WetInk Next

Complete current source bundle - P0 through P8

Status update - 10 August 2026

The project includes the P0-P7 rendering foundation and the current P8 brush-engine hardening work. This update records the implemented BrushSettings validation, pressure/spacing/flow pipeline, STAMP trajectory handling, shape and grain texture support, incremental preview checks, expanded textured dirty regions, and transactional Undo/Redo history.

Verified in this update: :app:compileDebugKotlin and :app:testDebugUnitTest completed successfully. The debug APK was installed on the connected Android tablet.

Included Kotlin files: 98. Generated from E:\dev\WetInk-Next.

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

```
package com.wetinknext.app

import androidx.compose.foundation.background
import androidx.compose.foundation.layout.*
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.unit.dp
import androidx.compose.ui.viewinterop.AndroidView
import androidx.compose.ui.platform.LocalContext
import com.wetinknext.engine.brush.BrushPreset
import com.wetinknext.engine.brush.BrushLibrary
import com.wetinknext.engine.core.EditorUiState
import com.wetinknext.engine.core.PaintSurfaceView
import com.wetinknext.ui.animation.AnimationTimelineToolbar
import com.wetinknext.ui.color.ColorPanel
import com.wetinknext.ui.color.GlesColorState
import com.wetinknext.ui.components.*
import com.wetinknext.ui.state.LayerState
import com.wetinknext.ui.theme.WetInkTheme
import com.wetinknext.ui.theme.rememberThemeController

private enum class EditorPanel {
    NONE,
    BRUSH,
    LAYERS,
    COLOR,
    SELECTION,
    TRANSFORM,
    ADJUSTMENTS,
    ANIMATION
}

@Composable
fun EditorScreen(
    onBack: () -> Unit = {}
) {
    val context = LocalContext.current
    val themeController = rememberThemeController()
    val theme = themeController.current
    var uiState by remember { mutableStateOf(EditorUiState.empty) }
    var openPanel by remember { mutableStateOf(EditorPanel.NONE) }
    var isEraser by remember { mutableStateOf(false) }

    val colorState = remember { GlesColorState(context) }
    var brushColor by remember { mutableStateOf(Color.Black) }
    var selectedBrush by remember { mutableStateOf<BrushPreset?>(BrushLibrary.pencil6B) }

    val layerState = remember { LayerState() }

    val surface = remember {
        PaintSurfaceView(context).also { view ->
            view.onEditorStateChange = { uiState = it }
        }
    }

    LaunchedEffect(uiState) {
        layerState.syncFromEditor(uiState)
    }

    // Initialize state
    LaunchedEffect(surface) {
        surface.requestState()
        surface.applyBrushPreset(BrushLibrary.pencil6B)
        selectedBrush = BrushLibrary.pencil6B
    }

    WetInkTheme(theme = theme, fontMode = themeController.font) {
        Box(
            modifier = Modifier
                .fillMaxSize()
                .background(theme.appBg),
        ) {
            AndroidView(
                modifier = Modifier.fillMaxSize(),
                factory = { surface },
            )

            // Top Toolbar
            Box(
                modifier = Modifier
                    .align(Alignment.TopCenter)
                    .padding(top = 12.dp)
            ) {
                TopToolbar(
                    theme = theme,
                    currentColor = brushColor,
                    canUndo = uiState.canUndo,
```

```
                    canRedo = uiState.canRedo,
                    isSelectionActive = openPanel == EditorPanel.SELECTION,
                    isTransformActive = openPanel == EditorPanel.TRANSFORM,
                    isAnimationActive = openPanel == EditorPanel.ANIMATION,
                    isAdjustmentsActive = openPanel == EditorPanel.ADJUSTMENTS,
                    onUndoClick = { surface.undo() },
                    onRedoClick = { surface.redo() },
                    onSelectionClick = {
                        openPanel = if (openPanel == EditorPanel.SELECTION) EditorPanel.NONE else EditorPanel.SELECTION
                    },
                    onTransformClick = {
                        openPanel = if (openPanel == EditorPanel.TRANSFORM) EditorPanel.NONE else EditorPanel.TRANSFORM
                    },
                    onAnimationClick = {
                        openPanel = if (openPanel == EditorPanel.ANIMATION) EditorPanel.NONE else EditorPanel.ANIMATION
                    },
                    onAdjustmentsClick = {
                        openPanel = if (openPanel == EditorPanel.ADJUSTMENTS) EditorPanel.NONE else EditorPanel.ADJUSTME
    NTS
                    },
                    onLayersClick = {
                        openPanel = if (openPanel == EditorPanel.LAYERS) EditorPanel.NONE else EditorPanel.LAYERS
                    },
                    onColorClick = {
                        openPanel = if (openPanel == EditorPanel.COLOR) EditorPanel.NONE else EditorPanel.COLOR
                    },
                )
            }

            // Side Toolbar
            Box(
                modifier = Modifier
                    .align(Alignment.CenterStart)
                    .padding(start = 12.dp)
            ) {
                SideToolbar(
                    theme = theme,
                    brushSize = uiState.brushSizePx,
                    brushOpacity = uiState.brushOpacity,
                    isEraser = isEraser,
                    onBrushSizeChange = { surface.setBrushSize(it) },
                    onBrushOpacityChange = { surface.setBrushOpacity(it) },
