer.isLocked) return null

        layers.removeAt(index)
        layer.release()
        if (activeLayerId == id) {
            activeLayerId = layers[index.coerceAtMost(layers.lastIndex)].id
        }
        return layer
    }

    fun setActive(id: Long): Boolean {
        if (findLayerById(id) == null) return false
        activeLayerId = id
        return true
    }

    fun moveLayer(id: Long, delta: Int): Boolean {
        val index = layers.indexOfFirst { it.id == id }
        if (index < 0) return false
        val newIndex = (index + delta).coerceIn(0, layers.lastIndex)
        if (newIndex == index) return false
        val layer = layers.removeAt(index)
        layers.add(newIndex, layer)
        return true
    }
    fun release() {
```

```
        layers.forEach(PaintLayer::release)
        layers.clear()
        activeLayerId = NO_LAYER
        canvasWidth = 0
        canvasHeight = 0
    }

    private fun newLayer(name: String, useHalfFloat: Boolean): PaintLayer =
        PaintLayer(nextId++, name).also { it.create(canvasWidth, canvasHeight, useHalfFloat) }

    private companion object {
        const val NO_LAYER = -1L
    }
}
```

app/src/main/java/com/wetinknext/engine/canvas/PaintLayer.kt

```
package com.wetinknext.engine.canvas

import com.wetinknext.engine.gl.RenderTarget

/** One paintable document layer and its private GPU target. */
class PaintLayer(
    val id: Long,
    var name: String,
) {
    val target = RenderTarget()

    var isVisible = true
    var isLocked = false
    var opacity = 1f
    var blendMode = BlendMode.NORMAL
    var version = 0L

    var created = false
        private set

    fun create(width: Int, height: Int, preferHalfFloat: Boolean) {
        if (created && target.width == width && target.height == height) return
        target.create(width, height, preferHalfFloat)
        target.clear(0f, 0f, 0f, 0f)
        created = true
    }

    fun clear() {
        if (created) target.clear(0f, 0f, 0f, 0f)
    }

    fun release() {
        target.release()
        created = false
    }
}
```

app/src/main/java/com/wetinknext/engine/canvas/StrokeBlitter.kt

```
package com.wetinknext.engine.canvas

import android.opengl.GLES30
import com.wetinknext.engine.gl.CanvasGeometry
import com.wetinknext.engine.gl.GlProgram
import com.wetinknext.engine.gl.RenderTarget
import com.wetinknext.engine.gl.ShaderLib

/** ■■■■■■■■■■ ■■■■■■■■ ■■■■■■ ■■ strokeTarget ■ ■■■■: premultiplied source-over * opacity. */
class StrokeBlitter {
    private var program: GlProgram? = null
    private var uCanvasToClip = -1
    private var uCanvasSize = -1
    private var uStrokeTex = -1
    private var uOpacity = -1

    fun create() {
        release()
        program = GlProgram(ShaderLib.compositorVertex, ShaderLib.strokeBlitFragment)
        val p = checkNotNull(program)
        p.use()
        uCanvasToClip = GLES30.glGetUniformLocation(p.id, "uCanvasToClip")
        uCanvasSize = GLES30.glGetUniformLocation(p.id, "uCanvasSize")
        uStrokeTex = GLES30.glGetUniformLocation(p.id, "uStrokeTex")
        uOpacity = GLES30.glGetUniformLocation(p.id, "uOpacity")
        check(uCanvasToClip >= 0 && uCanvasSize >= 0 && uStrokeTex >= 0 && uOpacity >= 0) {
            "StrokeBlitter uniforms missing"
        }
    }

    fun blit(
        layer: RenderTarget,
        geometry: CanvasGeometry,
        strokeTextureId: Int,
        canvasToFbo: FloatArray,
        width: Int,
        height: Int,
        opacity: Float,
    ) {
        val p = program ?: return
        if (strokeTextureId == 0) return
        layer.bind()
        GLES30.glViewport(0, 0, width, height)
        GLES30.glDisable(GLES30.GL_SCISSOR_TEST)
        GLES30.glEnable(GLES30.GL_BLEND)
        GLES30.glBlendEquation(GLES30.GL_FUNC_ADD)
        GLES30.glBlendFunc(GLES30.GL_ONE, GLES30.GL_ONE_MINUS_SRC_ALPHA)
        p.use()
        GLES30.glUniformMatrix4fv(uCanvasToClip, 1, false, canvasToFbo, 0)
        GLES30.glUniform2f(uCanvasSize, width.toFloat(), height.toFloat())
        GLES30.glActiveTexture(GLES30.GL_TEXTURE0)
        GLES30.glBindTexture(GLES30.GL_TEXTURE_2D, strokeTextureId)
        GLES30.glUniform1i(uStrokeTex, 0)
        GLES30.glUniform1f(uOpacity, opacity.coerceIn(0f, 1f))
        geometry.draw()
        GLES30.glDisable(GLES30.GL_BLEND)
        GLES30.glBindTexture(GLES30.GL_TEXTURE_2D, 0)
        GLES30.glBindFramebuffer(GLES30.GL_FRAMEBUFFER, 0)
    }

    fun release() {
        program?.release()
        program = null
        uCanvasToClip = -1; uCanvasSize = -1; uStrokeTex = -1; uOpacity = -1
    }
}
```

app/src/main/java/com/wetinknext/engine/core/Camera.kt

```
package com.wetinknext.engine.core

import java.util.concurrent.atomic.AtomicReference
import kotlin.math.min

/** Thread-safe camera state shared by the UI and GL threads. */
class Camera(initial: ViewTransform = ViewTransform()) {
    private val state = AtomicReference(initial)

    fun snapshot(): ViewTransform = state.get()

    fun set(transform: ViewTransform) {
        state.set(transform)
    }

    fun update(transform: (ViewTransform) -> ViewTransform) {
        while (true) {
            val current = state.get()
            if (state.compareAndSet(current, transform(current))) return
        }
    }

    fun reset() {
        state.set(ViewTransform())
    }

    fun fitCanvas(
        canvasWidth: Int,
        canvasHeight: Int,
        viewWidth: Int,
        viewHeight: Int,
        marginRatio: Float = 0.04f,
    ) {
        if (canvasWidth <= 0 || canvasHeight <= 0 || viewWidth <= 0 || viewHeight <= 0) return
        val usableWidth = viewWidth * (1f - marginRatio * 2f)
        val usableHeight = viewHeight * (1f - marginRatio * 2f)
        val scale = min(usableWidth / canvasWidth, usableHeight / canvasHeight)
            .coerceIn(ViewTransform.MIN_SCALE, ViewTransform.MAX_SCALE)
        state.set(
            ViewTransform(
                scale = scale,
                translateX = (viewWidth - canvasWidth * scale) * 0.5f,
                translateY = (viewHeight - canvasHeight * scale) * 0.5f,
            ),
        )
    }
}
```

app/src/main/java/com/wetinknext/engine/core/DirtyRect.kt

```
package com.wetinknext.engine.core

import kotlin.math.ceil
import kotlin.math.floor
import kotlin.math.max
import kotlin.math.min

/** Reusable mutable dirty rectangle in canvas coordinates. */
class DirtyRect {
    var left = 0f; private set
    var top = 0f; private set
    var right = 0f; private set
    var bottom = 0f; private set
    var isEmpty = true; private set

    fun clear() {
        left = 0f; top = 0f; right = 0f; bottom = 0f; isEmpty = true
    }

    fun include(x: Float, y: Float, radius: Float) = include(x - radius, y - radius, x + radius, y + radius)

    fun include(left: Float, top: Float, right: Float, bottom: Float) {
        if (right < left || bottom < top) return
        if (isEmpty) {
            this.left = left; this.top = top; this.right = right; this.bottom = bottom; isEmpty = false
            return
        }
        this.left = min(this.left, left); this.top = min(this.top, top)
        this.right = max(this.right, right); this.bottom = max(this.bottom, bottom)
    }

    fun include(other: DirtyRect) {
        if (!other.isEmpty) include(other.left, other.top, other.right, other.bottom)
    }

    fun expand(pixels: Float) {
        if (!isEmpty && pixels > 0f) {
            left -= pixels; top -= pixels; right += pixels; bottom += pixels
        }
    }

    fun clamp(canvasWidth: Float, canvasHeight: Float) {
        if (isEmpty) return
        left = left.coerceIn(0f, canvasWidth); top = top.coerceIn(0f, canvasHeight)
        right = right.coerceIn(0f, canvasWidth); bottom = bottom.coerceIn(0f, canvasHeight)
        if (right <= left || bottom <= top) clear()
    }

    fun copyTo(out: DirtyRect) {
        out.left = left; out.top = top; out.right = right; out.bottom = bottom; out.isEmpty = isEmpty
    }

    fun toPixelBounds(out: IntArray) {
        require(out.size >= 4)
        if (isEmpty) {
            out[0] = 0; out[1] = 0; out[2] = 0; out[3] = 0
            return
        }
        out[0] = floor(left).toInt(); out[1] = floor(top).toInt()
        out[2] = ceil(right).toInt(); out[3] = ceil(bottom).toInt()
    }
}
```

app/src/main/java/com/wetinknext/engine/core/EditorState.kt

```
package com.wetinknext.engine.core

/** Immutable state published by the GL thread for Compose to render. */
data class LayerUiModel(
    val id: Long,
    val name: String,
    val isVisible: Boolean,
    val isLocked: Boolean,
    val opacity: Float,
    val active: Boolean,
    val canDelete: Boolean,
)

data class EditorUiState(
    val layers: List<LayerUiModel>,
    val canUndo: Boolean,
    val canRedo: Boolean,
    val brushSizePx: Float,
    val brushOpacity: Float,
    val activeLayerId: Long,
    val ready: Boolean,
) {
    companion object {
        val empty = EditorUiState(
            layers = emptyList(),
            canUndo = false,
            canRedo = false,
            brushSizePx = 16f,
            brushOpacity = 1f,
            activeLayerId = -1L,
            ready = false,
        )
    }
}
```

app/src/main/java/com/wetinknext/engine/core/EngineRenderer.kt

```
package com.wetinknext.engine.core

import android.opengl.GLES30
import android.opengl.GLSurfaceView
import android.view.MotionEvent
import com.wetinknext.engine.brush.BrushSettings
import com.wetinknext.engine.brush.BrushRenderMode
import com.wetinknext.engine.brush.CapsuleEmitter
import com.wetinknext.engine.brush.CapsuleStrokeRenderer
import com.wetinknext.engine.brush.ColorSpaces
import com.wetinknext.engine.brush.DabBuffer
import com.wetinknext.engine.brush.DabRenderer
import com.wetinknext.engine.brush.StampEmitter
import com.wetinknext.engine.canvas.Compositor
import com.wetinknext.engine.canvas.LayerStack
import com.wetinknext.engine.canvas.StrokeBlitter
import com.wetinknext.engine.gl.CanvasGeometry
import com.wetinknext.engine.gl.GlCaps
import com.wetinknext.engine.gl.GlCheck
import com.wetinknext.engine.gl.RenderTarget
import com.wetinknext.engine.input.InputAction
import com.wetinknext.engine.input.InputBatch
import com.wetinknext.engine.input.InputBatchPool
import com.wetinknext.engine.input.StrokeInputCapturer
import com.wetinknext.engine.undo.TileSnapshotCapture
import com.wetinknext.engine.undo.TileSnapshotRestore
import com.wetinknext.engine.undo.UndoEntry
import com.wetinknext.engine.undo.UndoManager
import java.util.concurrent.ArrayBlockingQueue
import java.util.concurrent.atomic.AtomicBoolean
import javax.microedition.khronos.egl.EGLConfig
import javax.microedition.khronos.opengles.GL10