                    onBrushClick = {
                        openPanel = if (openPanel == EditorPanel.BRUSH) EditorPanel.NONE else EditorPanel.BRUSH
                    },
                    onEraserClick = { isEraser = it }
                )
            }

            // Panel Host
            EditorPanelHost(
                panel = openPanel,
                theme = theme,
                surface = surface,
                layerState = layerState,
                colorState = colorState,
                brushColor = brushColor,
                selectedBrush = selectedBrush,
                onBrushSelected = { preset ->
                    selectedBrush = preset
                    surface.applyBrushPreset(preset)
                },
                onColorChange = {
                    brushColor = it
                    surface.setBrushColor(it)
                },
                onDismiss = { openPanel = EditorPanel.NONE }
            )
        }
    }
}

@Composable
private fun EditorPanelHost(
    panel: EditorPanel,
    theme: com.wetinknext.ui.theme.AppTheme,
    surface: PaintSurfaceView?,
    layerState: LayerState,
    colorState: GlesColorState,
    brushColor: Color,
    selectedBrush: BrushPreset?,
    onBrushSelected: (BrushPreset) -> Unit,
    onColorChange: (Color) -> Unit,
    onDismiss: () -> Unit
) {
    Box(modifier = Modifier.fillMaxSize()) {
        when (panel) {
            EditorPanel.NONE -> Unit

            EditorPanel.BRUSH -> {
                Box(modifier = Modifier.align(Alignment.CenterEnd).padding(end = 12.dp)) {
                    BrushPanel(
                        currentBrush = selectedBrush ?: BrushLibrary.pencil6B,
```

```
            onBrushSelect = onBrushSelected,
            onBrushStudioOpen = { /* TODO */ },
            theme = theme,
            onDismiss = onDismiss,
            onDuplicate = { /* TODO */ },
            onDelete = { /* TODO */ }
        )
    }
}

EditorPanel.LAYERS -> {
    Box(modifier = Modifier.align(Alignment.CenterEnd).padding(end = 12.dp)) {
        LayersPanel(
            state = layerState,
            theme = theme,
            onDismiss = onDismiss,
            onAddLayer = { surface?.addLayer() },
            onSelectLayer = { surface?.setActiveLayer(it) },
            onVisibleChange = { id, visible -> surface?.setLayerVisible(id, visible) },
            onOpacityChange = { id, opacity -> surface?.setLayerOpacity(id, opacity) },
            onRemoveLayer = { id -> surface?.removeLayer(id) }
        )
    }
}

EditorPanel.COLOR -> {
    Box(modifier = Modifier.align(Alignment.CenterEnd).padding(end = 12.dp)) {
        ColorPanel(
            state = colorState,
            color = brushColor,
            theme = theme,
            onColorChange = onColorChange,
            onAddFromPhoto = { /* TODO */ },
            onDismiss = onDismiss
        )
    }
}

EditorPanel.SELECTION -> {
    Box(modifier = Modifier.align(Alignment.BottomCenter)) {
        SelectionToolbar(
            theme = theme,
            currentShape = SelectionShapeUi.FREEHAND,
            onShapeChange = { /* TODO */ },
            onDone = onDismiss
        )
    }
}

EditorPanel.TRANSFORM -> {
    Box(modifier = Modifier.align(Alignment.BottomCenter).padding(bottom = 20.dp)) {
        TransformMenuView(
            theme = theme,
            currentMode = TransformModeUi.UNIFORM,
            actions = object : TransformActions {
                override fun reset() { /* TODO */ }
                override fun cancel() { onDismiss() }
                override fun apply() { onDismiss() }
                override fun flipHorizontal() { /* TODO */ }
                override fun flipVertical() { /* TODO */ }
                override fun setMode(mode: TransformModeUi) { /* TODO */ }
            }
        )
    }
}

EditorPanel.ADJUSTMENTS -> {
    Box(modifier = Modifier.align(Alignment.CenterEnd).padding(end = 12.dp)) {
        AdjustmentsPanel(
            theme = theme,
            onClose = onDismiss,
            onAction = { /* TODO */ }
        )
    }
}

EditorPanel.ANIMATION -> {
    Box(modifier = Modifier.align(Alignment.BottomCenter)) {
        AnimationTimelineToolbar(
            theme = theme,
            document = com.wetinknext.domain.animation.AnimationDocument(enabled = true),
            currentFrameId = 0L,
            onFrameSelect = {},
            onFrameMove = { _, _ -> },
            onAddFrame = {},
            onDuplicateFrame = {},
            onDeleteFrame = {},
            onHoldChange = { _, _ -> },
            onRoleToggle = { _, _ -> },
            onAssembleLayers = {},
            onLayersClick = {},
            onClose = onDismiss
        )
```

```
                }
            }
        }
    }
}

private fun PaintSurfaceView.applyBrushPreset(
    preset: BrushPreset
) {
    val settings = preset.settings

    applyBrush(settings)

    val grainPath = settings.grainAssetPath
        ?.trim()
        ?.takeIf { it.isNotEmpty() }

    if (grainPath == null) {
        clearGrainTexture()
    } else {
        loadGrainTexture(
            path = grainPath,
            scale = settings.grainScale,
            canvasLocked = settings.grainCanvasLocked,
            depth = settings.textureDepth,
            contrast = settings.textureContrast,
        )
    }

    val shapePath = settings.shapeAssetPath?.trim()?.takeIf { it.isNotEmpty() }
    if (shapePath == null) {
        clearShapeTexture()
    } else {
        loadShapeTexture(path = shapePath, rgbToAlpha = settings.rgbToAlpha)
    }
}
```