/**
 * The P6 render-thread owner. Active dabs are accumulated only in strokeTarget;
 * UP snapshots and merges them into the selected PaintLayer atomically.
 */
class EngineRenderer(
    private val documentWidth: Int = DEFAULT_CANVAS_WIDTH,
    private val documentHeight: Int = DEFAULT_CANVAS_HEIGHT,
) : GLSurfaceView.Renderer {
    private var caps: GlCaps? = null
    private val layerStack = LayerStack()
    private val undoManager = UndoManager()

    private var geometry: CanvasGeometry? = null
    private var compositor: Compositor? = null
    private var dabRenderer: DabRenderer? = null
    private var capsuleRenderer: CapsuleStrokeRenderer? = null
    private val strokeTarget = RenderTarget()
    private val canvasToFboMatrix = FloatArray(16)
    private val canvasToClipMatrix = FloatArray(16)

    private var screenWidth = 1
    private var screenHeight = 1

    private val inputPool = InputBatchPool(batchCount = 64, maxSamplesPerBatch = 256)
    private val inputQueue = ArrayBlockingQueue<InputBatch>(64)
    private val inputCapturer = StrokeInputCapturer(layerStack.camera, inputPool, inputQueue)

    private var brushSettings = BrushSettings(
        name = "G-Pen",
        renderMode = BrushRenderMode.RIBBON,
        baseRadiusPx = 9f,
        spacing = 0.12f,
        colorArgb = 0xFF1B1F24L,
        smoothing = 0.35f,
        streamline = 0.22f,
        pressureToOpacity = false,
        pressureGamma = 1.7f,
        minSizeRatio = 0.06f,
        ribbon = com.wetinknext.engine.brush.RibbonSettings(
            cap = com.wetinknext.engine.brush.RibbonCap.ROUND,
            join = com.wetinknext.engine.brush.RibbonJoin.ROUND,
            miterLimit = 2.5f,
            minPointDistancePx = 1.25f,
            aaWidthPx = 1f,
            taperStartPx = 16f,
            taperEndPx = 64f,
        ),
    )
    private val stampEmitter = StampEmitter(brushSettings)
    private val capsuleEmitter = CapsuleEmitter(brushSettings)
    private val dabBuffer = DabBuffer()
    private val strokeColorLinear = FloatArray(3)
    private val strokeDirtyRect = DirtyRect()
    private val dirtyBounds = IntArray(4)
```

```
    private var strokeActive = false
    private var pendingBrush: BrushSettings? = null
    private var strokeBlitter: StrokeBlitter? = null
    private val cancelRequested = AtomicBoolean(false)

    /** Assigned by PaintSurfaceView; invoked on the GL thread. */
    var onStateChange: ((EditorUiState) -> Unit)? = null

    fun onTouchEvent(event: MotionEvent): Boolean = inputCapturer.onTouchEvent(event)

    /** These methods are called from GLSurfaceView.queueEvent by the UI layer. */
    fun undo() {
        resetStroke()
        val entry = undoManager.popUndo() ?: return
        val layer = layerStack.findLayerById(entry.layerId) ?: return
        TileSnapshotRestore.restore(layer.target, entry.beforeTiles)
        layer.version++
        publishState()
    }

    fun redo() {
        resetStroke()
        val entry = undoManager.popRedo() ?: return
        val layer = layerStack.findLayerById(entry.layerId) ?: return
        TileSnapshotRestore.restore(layer.target, entry.afterTiles)
        layer.version++
        publishState()
    }

    fun setActiveLayer(id: Long): Boolean {
        resetStroke()
        return layerStack.setActive(id).also { changed ->
            if (changed) publishState()
        }
    }

    fun addLayer(): Long = addLayer(nextLayerName())

    fun addLayer(name: String): Long {
        resetStroke()
        return layerStack.addLayer(name).id.also { publishState() }
    }

    fun removeLayer(id: Long): Boolean {
        resetStroke()
        return (layerStack.removeLayer(id) != null).also { removed ->
            if (removed) publishState()
        }
    }

    fun setLayerVisible(id: Long, visible: Boolean) {
        resetStroke()
        val layer = layerStack.findLayerById(id) ?: return
        layer.isVisible = visible
        publishState()
    }

    fun setLayerOpacity(id: Long, opacity: Float) {
        val layer = layerStack.findLayerById(id) ?: return
        layer.opacity = opacity.coerceIn(0f, 1f)
        publishState()
    }

    /** px = ******** ****** * canvas-*********, ****** **, *** *********** UI. */
    fun setBrushSize(px: Float) =
        updateBrush(brushSettings.copy(baseRadiusPx = (px * .5f).coerceIn(.5f, 200f)))

    fun setBrushOpacity(opacity: Float) =
        updateBrush(brushSettings.copy(opacity = opacity.coerceIn(0f, 1f)))

    /** ******* ****** ** ****** ******* ******* ** ********* ******: ********** ** ********** DOWN. */
    private fun updateBrush(next: BrushSettings) {
        brushSettings = next
        if (strokeActive) pendingBrush = next else applyBrushToEmitters(next)
        publishState()
    }

    private fun applyBrushToEmitters(settings: BrushSettings) {
        stampEmitter.updateSettings(settings)
        capsuleEmitter.updateSettings(settings)
    }

    fun requestState() {
        publishState()
    }

    override fun onSurfaceCreated(gl: GL10?, config: EGLConfig?) {
        releaseGlObjects()
        val nextCaps = GlCaps.query()
        caps = nextCaps
```

```
        layerStack.create(nextCaps, documentWidth, documentHeight)
        strokeTarget.create(documentWidth, documentHeight, nextCaps.supportsHalfFloatColorBuffer)
        strokeTarget.clear(0f, 0f, 0f, 0f)
        ViewTransform.buildCanvasToFbo(
            documentWidth.toFloat(),
            documentHeight.toFloat(),
            canvasToFboMatrix,
        )
        geometry = CanvasGeometry().also { it.create(documentWidth, documentHeight) }
        compositor = Compositor().also { it.create() }
        dabRenderer = DabRenderer(dabBuffer.capacity).also { it.create() }
        capsuleRenderer = CapsuleStrokeRenderer().also {
            it.create()
        }
        strokeBlitter = StrokeBlitter().also { it.create() }
        ColorSpaces.srgb8ToLinear(brushSettings.colorArgb, strokeColorLinear)
        publishState()
        GlCheck.noError("P6 surface creation")
    }

    override fun onSurfaceChanged(gl: GL10?, width: Int, height: Int) {
        screenWidth = width.coerceAtLeast(1)
        screenHeight = height.coerceAtLeast(1)
        GLES30.glViewport(0, 0, screenWidth, screenHeight)
        layerStack.camera.fitCanvas(layerStack.canvasWidth, layerStack.canvasHeight, screenWidth, screenHeight)
        GlCheck.noError("P6 surface changed")
    }

    override fun onDrawFrame(gl: GL10?) {
        if (cancelRequested.compareAndSet(true, false)) resetStroke()
        drainInput()

        if (strokeActive && brushSettings.renderMode == BrushRenderMode.RIBBON) {
            renderCapsulePreview()
        } else if (strokeActive && dabBuffer.count > 0) {
            strokeTarget.clear(0f, 0f, 0f, 0f)
            dabRenderer?.drawInto(
                target = strokeTarget,
                width = layerStack.canvasWidth,
                height = layerStack.canvasHeight,
                canvasToFbo = canvasToFboMatrix,
                dabs = dabBuffer,
                colorLinear = strokeColorLinear,
            )
        }

        GLES30.glBindFramebuffer(GLES30.GL_FRAMEBUFFER, 0)
        GLES30.glViewport(0, 0, screenWidth, screenHeight)
        GLES30.glDisable(GLES30.GL_SCISSOR_TEST)
        GLES30.glDisable(GLES30.GL_BLEND)
        GLES30.glClearColor(0.08f, 0.09f, 0.12f, 1f)
        GLES30.glClear(GLES30.GL_COLOR_BUFFER_BIT)

        layerStack.camera.snapshot().buildCanvasToClip(
            screenWidth.toFloat(),
            screenHeight.toFloat(),
            canvasToClipMatrix,
        )
        compositor?.render(
            geometry = geometry ?: return,
            layers = layerStack,
            activeLayerId = layerStack.activeLayerId,
            strokeTextureId = if (
                strokeActive &&
                (
                    dabBuffer.count > 0 ||
                        (
                            brushSettings.renderMode == BrushRenderMode.RIBBON &&
                                capsuleEmitter.hasStroke
                            )
                    )
            ) {
                strokeTarget.textureId
            } else {
                0
            },
            strokeOpacity = (brushSettings.opacity * brushSettings.flow).coerceIn(0f, 1f),
            canvasToClip = canvasToClipMatrix,
        )
    }

    /** Safe to call from the UI thread; cancellation is executed on the next GL frame. */
    fun cancelActiveStroke() {
        cancelRequested.set(true)
    }

    /** Must run on the GL thread while the context is still current. */
    fun releaseGlObjects() {
        discardPendingInput()
        strokeActive = false
```

```
        stampEmitter.reset()
        dabBuffer.clear()
        undoManager.clear()
        dabRenderer?.release()
        dabRenderer = null
        capsuleRenderer?.release()
        capsuleRenderer = null
        strokeBlitter?.release()
        strokeBlitter = null
        pendingBrush = null
        compositor?.release()
        compositor = null
        geometry?.release()
        geometry = null
        strokeTarget.release()
        layerStack.release()
        caps = null
    }

    private fun drainInput() {
        while (true) {
            val batch = inputQueue.poll() ?: return
            try {
                when (batch.action) {
                    InputAction.DOWN -> {
                        if (!batch.isEmpty()) {
                            resetStroke()
                            pendingBrush?.let { applyBrushToEmitters(it); pendingBrush = null }
                            strokeActive = true

                            ColorSpaces.srgb8ToLinear(
                                brushSettings.colorArgb,
                                strokeColorLinear,
                            )

                            if (brushSettings.renderMode == BrushRenderMode.RIBBON) {
                                capsuleEmitter.begin(
                                    batch = batch,
                                    out = checkNotNull(capsuleRenderer),
                                )
                            } else {
                                stampEmitter.begin(batch, dabBuffer)
                            }
                        }
                    }

                    InputAction.MOVE -> if (strokeActive) {
                        if (brushSettings.renderMode == BrushRenderMode.RIBBON) {
                            capsuleEmitter.append(
                                batch = batch,
                                out = checkNotNull(capsuleRenderer),
                            )
                        } else {
                            stampEmitter.append(batch, dabBuffer)
                        }
                    }

                    InputAction.UP -> if (strokeActive) {
                        if (brushSettings.renderMode == BrushRenderMode.RIBBON) {
                            val renderer = checkNotNull(capsuleRenderer)

                            // UP [redacted] [redacted] [redacted] historical samples,
                            // [redacted] [redacted] [redacted], [redacted] [redacted] [redacted] capturer.
                            capsuleEmitter.append(
                                batch = batch,
                                out = renderer,
                            )
                            capsuleEmitter.finish(
                                out = renderer,
                                cancel = false,
                            )
                            commitCapsuleStroke()
                        } else {
                            stampEmitter.append(batch, dabBuffer)
                            stampEmitter.finish(dabBuffer, cancel = false)
                            commitStroke()
                        }

                        resetStroke()
                    }

                    InputAction.CANCEL -> resetStroke()
                }
            } finally {
                inputPool.release(batch)
            }
        }
    }

    private fun commitStroke() {
```

```
    val layer = layerStack.activeLayer() ?: return
    if (layer.isLocked || dabBuffer.count == 0) return
    if (!computeDirtyPixelBounds(dirtyBounds)) return
    val renderer = dabRenderer ?: return

    // Both snapshots use the actual layer format (RGBA8 or RGBA16F).
    val beforeTiles = TileSnapshotCapture.capture(layer.target, dirtyBounds)
    renderer.drawInto(
        target = layer.target,
        width = layerStack.canvasWidth,
        height = layerStack.canvasHeight,
        canvasToFbo = canvasToFboMatrix,
        dabs = dabBuffer,
        colorLinear = strokeColorLinear,
    )
    layer.version++
    val afterTiles = TileSnapshotCapture.capture(layer.target, dirtyBounds)
    undoManager.push(UndoEntry(layer.id, beforeTiles, afterTiles))
    publishState()
}

/** Returns a clamped pixel region covering the exact emitted dabs and their AA fringe. */
private fun computeDirtyPixelBounds(out: IntArray): Boolean {
    if (dabBuffer.count == 0) return false
    strokeDirtyRect.clear()
    for (index in 0 until dabBuffer.count) {
        val offset = index * DabBuffer.FLOATS_PER_DAB
        val x = dabBuffer.floats.get(offset)
        val y = dabBuffer.floats.get(offset + 1)
        val radius = dabBuffer.floats.get(offset + 2)
        strokeDirtyRect.include(x, y, radius)
    }
    strokeDirtyRect.expand(AA_MARGIN_PX)
    strokeDirtyRect.clamp(layerStack.canvasWidth.toFloat(), layerStack.canvasHeight.toFloat())
    strokeDirtyRect.toPixelBounds(out)
    return out[2] > out[0] && out[3] > out[1]
}

private fun resetStroke() {
    strokeActive = false
    stampEmitter.reset()
    dabBuffer.clear()
    capsuleEmitter.reset()
    capsuleRenderer?.clearStrokeData()
    if (strokeTarget.framebufferId != 0) {
        strokeTarget.clear(0f, 0f, 0f, 0f)
    }
}