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/domain/animation/AnimationModels.kt

```
package com.wetinknext.domain.animation

import kotlinx.serialization.Serializable

@Serializable
enum class AnimationPlaybackMode {
    LOOP,
    PING_PONG,
    ONE_SHOT,
}

@Serializable
enum class AnimationFrameRole {
    NORMAL,
    BACKGROUND,
    FOREGROUND,
}

@Serializable
data class OnionSkinSettings(
    val enabled: Boolean = false, // Disabled by default until renderer supports it
    val previousFrames: Int = 1,
    val nextFrames: Int = 1,
    val opacity: Float = 0.35f,
    val previousColorArgb: Int = 0xFFFF4FA3.toInt(),
    val nextColorArgb: Int = 0xFF4A8DFF.toInt(),
    val blendPrimaryFrame: Boolean = false,
)

@Serializable
```

```
data class AnimationFrame(
    val id: Long,
    val layerIds: List<Long>,
    val holdFrames: Int = 1,
    val role: AnimationFrameRole = AnimationFrameRole.NORMAL,
)

@Serializable
data class AnimationDocument(
    val enabled: Boolean = false,
    val framesPerSecond: Int = 12,
    val playbackMode: AnimationPlaybackMode = AnimationPlaybackMode.LOOP,
    val onionSkin: OnionSkinSettings = OnionSkinSettings(),
    val frames: List<AnimationFrame> = emptyList(),
)
```

app/src/main/java/com/wetinknext/domain/animation/AnimationRenderPlan.kt

```
package com.wetinknext.domain.animation

/**
 * Defines which frames and roles should be rendered in the current view.
 */
data class AnimationRenderPlan(
    val backgroundFrame: AnimationFrame?,
    val currentFrame: AnimationFrame?,
    val foregroundFrame: AnimationFrame?,
    val previousOnionFrames: List<AnimationFrame>,
    val nextOnionFrames: List<AnimationFrame>,
)

/**
 * Logic to decide which layers are visible based on the animation document.
 * This is a pure function for easy testing.
 */
fun buildAnimationRenderPlan(
    document: AnimationDocument,
    currentFrameId: Long,
): AnimationRenderPlan {
    if (!document.enabled) {
        return AnimationRenderPlan(null, null, null, emptyList(), emptyList())
    }

    val frames = document.frames
    val currentIndex = frames.indexOfFirst { it.id == currentFrameId }

    val bgFrame = frames.find { it.role == AnimationFrameRole.BACKGROUND }
    val fgFrame = frames.find { it.role == AnimationFrameRole.FOREGROUND }
    val current = if (currentIndex >= 0) frames[currentIndex] else null

    val prevOnions = mutableListOf<AnimationFrame>()
    val nextOnions = mutableListOf<AnimationFrame>()

    if (document.onionSkin.enabled && currentIndex >= 0) {
        // Previous frames
        for (i in 1..document.onionSkin.previousFrames) {
            val idx = currentIndex - i
            if (idx >= 0) {
                val f = frames[idx]
                if (f.role == AnimationFrameRole.NORMAL) prevOnions.add(f)
            }
        }
        // Next frames
        for (i in 1..document.onionSkin.nextFrames) {
            val idx = currentIndex + i
            if (idx < frames.size) {
                val f = frames[idx]
                if (f.role == AnimationFrameRole.NORMAL) nextOnions.add(f)
            }
        }
    }

    return AnimationRenderPlan(
        backgroundFrame = bgFrame,
        currentFrame = current,
        foregroundFrame = fgFrame,
        previousOnionFrames = prevOnions,
        nextOnionFrames = nextOnions
    )
}
```

app/src/main/java/com/wetinknext/domain/animation/AnimationTimeline.kt

```
package com.wetinknext.domain.animation

/**
 * Creates a default set of animation frames from a list of layer IDs.
 * Each layer becomes its own frame. System "Background" (usually ID 1) should be filtered out by the caller.
 */
fun createFramesFromLayers(layerIds: List<Long>): List<AnimationFrame> {
    return layerIds.map { layerId ->
        AnimationFrame(
            id = layerId, // Use layerId as initial frame ID for simplicity in 1:1 mapping
            layerIds = listOf(layerId)
        )
    }
}

/**
 * Ensures the animation document is valid given the currently existing layers.
 * Removes dead layer references, empty frames, and enforces range limits.
 */
fun normalizedAnimationDocument(
    document: AnimationDocument,
    existingLayerIds: Set<Long>,
): AnimationDocument {
    return normalizedAnimationDocument(
        document = document,
        layersInDocumentOrder = existingLayerIds.map { AnimationLayerInfo(id = it) },
    )
}

/** Minimal layer data needed by the timeline. Kept independent from OpenGL resources. */
data class AnimationLayerInfo(
    val id: Long,
    val isCanvasBackgroundLayer: Boolean = false,
    val isClipping: Boolean = false,
)

/**
 * Normalizes frames against the real document order. Canvas background never becomes a frame.
 * Layer IDs inside every frame are also kept in document order, which is required for clipping.
 */
fun normalizedAnimationDocument(
    document: AnimationDocument,
    layersInDocumentOrder: List<AnimationLayerInfo>,
): AnimationDocument {
    val order = layersInDocumentOrder.mapIndexed { index, layer -> layer.id to index }.toMap()
    val animatableIds = layersInDocumentOrder
        .filterNot { it.isCanvasBackgroundLayer }
        .map { it.id }
    val usedLayerIds = mutableSetOf<Long>()

    val normalizedFrames = document.frames.mapNotNull { frame ->
        val validLayerIds = frame.layerIds
            .filter { id -> id in order && id !in usedLayerIds }
            .sortedBy { order.getOrDefault(it, 0) }

        if (validLayerIds.isEmpty()) {
            null
        } else {
            usedLayerIds.addAll(validLayerIds)
            frame.copy(
                layerIds = validLayerIds,
                holdFrames = frame.holdFrames.coerceIn(1, 120)
            )
        }
    }
    // A layer can belong to only one frame. Any newly created document layer gets its own frame.
    val missingLayerIds = animatableIds.filter { it !in usedLayerIds }
    val newFrames = missingLayerIds.map { layerId ->
        AnimationFrame(id = layerId, layerIds = listOf(layerId))
    }
    return document.copy(
        framesPerSecond = document.framesPerSecond.coerceIn(1, 60),
        frames = normalizedFrames + newFrames
    )
}

/**
 * Combines multiple layers into a single animation frame.
 */
fun groupLayersIntoFrame(
    document: AnimationDocument,
    layerIdsToGroup: List<Long>,
    newFrameId: Long = System.nanoTime()
): AnimationDocument {
    val orderedIds = layerIdsToGroup.distinct()
    val idsSet = orderedIds.toSet()
    if (idsSet.isEmpty()) return document

    val insertionIndex = document.frames.indexOfFirst { frame -> frame.layerIds.any { it in idsSet } }
    val cleanedFrames = document.frames.map { frame ->
        frame.copy(layerIds = frame.layerIds.filter { it !in idsSet })
    }.filter { it.layerIds.isNotEmpty() }.toMutableList()
    val newFrame = AnimationFrame(id = newFrameId, layerIds = orderedIds)
```

```
    val targetIndex = if (insertionIndex < 0) cleanedFrames.size else insertionIndex.coerceAtMost(cleanedFrames.size)
    cleanedFrames.add(targetIndex, newFrame)
    return document.copy(frames = cleanedFrames)
}

sealed interface AnimationFrameGroupingValidation {
    data class Valid(val layerIdsInDocumentOrder: List<Long>) : AnimationFrameGroupingValidation
    data object EmptySelection : AnimationFrameGroupingValidation
    data class MissingClippingBase(val clippingLayerId: Long) : AnimationFrameGroupingValidation
    data class MissingClippingLayer(val baseLayerId: Long, val clippingLayerId: Long) : AnimationFrameGroupingValidation
}

/**
 * A clipping layer and its base must live in one animation frame: otherwise a frame would render
 * with a different compositing result than the document. The selection is returned in document order.
 */
fun validateAnimationFrameGrouping(
    selectedLayerIds: Set<Long>,
    layersInDocumentOrder: List<AnimationLayerInfo>,
): AnimationFrameGroupingValidation {
    val layers = layersInDocumentOrder.filterNot { it.isCanvasBackgroundLayer }
    val selected = selectedLayerIds.intersect(layers.map { it.id }.toSet())
    if (selected.isEmpty()) return AnimationFrameGroupingValidation.EmptySelection

    val baseByClippingId = mutableMapOf<Long, Long>()
    layers.forEachIndexed { index, layer ->
        if (layer.isClipping) {
            val base = layers.subList(0, index).lastOrNull { !it.isClipping }
                ?: return AnimationFrameGroupingValidation.MissingClippingBase(layer.id)
            baseByClippingId[layer.id] = base.id
            if (layer.id in selected && base.id !in selected) {
                return AnimationFrameGroupingValidation.MissingClippingBase(layer.id)
            }
        }
    }
    baseByClippingId.forEach { (clippingId, baseId) ->
        if (baseId in selected && clippingId !in selected) {
            return AnimationFrameGroupingValidation.MissingClippingLayer(baseId, clippingId)
        }
    }
    return AnimationFrameGroupingValidation.Valid(layers.filter { it.id in selected }.map { it.id })
}

/**
 * Expands frames into a flat list of frame IDs to be played back, accounting for holdFrames.
 */
fun expandedPlaybackFrameIds(
    frames: List<AnimationFrame>,
): List<Long> {
    return frames.flatMap { frame ->
        List(frame.holdFrames) { frame.id }
    }
}

data class PlaybackStep(
    val nextIndex: Int,
    val nextDirection: Int, // 1 for forward, -1 for backward
    val shouldStop: Boolean = false
)

/**
 * Calculates the next index in the playback sequence.
 */
fun nextPlaybackIndex(
    currentIndex: Int,
    direction: Int,
    frameCount: Int,
    mode: AnimationPlaybackMode,
): PlaybackStep {
    if (frameCount <= 0) return PlaybackStep(0, 1, true)
    if (frameCount == 1) return PlaybackStep(0, 1, mode == AnimationPlaybackMode.ONE_SHOT)

    var nextDir = direction
    var nextIdx = currentIndex + direction
    var stop = false

    when (mode) {
        AnimationPlaybackMode.LOOP -> {
            if (nextIdx >= frameCount) nextIdx = 0
            if (nextIdx < 0) nextIdx = frameCount - 1
        }
        AnimationPlaybackMode.PING_PONG -> {
            if (nextIdx >= frameCount) {
                nextIdx = (frameCount - 2).coerceAtLeast(0)
                nextDir = -1
            } else if (nextIdx < 0) {
                nextIdx = (1).coerceAtMost(frameCount - 1)
                nextDir = 1
            }
        }
        AnimationPlaybackMode.ONE_SHOT -> {
            if (nextIdx >= frameCount) {
                nextIdx = frameCount - 1
```

```
                stop = true
            }
        }
    }

    return PlaybackStep(nextIdx, nextDir, stop)
}
```