/** Releases pooled batches without interpreting them during shutdown/context recreation. */
private fun discardPendingInput() {
    while (true) {
        val batch = inputQueue.poll() ?: return
        inputPool.release(batch)
    }
}

private fun publishState() {
    val listener = onStateChange ?: return
    val layers = layerStack.allLayers()
    val activeLayerId = layerStack.activeLayerId
    val canDeleteLayers = layers.size > 1
    listener(
        EditorUiState(
            layers = layers.map { layer ->
                LayerUiModel(
                    id = layer.id,
                    name = layer.name,
                    isVisible = layer.isVisible,
                    isLocked = layer.isLocked,
                    opacity = layer.opacity,
                    active = layer.id == activeLayerId,
                    canDelete = !layer.isLocked && canDeleteLayers,
                )
            },
            canUndo = undoManager.canUndo,
            canRedo = undoManager.canRedo,
            brushSizePx = brushSettings.baseRadiusPx * 2f,
            brushOpacity = brushSettings.opacity,
            activeLayerId = activeLayerId,
            ready = layerStack.canvasWidth > 0 && layerStack.canvasHeight > 0,
        ),
    )
}

private fun nextLayerName(): String = "■■■■ ${layerStack.count + 1}"

private fun renderCapsulePreview() {
    val renderer = capsuleRenderer ?: return
    // ■■■■■■ ■■■■■■■■■■■ drawAll, ■■■■■■ ■■■ strokeTarget ■■■■■■■■■■
```

```
    // [redacted]. [redacted], [redacted] [redacted] [redacted] [redacted] mesh-[redacted].
    strokeTarget.clear(
        red = 0f,
        green = 0f,
        blue = 0f,
        alpha = 0f,
    )

    renderer.drawAll(
        target = strokeTarget,
        width = layerStack.canvasWidth,
        height = layerStack.canvasHeight,
        canvasToClip = canvasToFboMatrix,
        colorLinear = strokeColorLinear,
    )
}

private fun commitCapsuleStroke() {
    val layer = layerStack.activeLayer() ?: return
    val renderer = capsuleRenderer ?: return

    if (layer.isLocked) return
    if (!capsuleEmitter.hasStroke) return
    if (!computeCapsuleDirtyBounds(dirtyBounds)) return

    /*
     * [redacted] [redacted] stroke [redacted] [redacted] offscreen target.
     * [redacted] [redacted] [redacted] [redacted] renderer [redacted] [redacted] [redacted] FBO [redacted] [redacted].
     */
    strokeTarget.clear(
        red = 0f,
        green = 0f,
        blue = 0f,
        alpha = 0f,
    )

    renderer.drawAll(
        target = strokeTarget,
        width = layerStack.canvasWidth,
        height = layerStack.canvasHeight,
        canvasToClip = canvasToFboMatrix,
        colorLinear = strokeColorLinear,
    )

    val before = TileSnapshotCapture.capture(
        layer.target,
        dirtyBounds,
    )

    /*
     * [redacted] [redacted] StrokeBlitter:
     * strokeTarget -> layer.target
     *
     * [redacted] [redacted] [redacted]:
     * GL_FUNC_ADD
     * GL_ONE, GL_ONE_MINUS_SRC_ALPHA
     * premultiplied source-over
     * opacity = brushSettings.opacity * brushSettings.flow
     */
    strokeBlitter?.blit(
        layer = layer.target,
        geometry = checkNotNull(geometry),
        strokeTextureId = strokeTarget.textureId,
        canvasToFbo = canvasToFboMatrix,
        width = layerStack.canvasWidth,
        height = layerStack.canvasHeight,
        opacity = (
            brushSettings.opacity * brushSettings.flow
        ).coerceIn(0f, 1f),
    ) ?: error(
        "Capsule commit requires StrokeBlitter"
    )

    layer.version++

    val after = TileSnapshotCapture.capture(
        layer.target,
        dirtyBounds,
    )

    undoManager.push(
        UndoEntry(
            layerId = layer.id,
            beforeTiles = before,
            afterTiles = after,
        ),
    )

    publishState()
}
```

```
    private fun computeCapsuleDirtyBounds(
        out: IntArray,
    ): Boolean {
        if (!capsuleEmitter.hasBounds) return false

        strokeDirtyRect.clear()
        strokeDirtyRect.include(
            left = capsuleEmitter.minX,
            top = capsuleEmitter.minY,
            right = capsuleEmitter.maxX,
            bottom = capsuleEmitter.maxY,
        )
        strokeDirtyRect.expand(AA_MARGIN_PX)
        strokeDirtyRect.clamp(
            layerStack.canvasWidth.toFloat(),
            layerStack.canvasHeight.toFloat(),
        )
        strokeDirtyRect.toPixelBounds(out)

        return out[2] > out[0] && out[3] > out[1]
    }

    companion object {
        const val DEFAULT_CANVAS_WIDTH = 1500
        const val DEFAULT_CANVAS_HEIGHT = 2000
        private const val AA_MARGIN_PX = 2f
    }
}
```

app/src/main/java/com/wetinknext/engine/core/PaintEngine.kt

```
package com.wetinknext.engine.core

import com.wetinknext.engine.canvas.LayerStack
import com.wetinknext.engine.gl.GlCaps
import com.wetinknext.engine.gl.RenderTarget

/** Lightweight document facade retained for engine callers outside the renderer. */
class PaintEngine {
    val layerStack = LayerStack()
    val camera: Camera get() = layerStack.camera
    val canvasTarget: RenderTarget get() = checkNotNull(layerStack.activeLayer()).target

    val canvasWidth: Int get() = layerStack.canvasWidth
    val canvasHeight: Int get() = layerStack.canvasHeight
    var initialized = false
        private set

    fun create(caps: GlCaps, width: Int, height: Int) {
        layerStack.create(caps, width, height)
        initialized = true
    }

    fun addLayer(name: String): Long = layerStack.addLayer(name).id

    fun removeLayer(id: Long): Boolean = layerStack.removeLayer(id) != null

    fun setActiveLayer(id: Long): Boolean = layerStack.setActive(id)

    fun release() {
        layerStack.release()
        initialized = false
    }
}
```

app/src/main/java/com/wetinknext/engine/core/PaintSurfaceView.kt

```
package com.wetinknext.engine.core

import android.content.Context
import android.opengl.GLSurfaceView
import android.util.AttributeSet
import android.view.MotionEvent

/** Thread boundary between Compose commands and the GL-owned paint engine. */
class PaintSurfaceView @JvmOverloads constructor(
    context: Context,
    attrs: AttributeSet? = null,
) : GLSurfaceView(context, attrs) {
    private val engineRenderer = EngineRenderer()

    var onEditorStateChange: ((EditorUiState) -> Unit)? = null

    init {
        setEGLContextClientVersion(3)
        setRenderer(engineRenderer)
        renderMode = RENDERMODE_CONTINUOUSLY
        engineRenderer.onStateChange = { state ->
            post { onEditorStateChange?.invoke(state) }
        }
    }

    override fun onTouchEvent(event: MotionEvent): Boolean =
        engineRenderer.onTouchEvent(event) || super.onTouchEvent(event)

    fun requestState() = queueEvent { engineRenderer.requestState() }

    fun undo() = queueEvent { engineRenderer.undo() }

    fun redo() = queueEvent { engineRenderer.redo() }

    fun setActiveLayer(id: Long) = queueEvent { engineRenderer.setActiveLayer(id) }

    fun addLayer() = queueEvent { engineRenderer.addLayer() }

    fun removeLayer(id: Long) = queueEvent { engineRenderer.removeLayer(id) }

    fun setLayerVisible(id: Long, visible: Boolean) =
        queueEvent { engineRenderer.setLayerVisible(id, visible) }

    fun setLayerOpacity(id: Long, opacity: Float) =
        queueEvent { engineRenderer.setLayerOpacity(id, opacity) }

    fun setBrushSize(px: Float) = queueEvent { engineRenderer.setBrushSize(px) }

    fun setBrushOpacity(opacity: Float) = queueEvent { engineRenderer.setBrushOpacity(opacity) }

    override fun onPause() {
        engineRenderer.cancelActiveStroke()
        super.onPause()
    }

    override fun onDetachedFromWindow() {
        queueEvent { engineRenderer.releaseGlObjects() }
        super.onDetachedFromWindow()
    }
}
```

app/src/main/java/com/wetinknext/engine/core/ViewTransform.kt

```
package com.wetinknext.engine.core

import kotlin.math.abs
import kotlin.math.cos
import kotlin.math.sin

/** Canonical canvas-to-view-local-screen transformation. */
data class ViewTransform(
    val translateX: Float = 0f,
    val translateY: Float = 0f,
    val scale: Float = 1f,
    val rotationRad: Float = 0f,
    val flipX: Boolean = false,
) {
    private val flipSign: Float
        get() = if (flipX) -1f else 1f

    val m00: Float get() = cos(rotationRad) * scale * flipSign
    val m01: Float get() = -sin(rotationRad) * scale
    val m10: Float get() = sin(rotationRad) * scale * flipSign
    val m11: Float get() = cos(rotationRad) * scale

    fun canvasToScreen(canvasX: Float, canvasY: Float, out: FloatArray) {
        require(out.size >= 2)
        out[0] = m00 * canvasX + m01 * canvasY + translateX
        out[1] = m10 * canvasX + m11 * canvasY + translateY
    }

    fun screenToCanvas(screenX: Float, screenY: Float, out: FloatArray) {
        require(out.size >= 2)
        val determinant = m00 * m11 - m01 * m10
        if (abs(determinant) < 1e-8f) {
            out[0] = 0f
            out[1] = 0f
            return
        }
        val inverseDeterminant = 1f / determinant
        val dx = screenX - translateX
        val dy = screenY - translateY
        out[0] = (m11 * dx - m01 * dy) * inverseDeterminant
        out[1] = (-m10 * dx + m00 * dy) * inverseDeterminant
    }

    fun zoomAround(anchorX: Float, anchorY: Float, newScale: Float): ViewTransform {
        val safeScale = newScale.coerceIn(MIN_SCALE, MAX_SCALE)
        val factor = safeScale / scale
        return copy(
            scale = safeScale,
            translateX = anchorX - (anchorX - translateX) * factor,
            translateY = anchorY - (anchorY - translateY) * factor,
        )
    }

    fun rotateAround(anchorX: Float, anchorY: Float, deltaRad: Float): ViewTransform {
        val c = cos(deltaRad)
        val s = sin(deltaRad)
        val dx = translateX - anchorX
        val dy = translateY - anchorY
        return copy(
            rotationRad = rotationRad + deltaRad,
            translateX = anchorX + dx * c - dy * s,
            translateY = anchorY + dx * s + dy * c,
        )
    }

    /** Builds a column-major matrix for canvas to OpenGL clip coordinates. */
    fun buildCanvasToClip(viewWidth: Float, viewHeight: Float, out: FloatArray) {
        require(out.size >= 16)
        require(viewWidth > 0f)
        require(viewHeight > 0f)
        val sx = 2f / viewWidth
        val sy = -2f / viewHeight
        out[0] = m00 * sx; out[1] = m10 * sy; out[2] = 0f; out[3] = 0f
        out[4] = m01 * sx; out[5] = m11 * sy; out[6] = 0f; out[7] = 0f
        out[8] = 0f; out[9] = 0f; out[10] = 1f; out[11] = 0f
        out[12] = translateX * sx - 1f; out[13] = translateY * sy + 1f
        out[14] = 0f; out[15] = 1f
    }

    companion object {
        const val MIN_SCALE = 0.02f
        const val MAX_SCALE = 64f

        fun buildCanvasToFbo(canvasWidth: Float, canvasHeight: Float, out: FloatArray) {
            require(out.size >= 16); require(canvasWidth > 0f); require(canvasHeight > 0f)
            out[0]=2f/canvasWidth;out[1]=0f;out[2]=0f;out[3]=0f;out[4]=0f;out[5]=2f/canvasHeight;out[6]=0f;out[7]=0f;
out[8]=0f;out[9]=0f;out[10]=1f;out[11]=0f;out[12]=-1f;out[13]=-1f;out[14]=0f;out[15]=1f
```

```
        }
    }
}
```

app/src/main/java/com/wetinknext/engine/gl/CanvasGeometry.kt

```
package com.wetinknext.engine.gl

import android.opengl.GLES30
import java.nio.ByteBuffer
import java.nio.ByteOrder