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

```
package com.wetinknext.engine.brush

import android.opengl.GLES30

class BlendController {

    fun begin(policy: BlendPolicy) {
        GLES30.glEnable(GLES30.GL_BLEND)

        when (policy) {
            BlendPolicy.NORMAL_BUILDUP -> {
                GLES30.glBlendEquation(GLES30.GL_FUNC_ADD)
                GLES30.glBlendFunc(
                    GLES30.GL_ONE,
                    GLES30.GL_ONE_MINUS_SRC_ALPHA,
                )
            }

            BlendPolicy.NON_BUILDUP -> {
                /*
                 * ■■■■■■■■■■ ■■■■■■■■■■■ ■■■ ■■■■■■■■■ P7.
                 * ■■ ■■■■■■■■ GL_MAX ■■■■■■■■■■■■■■■ ■■■■■■■■ ■■■■■ non-buildup ■■■■■■■.
                 * ■■■■ ■■■■■■■■■■ ■■■■■■■ ■■■■■■ ■■■■■■■■■■■■■ ■■■■■■■■■■■ shader/pass.
                 */
                GLES30.glBlendEquation(GLES30.GL_MAX)
                GLES30.glBlendFunc(
                    GLES30.GL_ONE,
                    GLES30.GL_ONE,
                )
            }
        }
    }

    fun end() {
        GLES30.glBlendEquation(GLES30.GL_FUNC_ADD)
        GLES30.glBlendFunc(
            GLES30.GL_ONE,
            GLES30.GL_ONE_MINUS_SRC_ALPHA,
        )
        GLES30.glDisable(GLES30.GL_BLEND)
    }
}
```

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

```
package com.wetinknext.engine.brush

import kotlin.math.pow
import kotlin.math.sqrt

object BrushDynamics {

    fun tiltMagnitude(
        tiltX: Float,
        tiltY: Float,
    ): Float {
        return sqrt(
            tiltX * tiltX +
                tiltY * tiltY,
        ).coerceIn(0f, 1f)
    }

    /**
     * Resolves the final radius and opacity for a single dab based on input.
     */
    fun resolve(
        settings: BrushSettings,
        pressure: Float,
        tiltX: Float,
        tiltY: Float,
        out: ResolvedDab,
    ) {
        val p = pressure
            .coerceIn(0f, 1f)
            .let { it.toDouble().pow(settings.pressureGamma.toDouble()).toFloat() }
```

```
        val tilt = tiltMagnitude(tiltX, tiltY)

        val pressureSize = if (settings.pressureToSize) {
            settings.minSizeRatio + (1f - settings.minSizeRatio) * p
        } else {
            1f
        }
        val tiltSize = 1f + settings.tiltToSize * tilt
        out.radius = (
            settings.baseRadiusPx * pressureSize * tiltSize
        ).coerceAtLeast(0.25f)

        val pressureOpacity = if (settings.pressureToOpacity) p else 1f

        // Tilt must not make a dab more opaque.
        val tiltOpacity = 1f - (0.45f * settings.tiltToOpacity * tilt)
        out.coverage = (pressureOpacity * tiltOpacity).coerceIn(0f, 1f)

        // Local dab alpha only. Global brush opacity is applied during stroke blit.
        out.opacity = out.coverage
    }
}

data class ResolvedDab(
    var radius: Float = 0f,
    var opacity: Float = 1f,
    var coverage: Float = 1f,
)
```

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

```
package com.wetinknext.engine.brush

import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.Brush
import androidx.compose.material.icons.filled.Create
import androidx.compose.material.icons.filled.Star
import androidx.compose.ui.graphics.vector.ImageVector

data class BrushUiCategory(
    val name: String,
    val icon: ImageVector,
    val brushes: List<BrushPreset>,
)

object BrushLibrary {
    val pencil6B = BrushPreset(
        id = "pencil_6b",
        settings = BrushSettings(
            name = "Pencil 6B",
            category = "Pencil",
            renderMode = BrushRenderMode.STAMP,
            baseRadiusPx = 8f,
            opacity = 0.82f,
            flow = 0.65f,
            spacing = 0.08f,
            spacingUsesDiameter = true,
            hardness = 0.72f,
            smoothing = 0.08f,
            streamline = 0.04f,
            pressureToSize = true,
            pressureToOpacity = true,
            pressureGamma = 1.25f,
            minSizeRatio = 0.30f,
            tiltToSize = 0.55f,
            grainAssetPath = "asset:brush/pencil_6b_grain.png",
            grainCanvasLocked = true,
            grainScale = 5.5f,
            textureDepth = 0.65f,
            textureContrast = 1.35f,
            blendPolicy = BlendPolicy.NORMAL_BUILDUP,
        ),
    )

    val gPen = BrushPreset(
        id = "g_pen",
        settings = BrushSettings(
            name = "G-Pen",
            category = "Ink",
            renderMode = BrushRenderMode.RIBBON,
            baseRadiusPx = 4.5f,
            opacity = 1f,
            flow = 1f,
            smoothing = 0.35f,
            streamline = 0.22f,
            pressureToSize = true,
            pressureToOpacity = false,
            pressureGamma = 1.7f,
            minSizeRatio = 0.06f,
            blendPolicy = BlendPolicy.NON_BUILDUP,
```

```
        )
    )

    val allCategories = listOf(
        BrushUiCategory(
            name = "Pencil",
            icon = Icons.Default.Create,
            brushes = listOf(pencil6B)
        ),
        BrushUiCategory(
            name = "Ink",
            icon = Icons.Default.Brush,
            brushes = listOf(gPen)
        ),
        BrushUiCategory(
            name = "Favorites",
            icon = Icons.Default.Star,
            brushes = emptyList()
        )
    )
}
```

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

```
package com.wetinknext.engine.brush

import kotlinx.serialization.Serializable

@Serializable
data class BrushPreset(
    val id: String,
    val settings: BrushSettings,
)
```

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

```
package com.wetinknext.engine.brush

import kotlinx.serialization.encodeToString
import kotlinx.serialization.json.Json

class BrushRepository(
    private val json: Json = defaultJson,
) {
    fun encode(preset: BrushPreset): String =
        json.encodeToString(preset)

    fun decode(value: String): BrushPreset =
        json.decodeFromString<BrushPreset>(value).let { preset ->
            preset.copy(settings = preset.settings.resolved())
        }

    companion object {
        val defaultJson = Json {
            encodeDefaults = true
            ignoreUnknownKeys = true
            explicitNulls = false
        }
    }
}
```

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

```
package com.wetinknext.engine.brush

import kotlinx.serialization.Serializable

@Serializable
enum class BrushRenderMode { STAMP, RIBBON, WET }

@Serializable
enum class BlendPolicy { NORMAL_BUILDUP, NON_BUILDUP }

@Serializable
enum class RibbonCap { ROUND, BUTT }

@Serializable
enum class RibbonJoin { MITER, ROUND, BEVEL }

@Serializable
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,
)

@Serializable
data class WetSettings(val wetness: Float = 0f, val spread: Float = 0f, val bleed: Float = 0f)

@Serializable
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(),
) {
    /** Returns a settings copy safe for every engine path. */
    fun resolved(): BrushSettings {
        val safeRadius = baseRadiusPx.coerceIn(0.25f, 4096f)
        val safeOpacity = opacity.coerceIn(0f, 1f)
        val safeFlow = flow.coerceIn(0f, 1f)
        val safeSpacing = spacing.coerceIn(0.001f, 4f)
        val safeGamma = pressureGamma.coerceIn(0.05f, 8f)
        val safeMinSize = minSizeRatio.coerceIn(0.01f, 1f)

        val safeVelocityToSize = velocityToSize.coerceIn(0f, 1f)
        val safeVelocityToOpacity = velocityToOpacity.coerceIn(0f, 1f)
        val safeTiltToSize = tiltToSize.coerceIn(0f, 1f)
        val safeTiltToOpacity = tiltToOpacity.coerceIn(0f, 1f)
        val safeTiltToRotation = tiltToRotation.coerceIn(0f, 1f)
        val safeSizeJitter = sizeJitter.coerceIn(0f, 1f)
        val safeOpacityJitter = opacityJitter.coerceIn(0f, 1f)
        val safeRotationJitter = rotationJitter.coerceIn(0f, 1f)
        val safeScatter = scatter.coerceAtLeast(0f)

        val safeGrainScale = grainScale.coerceIn(0.0001f, 256f)
        val safeTextureDepth = textureDepth.coerceIn(0f, 1f)
        val safeTextureContrast = textureContrast.coerceIn(0f, 4f)
        val safeAa = antiAliasLevel.coerceIn(0, 3)

        val safeRibbon = ribbon.copy(
            minWidthRatio = ribbon.minWidthRatio.coerceIn(0.01f, 1f),
            taperStartPx = ribbon.taperStartPx.coerceAtLeast(0f),
            taperEndPx = ribbon.taperEndPx.coerceAtLeast(0f),
            miterLimit = ribbon.miterLimit.coerceIn(1f, 32f),
            aaWidthPx = ribbon.aaWidthPx.coerceIn(0f, 16f),
            minPointDistancePx = ribbon.minPointDistancePx.coerceIn(0.05f, 32f),
        )
        val safeWet = wet.copy(
            wetness = wet.wetness.coerceIn(0f, 1f),
            spread = wet.spread.coerceIn(0f, 1f),
            bleed = wet.bleed.coerceIn(0f, 1f),
        )

        return copy(
```

```
            baseRadiusPx = safeRadius,
            opacity = safeOpacity,
            flow = safeFlow,
            spacing = safeSpacing,
            pressureGamma = safeGamma,
            minSizeRatio = safeMinSize,
            velocityToSize = safeVelocityToSize,
            velocityToOpacity = safeVelocityToOpacity,
            tiltToSize = safeTiltToSize,
            tiltToOpacity = safeTiltToOpacity,
            tiltToRotation = safeTiltToRotation,
            sizeJitter = safeSizeJitter,
            opacityJitter = safeOpacityJitter,
            rotationJitter = safeRotationJitter,
            scatter = safeScatter,
            grainScale = safeGrainScale,
            textureDepth = safeTextureDepth,
            textureContrast = safeTextureContrast,
            antiAliasLevel = safeAa,
            ribbon = safeRibbon,
            wet = safeWet,
        )
    }
}
```