/** A document-sized quad; GL resources are owned by the render thread. */
class CanvasGeometry {
    var vaoId = 0
        private set
    private var vboId = 0

    fun create(width: Int, height: Int) {
        release()
        val vertices = floatArrayOf(0f, 0f, width.toFloat(), 0f, 0f, height.toFloat(), width.toFloat(), height.
toFloat())
        val data = ByteBuffer.allocateDirect(vertices.size * Float.SIZE_BYTES).order(ByteOrder.nativeOrder()).
asFloatBuffer()
        data.put(vertices).position(0)
        val ids = IntArray(1)
        GLES30.glGenVertexArrays(1, ids, 0); vaoId = ids[0]
        GLES30.glGenBuffers(1, ids, 0); vboId = ids[0]
        GLES30.glBindVertexArray(vaoId); GLES30.glBindBuffer(GLES30.GL_ARRAY_BUFFER, vboId)
        GLES30.glBufferData(GLES30.GL_ARRAY_BUFFER, vertices.size * Float.SIZE_BYTES, data, GLES30.GL_STATIC_DRAW)
        GLES30.glEnableVertexAttribArray(0); GLES30.glVertexAttribPointer(0, 2, GLES30.GL_FLOAT, false, 0, 0)
        GLES30.glBindBuffer(GLES30.GL_ARRAY_BUFFER, 0); GLES30.glBindVertexArray(0)
    }

    fun draw() {
        check(vaoId != 0)
        GLES30.glBindVertexArray(vaoId); GLES30.glDrawArrays(GLES30.GL_TRIANGLE_STRIP, 0, 4); GLES30.
glBindVertexArray(0)
    }

    fun release() {
        if (vboId != 0) GLES30.glDeleteBuffers(1, intArrayOf(vboId), 0)
        if (vaoId != 0) GLES30.glDeleteVertexArrays(1, intArrayOf(vaoId), 0)
        vaoId = 0; vboId = 0
    }
}
```

app/src/main/java/com/wetinknext/engine/gl/GeometryCache.kt

```
package com.wetinknext.engine.gl

import android.opengl.GLES30
import java.nio.ByteBuffer
import java.nio.ByteOrder

/** A single cached fullscreen triangle for future present passes. */
class GeometryCache {
    private var vaoId = 0
    private var vboId = 0
    fun create() {
        if (vaoId != 0) return
        val vertices = floatArrayOf(-1f, -1f, 3f, -1f, -1f, 3f)
        val data = ByteBuffer.allocateDirect(vertices.size * Float.SIZE_BYTES).order(ByteOrder.nativeOrder()).
asFloatBuffer()
        data.put(vertices).position(0)
        val ids = IntArray(1); GLES30.glGenVertexArrays(1, ids, 0); vaoId = ids[0]
        GLES30.glGenBuffers(1, ids, 0); vboId = ids[0]
        GLES30.glBindVertexArray(vaoId); GLES30.glBindBuffer(GLES30.GL_ARRAY_BUFFER, vboId)
        GLES30.glBufferData(GLES30.GL_ARRAY_BUFFER, vertices.size * Float.SIZE_BYTES, data, GLES30.GL_STATIC_DRAW)
        GLES30.glEnableVertexAttribArray(0); GLES30.glVertexAttribPointer(0, 2, GLES30.GL_FLOAT, false, 0, 0)
        GLES30.glBindBuffer(GLES30.GL_ARRAY_BUFFER, 0); GLES30.glBindVertexArray(0)
    }
    fun drawFullscreenTriangle() { check(vaoId != 0); GLES30.glBindVertexArray(vaoId); GLES30.glDrawArrays(GLES30.
GL_TRIANGLES, 0, 3); GLES30.glBindVertexArray(0) }
    fun release() { if (vboId != 0) GLES30.glDeleteBuffers(1, intArrayOf(vboId), 0); if (vaoId != 0) GLES30.
glDeleteVertexArrays(1, intArrayOf(vaoId), 0); vboId = 0; vaoId = 0 }
}
```

app/src/main/java/com/wetinknext/engine/gl/GlCaps.kt

```
package com.wetinknext.engine.gl

import android.opengl.GLES30

/** Capabilities queried only after a GLES 3 context is current. */
data class GlCaps(
    val renderer: String,
    val version: String,
    val supportsHalfFloatColorBuffer: Boolean,
) {
    companion object {
        fun query(): GlCaps {
            val extensions = GLES30.glGetString(GLES30.GL_EXTENSIONS).orEmpty()
            val declared = extensions.contains("EXT_color_buffer_half_float") ||
                extensions.contains("EXT_color_buffer_float")
            // Extension string [redacted]: [redacted] [redacted] FBO 4x4.
            val supported = declared && RenderTarget().probeHalfFloatColorBuffer()
            return GlCaps(
                renderer = GLES30.glGetString(GLES30.GL_RENDERER).orEmpty(),
                version = GLES30.glGetString(GLES30.GL_VERSION).orEmpty(),
                supportsHalfFloatColorBuffer = supported,
            )
        }
    }
}
```

app/src/main/java/com/wetinknext/engine/gl/GlCheck.kt

```
package com.wetinknext.engine.gl

import android.opengl.GLES30

/** GL error checks intended for setup boundaries, never the render hot path. */
object GlCheck {
    fun noError(label: String) {
        val error = GLES30.glGetError()
        check(error == GLES30.GL_NO_ERROR) { "$label: GL error 0x${error.toString(16)}" }
    }

    fun framebufferComplete(label: String) {
        val status = GLES30.glCheckFramebufferStatus(GLES30.GL_FRAMEBUFFER)
        check(status == GLES30.GL_FRAMEBUFFER_COMPLETE) {
            "$label: framebuffer incomplete (0x${status.toString(16)})"
        }
    }
}
```

app/src/main/java/com/wetinknext/engine/gl/GlProgram.kt

```
package com.wetinknext.engine.gl

import android.opengl.GLES30

/** Linked GLSL program. Construction and release must run on the GL thread. */
class GlProgram(vertexSource: String, fragmentSource: String) {
    val id: Int = GLES30.glCreateProgram().also { program ->
        check(program != 0) { "Unable to create GL program" }
        val vertex = compile(GLES30.GL_VERTEX_SHADER, vertexSource)
        val fragment = compile(GLES30.GL_FRAGMENT_SHADER, fragmentSource)
        GLES30.glAttachShader(program, vertex); GLES30.glAttachShader(program, fragment); GLES30.glLinkProgram(
program)
        val linked = IntArray(1); GLES30.glGetProgramiv(program, GLES30.GL_LINK_STATUS, linked, 0)
        GLES30.glDeleteShader(vertex); GLES30.glDeleteShader(fragment)
        check(linked[0] == GLES30.GL_TRUE) { "Program link failed: ${GLES30.glGetProgramInfoLog(program)}" }
    }
    fun use() = GLES30.glUseProgram(id)
    fun release() = GLES30.glDeleteProgram(id)

    private fun compile(type: Int, source: String): Int = GLES30.glCreateShader(type).also { shader ->
        check(shader != 0) { "Unable to create shader" }
        GLES30.glShaderSource(shader, source); GLES30.glCompileShader(shader)
        val compiled = IntArray(1); GLES30.glGetShaderiv(shader, GLES30.GL_COMPILE_STATUS, compiled, 0)
        check(compiled[0] == GLES30.GL_TRUE) { "Shader compile failed: ${GLES30.glGetShaderInfoLog(shader)}" }
    }
}
```

app/src/main/java/com/wetinknext/engine/gl/RenderTarget.kt

```
package com.wetinknext.engine.gl

import android.opengl.GLES30

/** Texture-backed FBO with RGBA16F probing and a guaranteed RGBA8 fallback. */
class RenderTarget {
    var framebufferId: Int = 0
        private set
    var textureId: Int = 0
        private set
    var width: Int = 0
        private set
    var height: Int = 0
        private set
    var usesHalfFloat: Boolean = false
        private set

    fun create(width: Int, height: Int, preferHalfFloat: Boolean) {
        require(width > 0 && height > 0)
        release()
        if (preferHalfFloat && createInternal(width, height, true)) return
        check(createInternal(width, height, false)) { "RGBA8 render target creation failed" }
    }

    /** A 4x4 completeness probe used before allocating a document-sized target. */
    fun probeHalfFloatColorBuffer(): Boolean {
        release()
        val result = createInternal(4, 4, true)
        release()
        return result
    }

    fun bind() {
        check(framebufferId != 0) { "RenderTarget is not created" }
        GLES30.glBindFramebuffer(GLES30.GL_FRAMEBUFFER, framebufferId)
    }

    fun clear(red: Float, green: Float, blue: Float, alpha: Float) {
        bind()
        GLES30.glDisable(GLES30.GL_SCISSOR_TEST)
        GLES30.glDisable(GLES30.GL_BLEND)
        GLES30.glClearColor(red, green, blue, alpha)
        GLES30.glClear(GLES30.GL_COLOR_BUFFER_BIT)
    }

    fun release() {
        if (framebufferId != 0) GLES30.glDeleteFramebuffers(1, intArrayOf(framebufferId), 0)
        if (textureId != 0) GLES30.glDeleteTextures(1, intArrayOf(textureId), 0)
        framebufferId = 0; textureId = 0; width = 0; height = 0; usesHalfFloat = false
    }

    private fun createInternal(targetWidth: Int, targetHeight: Int, halfFloat: Boolean): Boolean {
        val textures = IntArray(1)
        val framebuffers = IntArray(1)
        GLES30.glGenTextures(1, textures, 0)
        GLES30.glGenFramebuffers(1, framebuffers, 0)
        val texture = textures[0]
        val framebuffer = framebuffers[0]
        if (texture == 0 || framebuffer == 0) return false
        GLES30.glBindTexture(GLES30.GL_TEXTURE_2D, texture)
        GLES30.glTexParameteri(GLES30.GL_TEXTURE_2D, GLES30.GL_TEXTURE_MIN_FILTER, GLES30.GL_LINEAR)
        GLES30.glTexParameteri(GLES30.GL_TEXTURE_2D, GLES30.GL_TEXTURE_MAG_FILTER, GLES30.GL_LINEAR)
        GLES30.glTexParameteri(GLES30.GL_TEXTURE_2D, GLES30.GL_TEXTURE_WRAP_S, GLES30.GL_CLAMP_TO_EDGE)
        GLES30.glTexParameteri(GLES30.GL_TEXTURE_2D, GLES30.GL_TEXTURE_WRAP_T, GLES30.GL_CLAMP_TO_EDGE)
        GLES30.glTexImage2D(GLES30.GL_TEXTURE_2D, 0, if (halfFloat) GLES30.GL_RGBA16F else GLES30.GL_RGBA8,
            targetWidth, targetHeight, 0, GLES30.GL_RGBA, if (halfFloat) GLES30.GL_HALF_FLOAT else GLES30.
GL_UNSIGNED_BYTE, null)
        GLES30.glBindFramebuffer(GLES30.GL_FRAMEBUFFER, framebuffer)
        GLES30.glFramebufferTexture2D(GLES30.GL_FRAMEBUFFER, GLES30.GL_COLOR_ATTACHMENT0, GLES30.GL_TEXTURE_2D,
texture, 0)
        val complete = GLES30.glCheckFramebufferStatus(GLES30.GL_FRAMEBUFFER) == GLES30.GL_FRAMEBUFFER_COMPLETE
        GLES30.glBindFramebuffer(GLES30.GL_FRAMEBUFFER, 0)
        GLES30.glBindTexture(GLES30.GL_TEXTURE_2D, 0)
        if (!complete) {
            GLES30.glDeleteFramebuffers(1, framebuffers, 0); GLES30.glDeleteTextures(1, textures, 0)
            return false
        }
        framebufferId = framebuffer; textureId = texture; width = targetWidth; height = targetHeight; usesHalfFloat =
halfFloat
        return true
    }
}
```

app/src/main/java/com/wetinknext/engine/gl/ShaderLib.kt