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

```
package com.wetinknext.engine.brush

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

class BrushTexture {
    var textureId: Int = 0
        private set

    var width: Int = 0
        private set

    var height: Int = 0
        private set

    fun createFromRgba(
        width: Int,
        height: Int,
        rgba: ByteBuffer,
    ) {
        GlCheck.checkOnGlThread()
        release()

        val ids = IntArray(1)
        GLES30.glGenTextures(1, ids, 0)
        textureId = ids[0]

        check(textureId != 0) { "Failed to create brush texture" }

        this.width = width
        this.height = height

        GLES30.glBindTexture(GLES30.GL_TEXTURE_2D, textureId)

        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_REPEAT,
        )
        GLES30.glTexParameteri(
            GLES30.GL_TEXTURE_2D,
            GLES30.GL_TEXTURE_WRAP_T,
            GLES30.GL_REPEAT,
        )

        GLES30.glPixelStorei(
            GLES30.GL_UNPACK_ALIGNMENT,
            1,
        )

        GLES30.glTexImage2D(
            GLES30.GL_TEXTURE_2D,
```

```
            0,
            GLES30.GL_RGBA,
            width,
            height,
            0,
            GLES30.GL_RGBA,
            GLES30.GL_UNSIGNED_BYTE,
            rgba,
        )

        GLES30.glPixelStorei(
            GLES30.GL_UNPACK_ALIGNMENT,
            4,
        )

        GLES30.glBindTexture(
            GLES30.GL_TEXTURE_2D,
            0,
        )
    }

    fun release() {
        GlCheck.checkOnGlThread()
        if (textureId != 0) {
            GLES30.glDeleteTextures(
                1,
                intArrayOf(textureId),
                0,
            )
        }

        textureId = 0
        width = 0
        height = 0
    }

}
```

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] blit-pass.
 */
class CapsuleEmitter(
    private var settings: BrushSettings,
) {
    private val stabilizer = Stabilizer()
    private val pressureFilter = PressureFilter()
    private val resolvedDab = ResolvedDab()

    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
}

/**
 * ■■■■■■■ ■■■■■■ ■■■■■■■■■ stroke.
 * overflowCount ■■■■■■■■ ■■■■■■ ■■ ■■■■■■■■■■■■■.
 */
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)
    BrushDynamics.resolve(settings, filteredPressure, sample.tiltX, sample.tiltY, resolvedDab)
    val radius = resolvedDab.radius

    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)
        BrushDynamics.resolve(settings, filteredPressure, sample.tiltX, sample.tiltY, resolvedDab)
        val radius = resolvedDab.radius

        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
        }
    }

    /**
     * ■■■ ■■■■■■ ■■ ■■■■■■ ■■■■■■ ■■■■■■■■ ■■■, ■■■ ■■■ 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.BuildConfig
import com.wetinknext.engine.gl.GlCheck
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 ■■■ ■■■■■■■■ ■■■■■■■.
 *
 * ■■■■ instance:
 *   x0, y0, radius0,
 *   x1, y1, radius1
 *
 * ■■■■■■■■■ ■■■■■■■ ■■■■■■■■ ■■■■■■■■ AABB, ■ ■■■■■■■■■ ■■■■■
 * ■■■■■■■■■■■■■■■ ■ capsuleFragment ■■■■■ signed distance.
 *
 * ■■■■■■:
 * - target ■■■■■■■ ■■■■■ ■■■■■■■ ■■■■■■ ■■■■■■■■ ■■■■■■■ ■■■■■■■;
 * - ■■■■■■■■■■ ■■■■■■■■■■■■ ■ 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
    private val blendController = BlendController()
    private var uCanvasSize = -1
    private var uGrainTex = -1
    private var uGrainActive = -1
    private var uGrainScale = -1
    private var uGrainCanvasLocked = -1
    private var uTextureDepth = -1
    private var uTextureContrast = -1

    private var grainTextureId = 0
    private var grainScale = 1f
    private var grainCanvasLocked = true
    private var textureDepth = 1f
    private var textureContrast = 1f

    private val checkGlErrors: Boolean
        get() = BuildConfig.DEBUG

    /**
     * [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() {
        GlCheck.checkOnGlThread()
        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",
        )
        uCanvasSize = GLES30.glGetUniformLocation(
            currentProgram.id,
            "uCanvasSize",
        )
        uGrainTex = GLES30.glGetUniformLocation(
            currentProgram.id,
            "uGrainTex",
        )
        uGrainScale = GLES30.glGetUniformLocation(
            currentProgram.id,
            "uGrainScale",
```

```
)
uGrainActive = GLES30.glGetUniformLocation(
    currentProgram.id,
    "uGrainActive",
)
uGrainCanvasLocked = GLES30.glGetUniformLocation(
    currentProgram.id,
    "uGrainCanvasLocked",
)
uTextureDepth = GLES30.glGetUniformLocation(
    currentProgram.id,
    "uTextureDepth",
)
uTextureContrast = GLES30.glGetUniformLocation(
    currentProgram.id,
    "uTextureContrast",
)

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

// [redacted] AABB-[redacted].
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,
)

// [redacted] [redacted].
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
}