```
package com.wetinknext.engine.gl

object ShaderLib {
    const val compositorVertex="""#version 300 es
        layout(location=0) in vec2 aPosition;uniform mat4 uCanvasToClip;uniform vec2 uCanvasSize;out vec2 vUv;void
main(){vUv=aPosition/uCanvasSize;gl_Position=uCanvasToClip*vec4(aPosition,0.,1.);}"""
    const val compositorFragment = """#version 300 es
        precision highp float;

        in vec2 vUv;
        uniform sampler2D uLayerTex;
        uniform sampler2D uStrokeTex;
        uniform int uStrokeActive;
        uniform float uOpacity;
        uniform float uStrokeOpacity;

        out vec4 fragColor;

        vec3 linearToSrgb(vec3 color) {
            vec3 c = clamp(color, 0.0, 1.0);
            vec3 low = c * 12.92;
            vec3 high = 1.055 * pow(c, vec3(1.0 / 2.4)) - 0.055;
            return mix(low, high, step(vec3(0.0031308), c));
        }

        vec3 unpremultiply(vec4 color) {
            return color.a <= 0.0001 ? vec3(0.0) : color.rgb / color.a;
        }

        void main() {
            vec4 layer = texture(uLayerTex, vUv);
            if (uStrokeActive == 1) {
                vec4 stroke = texture(uStrokeTex, vUv) * uStrokeOpacity;
                layer = stroke + layer * (1.0 - stroke.a);
            }

            float alpha = layer.a * uOpacity;
            fragColor = vec4(linearToSrgb(unpremultiply(layer)) * alpha, alpha);
        }
    """
    const val dabVertex = """#version 300 es
        layout(location = 0) in vec2 aCorner;
        layout(location = 1) in vec4 iDab0;
        layout(location = 2) in float iAlpha;
        uniform mat4 uCanvasToClip;
        out vec2 vLocal; flat out float vAlpha;
        void main(){ float c=cos(iDab0.w), s=sin(iDab0.w); vec2 r=vec2(c*aCorner.x-s*aCorner.y,s*aCorner.x+c*aCorner.
y); vLocal=aCorner;vAlpha=iAlpha;gl_Position=uCanvasToClip*vec4(iDab0.xy+r*iDab0.z,0.,1.); }
    """
    const val dabFragment = """#version 300 es
        precision highp float; in vec2 vLocal; flat in float vAlpha; uniform vec3 uColorLinear; out vec4 fragColor;
        void main(){ float r=length(vLocal), aa=max(fwidth(r),.001), a=vAlpha*(1.-smoothstep(1.-aa,1.+aa,r));
fragColor=vec4(uColorLinear*a,a); }
    """
    const val strokeCompositeFragment = """#version 300 es
        precision highp float; in vec2 vUv; uniform sampler2D uCanvasTex; uniform sampler2D uStrokeTex; uniform int
uStrokeActive; out vec4 fragColor;
        vec3 toSrgb(vec3 c){c=clamp(c,0.,1.);return mix(c*12.92,1.055*pow(c,vec3(1./2.4))-.055,step(vec3(.0031308),
c));}
        vec3 unp(vec4 c){return c.a<=.0001?vec3(0.):c.rgb/c.a;}
        void main(){vec4 c=texture(uCanvasTex,vUv);vec4 s=texture(uStrokeTex,vUv);float a=uStrokeActive==1?s.a+c.a*(1.
-s.a):c.a;vec3 rgb=uStrokeActive==1&&a>.0001?(unp(s)*s.a+unp(c)*c.a*(1.-s.a))/a:unp(c);fragColor=vec4(toSrgb(rgb),1.);
}
    """
    const val canvasPresentVertex = """#version 300 es
        layout(location = 0) in vec2 aPosition;
        uniform mat4 uCanvasToClip;
        uniform vec2 uCanvasSize;
        out vec2 vUv;
        void main() {
            vUv = aPosition / uCanvasSize;
            gl_Position = uCanvasToClip * vec4(aPosition, 0.0, 1.0);
        }
    """
    const val presentFragment = """#version 300 es
        precision highp float;
        in vec2 vUv;
        uniform sampler2D uTexture;
        out vec4 fragColor;
        void main() { fragColor = texture(uTexture, vUv); }
    """
    const val fullscreenVertex = """#version 300 es
        layout(location = 0) in vec2 aPosition;
        out vec2 vUv;
        void main() { vUv = aPosition * 0.5 + 0.5; gl_Position = vec4(aPosition, 0.0, 1.0); }
    """
    const val solidFragment = """#version 300 es
```

```
        precision highp float;
        in vec2 vUv;
        out vec4 fragColor;
        void main() { fragColor = vec4(vUv, 0.0, 1.0); }
    """
    const val ribbonVertex = """#version 300 es
        layout(location=0) in vec2 aPos; layout(location=1) in float aCoverage; layout(location=2) in float aAlpha;
        uniform mat4 uCanvasToClip; out float vCoverage; out float vAlpha;
        void main(){vCoverage=aCoverage;vAlpha=aAlpha;gl_Position=uCanvasToClip*vec4(aPos,0.,1.);}
    """
    const val ribbonFragment = """#version 300 es
        precision highp float; in float vCoverage; in float vAlpha; uniform vec3 uColorLinear; uniform float uFlow;
        uniform int uAntiAliasLevel; uniform int uNoAntialias; out vec4 fragColor;
        void main(){float cov=vCoverage;if(uNoAntialias==1||uAntiAliasLevel==0)cov=step(.5,cov);else{float
e=uAntiAliasLevel==1?.35:uAntiAliasLevel==2?.5:.7;cov=smoothstep(.5-e,.5+e,cov);}float a=vAlpha*cov*uFlow;
fragColor=vec4(uColorLinear*a,a);}
    """

    /**
     * ■■■■ ■■■■■■■■ ■■■■■■■ = ■■■■■■■■ (round cone): ■■■ ■■■■■ ■ ■■■■■■■■■ ■■■■■■■■■■■■■■■ ■■■■■■■■.
     * ■■■■■■■■■■ ■■■■■■■ ■■■■■■■ ■■■■■■■ AABB ■■■■■■■■■, ■■■■■ ■■■■■■■■■■ ■■ ■■■■■■■■■■.
     */
    const val capsuleVertex = """#version 300 es
        layout(location = 0) in vec2 aCorner;   // -1..1, TRIANGLE_STRIP ■■ 4 ■■■■■■
        layout(location = 1) in vec3 iA;        // x, y, radius (■■■■■■■ ■■■■■■■■■)
        layout(location = 2) in vec3 iB;        // x, y, radius (■■■■■ ■■■■■■■■■)
        uniform mat4 uCanvasToClip;
        out vec2 vPos;
        flat out vec3 vA;
        flat out vec3 vB;
        void main() {
            // ■■■■■■ 2 px ■■ AA-■■■■■■■■, ■■■■■■ ■■■■ ■■■■■■■■■■ ■■ ■■■■■■■■ ■■■■■■.
            vec2 lo = min(iA.xy - iA.z, iB.xy - iB.z) - 2.0;
            vec2 hi = max(iA.xy + iA.z, iB.xy + iB.z) + 2.0;
            vec2 p = mix(lo, hi, aCorner * 0.5 + 0.5);
            vPos = p;
            vA = iA;
            vB = iB;
            gl_Position = uCanvasToClip * vec4(p, 0.0, 1.0);
        }
    """

    /**
     * ■■■■■■■■■■■■■■■ ■■■■■■■■■: ■■■ ■■■■■■■, ■■■■■■■■, miter ■ AA-■■■■■ ■■■ ■■■■■■■■■■ ■■■■■■■■■■.
     * ■■■■■■ ■■■■■■■ premultiplied linear: rgb = color * a, a = cov.
     * ■■■■■■■■■ ■■■■■■■ ■ glBlendEquation(GL_MAX) + glBlendFunc(GL_ONE, GL_ONE):
     * MAX ■■■■■■■■■■■■■■, ■■■■■■■■ ■■■■■■■■■■■■ ■■■■■■■■■■■ ■■ ■■■■■■■■■■■■■ ■■■■■■.
     */
    const val capsuleFragment = """#version 300 es
        precision highp float;
        in vec2 vPos;
        flat in vec3 vA;
        flat in vec3 vB;
        uniform vec3 uColorLinear;
        out vec4 fragColor;

        float sdRoundCone(vec2 p, vec2 a, vec2 b, float r1, float r2) {
            vec2 ba = b - a;
            float l2 = dot(ba, ba);
            float rr = r1 - r2;

            // Guard 1: ■■■■■■■■■■■■■ ■■■■■■■■ (■■■, ■■■■■■ ■■■■■■■) -> ■■■■■■■■ ■■■■■.
            if (l2 < 1e-6) return length(p - a) - max(r1, r2);
            // Guard 2: ■■■■■ ■■■■■ ■■■■■■■■■■ ■■■■■■■ (■■■■■■■ ■■■■■■■■■ ■■ ■■■■■■■■■ ■■■■■).
            // ■■■■ ■■■■■■ a2 < 0 -> sqrt(■■■■■■■■■■■■■■■■■) -> NaN -> ■■■■■■■ ■■■■■■■.
            if (rr * rr >= l2) return (r1 > r2) ? (length(p - a) - r1) : (length(p - b) - r2);

            float a2 = l2 - rr * rr;
            float il2 = 1.0 / l2;
            vec2 pa = p - a;
            float y = dot(pa, ba);
            float z = y - l2;
            vec2 xv = pa * l2 - ba * y;
            float x2 = dot(xv, xv);
            float y2 = y * y * l2;
            float z2 = z * z * l2;
            float k = sign(rr) * rr * rr * x2;

            if (sign(z) * a2 * z2 > k) return sqrt(x2 + z2) * il2 - r2;   // ■■■■■■■■ ■■■■■■ ■ B
            if (sign(y) * a2 * y2 < k) return sqrt(x2 + y2) * il2 - r1;   // ■■■■■■■■ ■■■■■■ ■ A
            return (sqrt(x2 * a2 * il2) + y * rr) * il2 - r1;             // ■■■■■■■■■■■■ ■■■■■■■■■
        }

        void main() {
            float d = sdRoundCone(vPos, vA.xy, vB.xy, vA.z, vB.z);
            float w = max(fwidth(d), 1e-4);
            float cov = 1.0 - smoothstep(-w, w, d);
            if (cov <= 0.0) discard;
            fragColor = vec4(uColorLinear * cov, cov);
        }
```

```
    """

    const val strokeBlitFragment = """#version 300 es
        precision highp float;
        in vec2 vUv;
        uniform sampler2D uStrokeTex;
        uniform float uOpacity;
        out vec4 fragColor;
        void main() { fragColor = texture(uStrokeTex, vUv) * uOpacity; }
    """
}
```

app/src/main/java/com/wetinknext/engine/input/InputAction.kt

```
package com.wetinknext.engine.input

enum class InputAction {
    DOWN,
    MOVE,
    UP,
    CANCEL,
}
```

app/src/main/java/com/wetinknext/engine/input/InputBatch.kt

```
package com.wetinknext.engine.input

/** Fixed-capacity container for passing input from the UI thread to the GL thread. */
class InputBatch(val maxSamples: Int) {
    val samples = Array(maxSamples) { InputSample() }

    var action: InputAction = InputAction.MOVE
        private set
    var sampleCount: Int = 0
        private set
    var prediction: Boolean = false
        private set

    fun begin(action: InputAction, prediction: Boolean = false) {
        this.action = action
        this.prediction = prediction
        sampleCount = 0
    }

    fun addSample(
        canvasX: Float, canvasY: Float, pressure: Float, tiltX: Float, tiltY: Float,
        orientationRad: Float, timestampNanos: Long, pointerId: Int, tool: PointerTool,
        historical: Boolean,
    ): Boolean {
        if (sampleCount >= maxSamples) return false
        samples[sampleCount].set(
            canvasX, canvasY, pressure.coerceIn(0f, 1f), tiltX.coerceIn(-1f, 1f),
            tiltY.coerceIn(-1f, 1f), orientationRad, timestampNanos, pointerId, tool, historical,
        )
        sampleCount += 1
        return true
    }

    fun isEmpty(): Boolean = sampleCount == 0

    fun clear() {
        for (index in 0 until sampleCount) samples[index].clear()
        sampleCount = 0
        prediction = false
        action = InputAction.MOVE
    }
}
```

app/src/main/java/com/wetinknext/engine/input/InputBatchPool.kt

```
package com.wetinknext.engine.input

import java.util.concurrent.atomic.AtomicInteger
import java.util.concurrent.atomic.AtomicReferenceArray

/** Fixed-size pool that returns null instead of allocating when exhausted. */
class InputBatchPool(
    batchCount: Int = DEFAULT_BATCH_COUNT,
    maxSamplesPerBatch: Int = DEFAULT_MAX_SAMPLES,
) {
    private val batches = AtomicReferenceArray<InputBatch>(batchCount)
    private val available = AtomicInteger(batchCount)

    init {
        require(batchCount > 0)
        require(maxSamplesPerBatch > 0)
        for (index in 0 until batchCount) batches.set(index, InputBatch(maxSamplesPerBatch))
    }

    fun acquire(): InputBatch? {
        while (true) {
            val current = available.get()
            if (current <= 0) return null
            if (available.compareAndSet(current, current - 1)) {
                for (index in 0 until batches.length()) {
                    val batch = batches.getAndSet(index, null)
                    if (batch != null) return batch
                }
                available.incrementAndGet()
                return null
            }
        }
    }

    fun release(batch: InputBatch) {
        batch.clear()
        for (index in 0 until batches.length()) {
            if (batches.compareAndSet(index, null, batch)) {
                available.incrementAndGet()
                return
            }
        }
        error("InputBatchPool overflow: released batch does not belong to pool")
    }

    val freeCount: Int get() = available.get()

    companion object {
        const val DEFAULT_BATCH_COUNT = 96
        const val DEFAULT_MAX_SAMPLES = 64
    }
}
```

app/src/main/java/com/wetinknext/engine/input/InputSample.kt

```
package com.wetinknext.engine.input

/** Mutable input sample allocated only while a batch pool is initialized. */
class InputSample {
    var canvasX = 0f
    var canvasY = 0f
    var pressure = 0f
    var tiltX = 0f
    var tiltY = 0f
    var orientationRad = 0f
    var timestampNanos = 0L
    var pointerId = -1
    var tool = PointerTool.UNKNOWN
    var historical = false

    fun set(
        canvasX: Float, canvasY: Float, pressure: Float, tiltX: Float, tiltY: Float,
        orientationRad: Float, timestampNanos: Long, pointerId: Int, tool: PointerTool,
        historical: Boolean,
    ) {
        this.canvasX = canvasX; this.canvasY = canvasY; this.pressure = pressure
        this.tiltX = tiltX; this.tiltY = tiltY; this.orientationRad = orientationRad
        this.timestampNanos = timestampNanos; this.pointerId = pointerId; this.tool = tool
        this.historical = historical
    }

    fun clear() {
        canvasX = 0f; canvasY = 0f; pressure = 0f; tiltX = 0f; tiltY = 0f
        orientationRad = 0f; timestampNanos = 0L; pointerId = -1
        tool = PointerTool.UNKNOWN; historical = false
    }
}
```

app/src/main/java/com/wetinknext/engine/input/PointerTool.kt

```
package com.wetinknext.engine.input

enum class PointerTool {
    FINGER,
    STYLUS,
    ERASER,
    UNKNOWN,
}
```

app/src/main/java/com/wetinknext/engine/input/StrokeInputCapturer.kt

```
package com.wetinknext.engine.input

import android.view.MotionEvent
import com.wetinknext.engine.core.Camera
import java.util.concurrent.ArrayBlockingQueue

/** Converts View-local MotionEvent samples to canvas coordinates on the UI thread. */
class StrokeInputCapturer(private val camera: Camera, private val pool: InputBatchPool, private val queue:
ArrayBlockingQueue<InputBatch>) {
    private var activePointerId = -1
    private val canvasPoint = FloatArray(2)

    /** [redacted]: [redacted] => [redacted] [redacted]. */
    var droppedBatches = 0L
        private set

    fun onTouchEvent(event: MotionEvent): Boolean = when (event.actionMasked) {
        MotionEvent.ACTION_DOWN -> {
            activePointerId = event.getPointerId(0)
            enqueue(event, InputAction.DOWN, 0, false)
        }
        MotionEvent.ACTION_POINTER_DOWN -> {
            // [redacted]: [redacted] [redacted] / [redacted] [redacted] [redacted] [redacted].
            if (activePointerId >= 0) true else {
                val i = event.actionIndex
                activePointerId = event.getPointerId(i)
                enqueue(event, InputAction.DOWN, i, false)
            }
        }
        MotionEvent.ACTION_MOVE -> {
            val i = event.findPointerIndex(activePointerId)
            i >= 0 && enqueue(event, InputAction.MOVE, i, true)
        }
        MotionEvent.ACTION_UP, MotionEvent.ACTION_POINTER_UP -> {
            val i = if (event.actionMasked == MotionEvent.ACTION_UP) 0 else event.actionIndex
            if (event.getPointerId(i) != activePointerId) true else {
                activePointerId = -1
                // historical = true: [redacted] [redacted] [redacted] [redacted] [redacted] [redacted].
                enqueue(event, InputAction.UP, i, true)
            }
        }
        MotionEvent.ACTION_CANCEL -> { activePointerId = -1; enqueueEmpty(InputAction.CANCEL) }
        else -> false
    }

    private fun enqueue(event:MotionEvent, action:InputAction, index:Int, historical:Boolean):Boolean {
        val batch = pool.acquire() ?: run { droppedBatches++; return false }
        batch.begin(action);val t=camera.snapshot();val id=event.getPointerId(index);val tool=tool(event,index);if(
historical)for(h in 0 until event.historySize){t.screenToCanvas(event.getHistoricalX(index,h),event.getHistoricalY(
index,h),canvasPoint);if(!batch.addSample(canvasPoint[0],canvasPoint[1],pressure(event,tool,index,h),0f,0f,
orientation(event,tool,index,h),event.getHistoricalEventTime(h)*1_000_000L,id,tool,true))break};t.screenToCanvas(
event.getX(index),event.getY(index),canvasPoint);batch.addSample(canvasPoint[0],canvasPoint[1],pressure(event,tool,
index,-1),0f,0f,orientation(event,tool,index,-1),event.eventTime*1_000_000L,id,tool,false);if(!queue.offer(batch)){
pool.release(batch);droppedBatches++;return false};return true }

    private fun enqueueEmpty(action:InputAction):Boolean {
        val batch = pool.acquire() ?: run { droppedBatches++; return false }
        batch.begin(action);if(!queue.offer(batch)){pool.release(batch);droppedBatches++;return false};return true }
    private fun tool(e:MotionEvent,i:Int)=when(e.getToolType(i)){MotionEvent.TOOL_TYPE_STYLUS->PointerTool.STYLUS;
MotionEvent.TOOL_TYPE_ERASER->PointerTool.ERASER;MotionEvent.TOOL_TYPE_FINGER->PointerTool.FINGER;else->PointerTool.
UNKNOWN}
    private fun pressure(e:MotionEvent,t:PointerTool,i:Int,h:Int):Float=if(t==PointerTool.FINGER||t==PointerTool.
UNKNOWN).65f else (if(h>=0)e.getHistoricalPressure(i,h) else e.getPressure(i)).coerceIn(0f,1f)
    private fun orientation(e:MotionEvent,t:PointerTool,i:Int,h:Int):Float=if(t==PointerTool.FINGER||t==PointerTool.
UNKNOWN)0f else if(h>=0)e.getHistoricalAxisValue(MotionEvent.AXIS_ORIENTATION,i,h) else e.getAxisValue(MotionEvent.
AXIS_ORIENTATION,i)
}
```

app/src/main/java/com/wetinknext/engine/undo/TileCompressor.kt

```
package com.wetinknext.engine.undo

/**
 * Extension point for P6.5 compression. P6 uses the identity implementation,
 * so the 96 MB budget reflects the exact bytes that are currently retained.
 */
interface TileCompressor {
    fun compress(data: ByteArray): ByteArray

    fun decompress(data: ByteArray, expectedSize: Int): ByteArray
}

/** P6 storage policy: no compression and no extra byte-array allocation. */
object IdentityTileCompressor : TileCompressor {
    override fun compress(data: ByteArray): ByteArray = data

    override fun decompress(data: ByteArray, expectedSize: Int): ByteArray {
        check(data.size == expectedSize) {
            "Raw tile has ${data.size} bytes, expected $expectedSize"
        }
        return data
    }
}
```

app/src/main/java/com/wetinknext/engine/undo/TileSnapshot.kt

```
package com.wetinknext.engine.undo

data class TileCoord(val tx: Int, val ty: Int)

/** One full 256 px tile (or an edge tile) in the source layer's native format. */
class TileSnapshot(
    val coord: TileCoord,
    val pixelLeft: Int,
    val pixelTop: Int,
    val pixelWidth: Int,
    val pixelHeight: Int,
    val bytesPerPixel: Int,
    private val storedData: ByteArray,
    private val compressor: TileCompressor = IdentityTileCompressor,
) {
    val memorySize: Int get() = storedData.size
    val rawSize: Int get() = pixelWidth * pixelHeight * bytesPerPixel

    /** Returns raw bytes. Under P6 identity storage this returns the retained array. */
    fun decompress(): ByteArray = compressor.decompress(storedData, rawSize)

    /** Avoids a redundant decompression/copy in the P6 identity restore path. */
    fun storedBytes(): ByteArray = storedData

    fun dispose() {
        // P6 keeps heap ByteArrays; a later off-heap implementation can release here.
    }

    companion object {
        const val TILE_SIZE = 256
        const val BYTES_PER_PIXEL_RGBA16F = 8
        const val BYTES_PER_PIXEL_RGBA8 = 4
    }
}
```

app/src/main/java/com/wetinknext/engine/undo/TileSnapshotCapture.kt

```
package com.wetinknext.engine.undo
import android.opengl.GLES30
import com.wetinknext.engine.gl.RenderTarget
import java.nio.ByteBuffer
import java.nio.ByteOrder
object TileSnapshotCapture {
    private var reusable: ByteBuffer? = null

    /** Captures every 256 px tile intersecting [bounds] = left, top, right, bottom. */
    fun capture(target: RenderTarget, bounds: IntArray): List<TileSnapshot> {
        require(bounds.size >= 4)
        val left = bounds[0].coerceIn(0, target.width)
        val top = bounds[1].coerceIn(0, target.height)
        val right = bounds[2].coerceIn(0, target.width)
        val bottom = bounds[3].coerceIn(0, target.height)
        if (right <= left || bottom <= top) return emptyList()

        val bytesPerPixel = if (target.usesHalfFloat) {
            TileSnapshot.BYTES_PER_PIXEL_RGBA16F
        } else {
            TileSnapshot.BYTES_PER_PIXEL_RGBA8
        }
        val glType = if (target.usesHalfFloat) GLES30.GL_HALF_FLOAT else GLES30.GL_UNSIGNED_BYTE
        val result = ArrayList<TileSnapshot>()
        target.bind()
        GLES30.glPixelStorei(GLES30.GL_PACK_ALIGNMENT, 1)
        for (tileY in top / TileSnapshot.TILE_SIZE..(bottom - 1) / TileSnapshot.TILE_SIZE) {
            for (tileX in left / TileSnapshot.TILE_SIZE..(right - 1) / TileSnapshot.TILE_SIZE) {
                val pixelLeft = tileX * TileSnapshot.TILE_SIZE
                val pixelTop = tileY * TileSnapshot.TILE_SIZE
                val pixelWidth = minOf(TileSnapshot.TILE_SIZE, target.width - pixelLeft)
                val pixelHeight = minOf(TileSnapshot.TILE_SIZE, target.height - pixelTop)
                val byteCount = pixelWidth * pixelHeight * bytesPerPixel
                val buffer = obtainBuffer(byteCount)
                buffer.clear()
                GLES30.glReadPixels(
                    pixelLeft, pixelTop, pixelWidth, pixelHeight,
                    GLES30.GL_RGBA, glType, buffer,
                )
                buffer.rewind()
                val rawBytes = ByteArray(byteCount)
                buffer.get(rawBytes)
                result += TileSnapshot(
                    coord = TileCoord(tileX, tileY),
                    pixelLeft = pixelLeft,
                    pixelTop = pixelTop,
                    pixelWidth = pixelWidth,
                    pixelHeight = pixelHeight,
                    bytesPerPixel = bytesPerPixel,
                    storedData = IdentityTileCompressor.compress(rawBytes),
                    compressor = IdentityTileCompressor,
                )
            }
        }
        GLES30.glPixelStorei(GLES30.GL_PACK_ALIGNMENT, 4)
        GLES30.glBindFramebuffer(GLES30.GL_FRAMEBUFFER, 0)
        return result
    }

    private fun obtainBuffer(size: Int): ByteBuffer {
        val current = reusable
        if (current != null && current.capacity() >= size) return current
        return ByteBuffer.allocateDirect(size).order(ByteOrder.nativeOrder()).also { reusable = it }
    }
}
```

app/src/main/java/com/wetinknext/engine/undo/TileSnapshotRestore.kt

```
package com.wetinknext.engine.undo

import android.opengl.GLES30
import com.wetinknext.engine.gl.RenderTarget
import java.nio.ByteBuffer
import java.nio.ByteOrder

/** Restores snapshot tiles into an already-created layer texture on the GL thread. */
object TileSnapshotRestore {
    private var reusableBuffer: ByteBuffer? = null

    fun restore(target: RenderTarget, tiles: List<TileSnapshot>) {
        if (tiles.isEmpty()) return
        GLES30.glBindTexture(GLES30.GL_TEXTURE_2D, target.textureId)
        GLES30.glPixelStorei(GLES30.GL_UNPACK_ALIGNMENT, 1)
        for (tile in tiles) {
            val data = tile.storedBytes()
            val buffer = obtainBuffer(data.size)
            buffer.clear()
            buffer.put(data)
            buffer.position(0)
            val glType = if (tile.bytesPerPixel == TileSnapshot.BYTES_PER_PIXEL_RGBA16F) {
                GLES30.GL_HALF_FLOAT
            } else {
                GLES30.GL_UNSIGNED_BYTE
            }
            GLES30.glTexSubImage2D(
                GLES30.GL_TEXTURE_2D,
                0,
                tile.pixelLeft,
                tile.pixelTop,
                tile.pixelWidth,
                tile.pixelHeight,
                GLES30.GL_RGBA,
                glType,
                buffer,
            )
        }
        GLES30.glPixelStorei(GLES30.GL_UNPACK_ALIGNMENT, 4)
        GLES30.glBindTexture(GLES30.GL_TEXTURE_2D, 0)
    }

    private fun obtainBuffer(size: Int): ByteBuffer {
        val current = reusableBuffer
        if (current != null && current.capacity() >= size) return current
        return ByteBuffer.allocateDirect(size).order(ByteOrder.nativeOrder()).also { reusableBuffer = it }
    }
}
```

app/src/main/java/com/wetinknext/engine/undo/UndoEntry.kt

```
package com.wetinknext.engine.undo

/** One committed stroke, retaining exact states from before and after its merge. */
data class UndoEntry(
    val layerId: Long,
    val beforeTiles: List<TileSnapshot>,
    val afterTiles: List<TileSnapshot>,
    val tag: String = "stroke",
) {
    val memorySize: Int
        get() = beforeTiles.sumOf(TileSnapshot::memorySize) + afterTiles.sumOf(TileSnapshot::memorySize)

    fun dispose() {
        beforeTiles.forEach(TileSnapshot::dispose)
        afterTiles.forEach(TileSnapshot::dispose)
    }
}
```

app/src/main/java/com/wetinknext/engine/undo/UndoManager.kt

```
package com.wetinknext.engine.undo

/**
 * Pure memory-bounded history. GPU reads and texture restores deliberately live
 * in EngineRenderer, keeping this class deterministic and unit-testable.
 */
class UndoManager(
    private val maxMemoryBytes: Long = DEFAULT_MAX_MEMORY_BYTES,
    private val maxSteps: Int = DEFAULT_MAX_STEPS,
) {
    private val undoStack = ArrayDeque<UndoEntry>()
    private val redoStack = ArrayDeque<UndoEntry>()
    private var undoMemoryBytes = 0L
    private var redoMemoryBytes = 0L

    val canUndo: Boolean get() = undoStack.isNotEmpty()
    val canRedo: Boolean get() = redoStack.isNotEmpty()
    val memoryBytes: Long get() = undoMemoryBytes + redoMemoryBytes
    val undoCount: Int get() = undoStack.size
    val redoCount: Int get() = redoStack.size

    /** Records a new operation and invalidates the alternate redo history. */
    fun push(entry: UndoEntry) {
        undoStack.addLast(entry)
        undoMemoryBytes += entry.memorySize
        clearRedo()
        trimToLimits()
    }

    /** Moves the newest undo entry onto redo and returns it for a before-restore. */
    fun popUndo(): UndoEntry? {
        val entry = undoStack.removeLastOrNull() ?: return null
        undoMemoryBytes -= entry.memorySize
        redoStack.addLast(entry)
        redoMemoryBytes += entry.memorySize
        trimToLimits()
        return entry
    }

    /** Moves the newest redo entry back to undo and returns it for an after-restore. */
    fun popRedo(): UndoEntry? {
        val entry = redoStack.removeLastOrNull() ?: return null
        redoMemoryBytes -= entry.memorySize
        undoStack.addLast(entry)
        undoMemoryBytes += entry.memorySize
        trimToLimits()
        return entry
    }

    fun clearRedo() {
        while (redoStack.isNotEmpty()) {
            val entry = redoStack.removeLast()
            redoMemoryBytes -= entry.memorySize
            entry.dispose()
        }
        redoMemoryBytes = 0L
    }

    fun clear() {
        while (undoStack.isNotEmpty()) undoStack.removeLast().dispose()
        undoMemoryBytes = 0L
        clearRedo()
    }

    private fun trimToLimits() {
        while (undoStack.size > maxSteps) evictOldestUndo()
        while (redoStack.size > maxSteps) evictOldestRedo()
        while (memoryBytes > maxMemoryBytes && undoStack.size + redoStack.size > 1) {
            when {
                redoStack.size > 1 && redoMemoryBytes >= undoMemoryBytes -> evictOldestRedo()
                undoStack.size > 1 -> evictOldestUndo()
                redoStack.size > 1 -> evictOldestRedo()
                else -> return
            }
        }
    }

    private fun evictOldestUndo() {
        val entry = undoStack.removeFirstOrNull() ?: return
        undoMemoryBytes -= entry.memorySize
        entry.dispose()
    }

    private fun evictOldestRedo() {
        val entry = redoStack.removeFirstOrNull() ?: return
        redoMemoryBytes -= entry.memorySize
        entry.dispose()
```

```
    }

    companion object {
        const val DEFAULT_MAX_MEMORY_BYTES = 96L * 1024L * 1024L
        const val DEFAULT_MAX_STEPS = 50
    }
}
```

app/src/main/java/com/wetinknext/ui/theme/WetInkNextPalette.kt

```
package com.wetinknext.ui.theme

import androidx.compose.ui.graphics.Color

data class WetInkNextPalette(
    val appBg: Color,
    val panelBg: Color,
    val panelStroke: Color,
    val textPrimary: Color,
    val textSecondary: Color,
    val accent: Color,
) {
    companion object {
        val default = WetInkNextPalette(
            appBg = Color(0xFF17181C),
            panelBg = Color(0xFF25272D),
            panelStroke = Color(0xFF3A3D46),
            textPrimary = Color(0xFFF3F4F6),
            textSecondary = Color(0xFFA4A8B3),
            accent = Color(0xFF4D7AFF),
        )
    }
}
```

app/src/test/java/com/wetinknext/engine/brush/ColorSpacesTest.kt

```
package com.wetinknext.engine.brush
import org.junit.Assert.*
import org.junit.Test
class ColorSpacesTest { @Test fun blackAndWhiteAreStable(){val b=FloatArray(3);val w=FloatArray(3);ColorSpaces.
srgb8ToLinear(0xFF000000L,b);ColorSpaces.srgb8ToLinear(0xFFFFFFFFL,w);assertEquals(0f,b[0],.0001f);assertEquals(1f,w[
0],.0001f)} @Test fun roundTrip(){assertEquals(.25f,ColorSpaces.srgbToLinear(ColorSpaces.linearToSrgb(.25f)),.0001f)}
}
```

app/src/test/java/com/wetinknext/engine/brush/DabBufferTest.kt

```
package com.wetinknext.engine.brush

import org.junit.Assert.*
import org.junit.Test

class DabBufferTest {
    @Test fun overflowIsCounted(){val b=DabBuffer(2);assertTrue(b.add(0f,0f,1f,0f,1f));assertTrue(b.add(1f,1f,1f,0f,
1f));assertFalse(b.add(2f,2f,1f,0f,1f));assertEquals(1L,b.overflowCount)}
    @Test fun clearResets(){val b=DabBuffer(1);b.add(0f,0f,1f,0f,1f);b.clear();assertEquals(0,b.count)}
}
```

app/src/test/java/com/wetinknext/engine/brush/OneEuroFilterTest.kt

```
package com.wetinknext.engine.brush

import org.junit.Assert.assertEquals
import org.junit.Assert.assertFalse
import org.junit.Test

class OneEuroFilterTest {
    @Test fun firstSamplePassesThrough() { val f=OneEuroFilter();val out=FloatArray(2);f.filter(0,10f,20f,out);
assertEquals(10f,out[0],.0001f);assertEquals(20f,out[1],.0001f) }
    @Test fun samplesRemainFinite() { val f=OneEuroFilter();val out=FloatArray(2);f.filter(0,0f,0f,out);f.filter(
8_000_000,10f,5f,out);assertFalse(out[0].isNaN());assertFalse(out[1].isInfinite()) }
}
```

app/src/test/java/com/wetinknext/engine/brush/PressureFilterTest.kt

```
package com.wetinknext.engine.brush

import org.junit.Assert.assertTrue
import org.junit.Test

class PressureFilterTest {
    @Test fun singleZeroPressureGlitchDoesNotCollapseWidth() {
        val filter = PressureFilter()
        filter.filter(0L, .6f)
        val filtered = filter.filter(16_000_000L, 0f)
        assertTrue("zero glitch must be ignored while stroke is active", filtered > .55f)
    }

    @Test fun realPressureChangeIsSmoothed() {
        val filter = PressureFilter()
        filter.filter(0L, .8f)
        val filtered = filter.filter(16_000_000L, .2f)
        assertTrue(filtered in .2f.. .8f)
    }
}
```

app/src/test/java/com/wetinknext/engine/brush/RibbonClosureTest.kt

```
package com.wetinknext.engine.brush

import com.wetinknext.engine.input.InputAction
import com.wetinknext.engine.input.InputBatch
import com.wetinknext.engine.input.PointerTool
import kotlin.math.cos
import kotlin.math.sin
import org.junit.Assert.assertFalse
import org.junit.Assert.assertTrue
import org.junit.Test

class RibbonClosureTest {
    private fun sample(x: Float, y: Float, width: Float) = RibbonSample(x, y, width)

    @Test fun closedOutlineHasNoCapsAndHasWrapSegment() {
        val points = (0 until 24).map { i ->
            val angle = 2.0 * Math.PI * i / 24
            sample(60f + 40f * cos(angle).toFloat(), 60f + 40f * sin(angle).toFloat(), 4f)
        }
        val outline = RibbonGeometry.build(points, RibbonCap.ROUND, RibbonJoin.ROUND, 2.5f, 1f, closed = true)
        assertTrue(outline.startCap.isEmpty() && outline.endCap.isEmpty())
        val mesh = RibbonTriangulation.build(outline, 1f, join = RibbonJoin.ROUND, miterLimit = 2.5f)
        assertTrue("wrap segment missing", mesh.triangleCount >= 2 * points.size)
    }

    @Test fun openStrokeKeepsCapsAndHasNoWrap() {
        val points = listOf(sample(0f, 0f, 4f), sample(40f, 15f, 4f), sample(80f, 0f, 4f))
        val outline = RibbonGeometry.build(points, RibbonCap.ROUND, RibbonJoin.ROUND, 2.5f, 1f, closed = false)
        assertTrue(outline.startCap.isNotEmpty() && outline.endCap.isNotEmpty())
        assertFalse(outline.closed)
    }

    @Test fun emitterRecognizesCircleReturnedToStart() {
        val emitter = RibbonEmitter(BrushSettings(renderMode = BrushRenderMode.RIBBON, baseRadiusPx = 9f, smoothing =
0f, streamline = 0f, ribbon = RibbonSettings(minPointDistancePx = .5f)))
        val down = batch(InputAction.DOWN, 100f, 60f, 0L)
        emitter.begin(down)
        val move = InputBatch(32).apply {
            begin(InputAction.MOVE)
            for (i in 1..24) {
                val a = 2.0 * Math.PI * i / 24
                addSample(60f + 40f * cos(a).toFloat(), 60f + 40f * sin(a).toFloat(), .5f, 0f, 0f, 0f, i *
16_000_000L, 0, PointerTool.STYLUS, i > 1)
            }
        }
        emitter.append(move)
        emitter.append(batch(InputAction.UP, 100f, 60f, 25 * 16_000_000L))
        emitter.finish(cancel = false)
        assertTrue(emitter.closedLoop)
        assertTrue(emitter.lastClosureDistance <= emitter.lastClosureThreshold)
    }

    @Test fun openStrokeIsNotClosed() {
        val emitter = RibbonEmitter(BrushSettings(renderMode = BrushRenderMode.RIBBON, baseRadiusPx = 9f, smoothing =
0f, streamline = 0f))
        emitter.begin(batch(InputAction.DOWN, 0f, 0f, 0L))
        emitter.append(batch(InputAction.MOVE, 120f, 80f, 16_000_000L))
        emitter.append(batch(InputAction.UP, 120f, 80f, 32_000_000L))
        emitter.finish(cancel = false)
        assertFalse(emitter.closedLoop)
    }

    private fun batch(action: InputAction, x: Float, y: Float, timestamp: Long) = InputBatch(8).apply {
        begin(action)
        addSample(x, y, .5f, 0f, 0f, 0f, timestamp, 0, PointerTool.STYLUS, false)
    }
}
```

app/src/test/java/com/wetinknext/engine/brush/RibbonGeometryTest.kt

```
package com.wetinknext.engine.brush

import kotlin.math.hypot
import org.junit.Assert.assertEquals
import org.junit.Assert.assertTrue
import org.junit.Test

class RibbonGeometryTest {
    private fun sample(x: Float, y: Float, width: Float) = RibbonSample(x, y, width)
    @Test fun horizontalStrokeKeepsAllCenters() {
        val outline = RibbonGeometry.build(listOf(sample(0f,0f,5f), sample(100f,0f,5f), sample(200f,0f,5f)),
RibbonCap.BUTT, RibbonJoin.ROUND, 3f, 1f)
        assertEquals(3, outline.centers.size); assertEquals(5f, outline.widths[0], .01f); assertTrue(outline.startCap.
isEmpty())
    }
    @Test fun tapBuildsCircleCap() {
        val outline = RibbonGeometry.build(listOf(sample(10f,20f,7f)), RibbonCap.ROUND, RibbonJoin.ROUND, 3f)
        assertTrue(outline.startCap.size >= 8); outline.startCap.forEach { assertEquals(7f, hypot(it.x-10f,it.y-20f),.
01f) }
    }
    @Test fun shortOpenStrokeBuildsOneRoundDab() {
        val outline = RibbonGeometry.build(
            listOf(sample(10f, 20f, 12f), sample(14f, 20f, 12f)),
            RibbonCap.ROUND,
            RibbonJoin.ROUND,
            3f,
        )

        assertEquals(1, outline.centers.size)
        assertEquals(12f, outline.widths[0], .01f)
        assertTrue(outline.startCap.size >= 8)
        assertTrue(outline.endCap.isEmpty())

        val mesh = RibbonTriangulation.build(outline, aaWidthPx = 1f)
        assertTrue(mesh.triangleCount > 0)
    }
    @Test fun closedLoopIsNotReducedToDab() {
        val outline = RibbonGeometry.build(
            listOf(sample(0f, 0f, 12f), sample(2f, 0f, 12f), sample(1f, 2f, 12f)),
            RibbonCap.ROUND,
            RibbonJoin.ROUND,
            3f,
            closed = true,
        )

        assertTrue(outline.closed)
        assertEquals(3, outline.centers.size)
    }
}
```

app/src/test/java/com/wetinknext/engine/brush/RibbonShortStrokeTest.kt

```
package com.wetinknext.engine.brush

import com.wetinknext.engine.input.InputAction
import com.wetinknext.engine.input.InputBatch
import com.wetinknext.engine.input.PointerTool
import org.junit.Assert.assertEquals
import org.junit.Assert.assertFalse
import org.junit.Assert.assertTrue
import org.junit.Test

class RibbonShortStrokeTest {

    @Test fun bigBrushShortStrokeIsNotClosed() {
        val e = RibbonEmitter(
            BrushSettings(
                renderMode = BrushRenderMode.RIBBON, baseRadiusPx = 132f,
                smoothing = 0f, streamline = 0f,
                ribbon = RibbonSettings(minPointDistancePx = 1.25f),
            ),
        )
        e.begin(batch(InputAction.DOWN, 500f, 500f, 0L))
        e.append(batch(InputAction.MOVE, 560f, 520f, 16_000_000L))
        e.append(batch(InputAction.MOVE, 620f, 505f, 32_000_000L))
        e.append(batch(InputAction.UP, 630f, 500f, 48_000_000L))
        e.finish(cancel = false)
        assertFalse("[redacted]", e.closedLoop)
    }

    @Test fun tapWithFewSamplesBecomesDisk() {
        val outline = RibbonGeometry.build(
            listOf(RibbonSample(10f, 10f, 60f), RibbonSample(12f, 11f, 60f), RibbonSample(13f, 10f, 60f)),
            RibbonCap.ROUND, RibbonJoin.ROUND, 3f, 1f,
        )
        assertEquals(1, outline.centers.size)
        assertTrue(outline.startCap.size >= 8)
    }

    @Test fun endCapPointsForwardNotIntoTheStroke() {
        val outline = RibbonGeometry.build(
            listOf(RibbonSample(0f, 0f, 5f), RibbonSample(100f, 0f, 5f)),
            RibbonCap.ROUND, RibbonJoin.ROUND, 3f, 1f,
        )
        assertTrue("end cap [redacted]", outline.endCap.any { it.x > 100.5f })
        assertTrue("start cap [redacted]", outline.startCap.any { it.x < -0.5f })
    }

    private fun batch(action: InputAction, x: Float, y: Float, t: Long) = InputBatch(8).apply {
        begin(action)
        addSample(x, y, .5f, 0f, 0f, 0f, t, 0, PointerTool.STYLUS, false)
    }
}
```

app/src/test/java/com/wetinknext/engine/brush/RibbonTriangulationTest.kt

```
package com.wetinknext.engine.brush

import kotlin.math.cos
import kotlin.math.hypot
import org.junit.Assert.assertEquals
import org.junit.Assert.assertTrue
import org.junit.Test

class RibbonTriangulationTest {
    private fun sample(x: Float, y: Float, width: Float) = RibbonSample(x,y,width)
    @Test fun stripContainsTriangles() { val m=RibbonTriangulation.build(RibbonGeometry.build(listOf(sample(0f,0f,5f),
sample(100f,0f,5f)),RibbonCap.BUTT,RibbonJoin.ROUND,3f));assertTrue(m.triangleCount>=2);assertEquals(0,m.indices.
size%3) }
    @Test fun aaAddsZeroCoverageVertices() { val m=RibbonTriangulation.build(RibbonGeometry.build(listOf(sample(0f,0f,
5f),sample(100f,0f,5f)),RibbonCap.BUTT,RibbonJoin.ROUND,3f),1f);assertTrue(m.coverage.any{it==0f}) }
    @Test fun emptyOutlineIsEmptyMesh(){assertTrue(RibbonTriangulation.build(RibbonOutline(emptyList(),FloatArray(0),
emptyList(),emptyList())).isEmpty)}
    @Test fun arcHasNoCenterlineChordsOnOuterBoundary() {
        val samples=(0..12).map { i -> val a=Math.toRadians(i*7.5); RibbonSample((200*cos(a)).toFloat(),(200*kotlin.
math.sin(a)).toFloat(),10f) }
        val mesh=RibbonTriangulation.build(RibbonGeometry.build(samples,RibbonCap.ROUND,RibbonJoin.ROUND,3f,1f),1f,
join=RibbonJoin.ROUND,miterLimit=3f)
        var outer=0; for(v in 0 until mesh.vertexCount){if(mesh.coverage[v]>=.999f){val d=hypot(mesh.vertices[v*2],
mesh.vertices[v*2+1]);if(d>150f){outer++;assertTrue(d in 185f..215f)}}};assertTrue(outer>0);assertEquals(0,mesh.
indices.size%3)
    }
}
```

app/src/test/java/com/wetinknext/engine/brush/StabilizerTest.kt

```
package com.wetinknext.engine.brush

import org.junit.Assert.assertEquals
import org.junit.Assert.assertTrue
import org.junit.Test

class StabilizerTest {
    @Test fun strengthZeroPassesRawThrough(){val s=Stabilizer();s.process(0,100f,200f);assertEquals(100f,s.x,.0001f);
assertEquals(200f,s.y,.0001f)}
    @Test fun velocityIsPositiveAfterMovement(){val s=Stabilizer();s.process(0,0f,0f);s.process(1_000_000_000,100f,
0f);assertTrue(s.velocity>0f)}
}
```

app/src/test/java/com/wetinknext/engine/brush/StampEmitterTest.kt

```
package com.wetinknext.engine.brush

import com.wetinknext.engine.input.*
import org.junit.Assert.assertEquals
import org.junit.Test

class StampEmitterTest {
    @Test fun constantSpacingWithoutEndpointDuplicate(){val e=StampEmitter(BrushSettings(baseRadiusPx=10f,spacing=10f,
spacingUsesDiameter=false,pressureToSize=false,pressureToOpacity=false));val d=DabBuffer(16);val down=InputBatch(1).
apply{begin(InputAction.DOWN);addSample(0f,0f,1f,0f,0f,0f,0,0,PointerTool.STYLUS,false)};val move=InputBatch(1).apply{
begin(InputAction.MOVE);addSample(20f,0f,1f,0f,0f,0f,1,0,PointerTool.STYLUS,false)};e.begin(down,d);e.append(move,d);
e.finish(d,false);assertEquals(3,d.count)}
}
```

app/src/test/java/com/wetinknext/engine/core/DirtyRectTest.kt

```
package com.wetinknext.engine.core

import org.junit.Assert.assertArrayEquals
import org.junit.Assert.assertTrue
import org.junit.Test

class DirtyRectTest {
    @Test fun includesDabsAndConvertsToPixelBounds() {
        val rect = DirtyRect()
        rect.include(100f, 120f, 10f)
        rect.include(150f, 140f, 5f)
        val pixels = IntArray(4)
        rect.toPixelBounds(pixels)
        assertArrayEquals(intArrayOf(90, 110, 155, 145), pixels)
    }

    @Test fun clampCanMakeRectEmpty() {
        val rect = DirtyRect()
        rect.include(-100f, -100f, -10f)
        rect.clamp(100f, 100f)
        assertTrue(rect.isEmpty)
    }

    @Test fun clearResetsRect() {
        val rect = DirtyRect()
        rect.include(0f, 0f, 10f)
        rect.clear()
        assertTrue(rect.isEmpty)
    }
}
```

app/src/test/java/com/wetinknext/engine/core/ViewTransformFboTest.kt

```
package com.wetinknext.engine.core
import org.junit.Assert.assertEquals
import org.junit.Test
class ViewTransformFboTest { @Test fun orthoValues(){val m=FloatArray(16);ViewTransform.buildCanvasToFbo(100f,200f,m);
assertEquals(.02f,m[0],.0001f);assertEquals(.01f,m[5],.0001f);assertEquals(-1f,m[12],.0001f);assertEquals(-1f,m[13],.
0001f)} }
```

app/src/test/java/com/wetinknext/engine/core/ViewTransformTest.kt

```
package com.wetinknext.engine.core

import org.junit.Assert.assertEquals
import org.junit.Test

class ViewTransformTest {
    @Test fun canvasAndScreenCoordinatesAreInverse() {
        val transform = ViewTransform(120f, 80f, 2.5f, 0.35f, true)
        val screen = FloatArray(2)
        val canvas = FloatArray(2)
        transform.canvasToScreen(240f, 310f, screen)
        transform.screenToCanvas(screen[0], screen[1], canvas)
        assertEquals(240f, canvas[0], 0.001f)
        assertEquals(310f, canvas[1], 0.001f)
    }

    @Test fun zoomAroundAnchorKeepsAnchorStable() {
        val initial = ViewTransform(translateX = 20f, translateY = 30f)
        val before = FloatArray(2)
        val after = FloatArray(2)
        initial.screenToCanvas(400f, 300f, before)
        initial.zoomAround(400f, 300f, 3f).screenToCanvas(400f, 300f, after)
        assertEquals(before[0], after[0], 0.001f)
        assertEquals(before[1], after[1], 0.001f)
    }
}
```

app/src/test/java/com/wetinknext/engine/undo/TileSnapshotTest.kt

```
package com.wetinknext.engine.undo

import org.junit.Assert.assertArrayEquals
import org.junit.Assert.assertEquals
import org.junit.Test

class TileSnapshotTest {
    @Test
    fun rawIdentityTileKeepsItsExactBytes() {
        val bytes = byteArrayOf(0, 10, 20, 30, 40, 50, 60, 70)
        val tile = TileSnapshot(
            coord = TileCoord(2, 3),
            pixelLeft = 512,
            pixelTop = 768,
            pixelWidth = 1,
            pixelHeight = 2,
            bytesPerPixel = TileSnapshot.BYTES_PER_PIXEL_RGBA8,
            storedData = bytes,
        )

        assertEquals(bytes.size, tile.memorySize)
        assertEquals(bytes.size, tile.rawSize)
        assertArrayEquals(bytes, tile.decompress())
    }

    @Test
    fun tileSizeIsTheSpecified256Pixels() {
        assertEquals(256, TileSnapshot.TILE_SIZE)
    }
}
```

app/src/test/java/com/wetinknext/engine/undo/UndoManagerTest.kt

```
package com.wetinknext.engine.undo

import org.junit.Assert.assertEquals
import org.junit.Assert.assertFalse
import org.junit.Assert.assertSame
import org.junit.Assert.assertTrue
import org.junit.Test

class UndoManagerTest {
    @Test
    fun undoAndRedoMoveTheSameEntryBetweenStacks() {
        val manager = UndoManager()
        val entry = entry(1L)
        manager.push(entry)

        assertTrue(manager.canUndo)
        assertFalse(manager.canRedo)
        assertSame(entry, manager.popUndo())
        assertFalse(manager.canUndo)
        assertTrue(manager.canRedo)
        assertSame(entry, manager.popRedo())
        assertTrue(manager.canUndo)
        assertFalse(manager.canRedo)
    }

    @Test
    fun aNewEntryClearsRedoHistory() {
        val manager = UndoManager()
        val first = entry(1L)
        manager.push(first)
        manager.popUndo()
        assertTrue(manager.canRedo)

        manager.push(entry(1L))

        assertFalse(manager.canRedo)
        assertEquals(1, manager.undoCount)
        assertEquals(0, manager.redoCount)
    }

    @Test
    fun stepLimitEvictsTheOldestHistoryEntry() {
        val manager = UndoManager(maxSteps = 2)
        manager.push(entry(1L))
        manager.push(entry(2L))
        manager.push(entry(3L))

        assertEquals(2, manager.undoCount)
    }

    private fun entry(layerId: Long): UndoEntry = UndoEntry(
        layerId = layerId,
        beforeTiles = listOf(tile(0)),
        afterTiles = listOf(tile(1)),
    )

    private fun tile(x: Int): TileSnapshot = TileSnapshot(
        coord = TileCoord(x, 0),
        pixelLeft = x,
        pixelTop = 0,
        pixelWidth = 1,
        pixelHeight = 1,
        bytesPerPixel = TileSnapshot.BYTES_PER_PIXEL_RGBA8,
        storedData = byteArrayOf(1, 2, 3, 4),
    )
}
```