/**
 * ********** * ******* ******* ****** *********.
 *
```

```
 * [redacted] [redacted] preview [redacted] [redacted] MOVE.
 * [redacted] [redacted] [redacted] target [redacted] [redacted] [redacted].
 */
fun drawPending(
    target: RenderTarget,
    width: Int,
    height: Int,
    canvasToClip: FloatArray,
    colorLinear: FloatArray,
    blendPolicy: BlendPolicy,
) {
    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,
        blendPolicy = blendPolicy,
    )

    if (drawn) {
        uploadedSegmentCount = segmentCount
    }
}

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

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

/**
 * [redacted] [redacted] [redacted] [redacted] [redacted] [redacted] [firstSegment, firstSegment + count).
 *
 * [redacted] [redacted]: glVertexAttribPointer [redacted] [redacted] [redacted],
 * [redacted] [redacted] offset [redacted] instanced-[redacted] [redacted] [redacted] [redacted] [redacted].
 */
private fun drawRange(
    target: RenderTarget,
    width: Int,
    height: Int,
    canvasToClip: FloatArray,
    colorLinear: FloatArray,
    firstSegment: Int,
    count: Int,
    blendPolicy: BlendPolicy,
): 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)

blendController.begin(blendPolicy)

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),
)

GLES30.glUniform2f(
    uCanvasSize,
    width.toFloat(),
    height.toFloat(),
)

GLES30.glUniform1f(
    uGrainScale,
    grainScale,
)

GLES30.glUniform1i(
    uGrainActive,
    if (grainTextureId != 0) 1 else 0,
)

GLES30.glUniform1i(
    uGrainCanvasLocked,
    if (grainCanvasLocked) 1 else 0,
)

GLES30.glUniform1f(
    uTextureDepth,
    textureDepth,
)

GLES30.glUniform1f(
    uTextureContrast,
    textureContrast,
)

if (grainTextureId != 0) {
    GLES30.glActiveTexture(GLES30.GL_TEXTURE2)
    GLES30.glBindTexture(
        GLES30.GL_TEXTURE_2D,
        grainTextureId,
    )
    GLES30.glUniform1i(uGrainTex, 2)
}

/*
 * 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,
    )

    if (grainTextureId != 0) {
        GLES30.glActiveTexture(GLES30.GL_TEXTURE2)
        GLES30.glBindTexture(
            GLES30.GL_TEXTURE_2D,
            0,
        )
    }

    GLES30.glActiveTexture(GLES30.GL_TEXTURE0)

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

    blendController.end()

    GLES30.glBindFramebuffer(
        GLES30.GL_FRAMEBUFFER,
        0,
    )

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

    if (checkGlErrors) {
        checkNoGlError("CapsuleStrokeRenderer.drawRange")
    }
    return true
}

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

fun setGrainTexture(
    textureId: Int,
    scale: Float,
    canvasLocked: Boolean,
    depth: Float,
    contrast: Float,
) {
    grainTextureId = textureId
    grainScale = scale.coerceAtLeast(0.0001f)
    grainCanvasLocked = canvasLocked
    textureDepth = depth.coerceIn(0f, 1f)
    textureContrast = contrast.coerceIn(0f, 2f)
}

fun clearGrainTexture() {
    grainTextureId = 0
    grainScale = 1f
    grainCanvasLocked = true
```

```
        textureDepth = 1f
        textureContrast = 1f
    }

    /**
     * ■■■ GL-■■■■■■■■ ■■■■■■■■■■ ■■■■■■■ ■■ GL-■■■■■■■.
     */
    fun release() {
        GlCheck.checkOnGlThread()
        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
        uCanvasSize = -1
        uGrainTex = -1
        uGrainScale = -1
        uGrainActive = -1
        uGrainCanvasLocked = -1
        uTextureDepth = -1
        uTextureContrast = -1
        grainTextureId = 0

        segmentCount = 0
        uploadedSegmentCount = 0

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

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

    companion object {

        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.p
    ow(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 com.wetinknext.engine.core.DirtyRect
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, coverage: Float, flow: Float): Boolean { if (count >= ca
    pacity) { overflowCount++; return false }; floats.put(x); floats.put(y); floats.put(radius); floats.put(rotation); float
    s.put(coverage.coerceIn(0f, 1f)); floats.put(flow.coerceIn(0f, 1f)); count++; return true }
    fun prepareForUpload() { floats.position(0); floats.limit(count * FLOATS_PER_DAB) }
    fun prepareForUpload(firstDab: Int, dabCount: Int) {
        require(firstDab >= 0)
        require(dabCount >= 0)
        require(firstDab + dabCount <= count)
        floats.position(firstDab * FLOATS_PER_DAB)
        floats.limit((firstDab + dabCount) * 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)
    }

    /**
     * Includes the exact circular footprint of each emitted dab plus a caller
     * supplied safety fringe for analytic AA and textured-tip filtering.
     */
    fun includeDirtyRect(rect: DirtyRect, extraMargin: Float) {
        val safeExtraMargin = extraMargin.coerceAtLeast(0f)
        for (index in 0 until count) {
            val offset = index * FLOATS_PER_DAB
            val x = floats.get(offset)
            val y = floats.get(offset + 1)
            val radius = floats.get(offset + 2).coerceAtLeast(0f) + safeExtraMargin
            rect.include(
                left = x - radius,
                top = y - radius,
                right = x + radius,
                bottom = y + radius,
            )
        }
    }
    companion object { const val FLOATS_PER_DAB = 6; 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.GlCheck
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
    private var uCanvasSize = -1
    private var uGrainTex = -1
    private var uGrainActive = -1
    private var uGrainScale = -1
    private var uGrainCanvasLocked = -1
    private var uTextureDepth = -1
    private var uTextureContrast = -1
    private var uStrokeOpacity = -1
    private var uShapeTex = -1
    private var uShapeActive = -1
    private var uReverseShape = -1
    private var uRgbToAlpha = -1

    private var grainTextureId = 0
    private var grainScale = 1f
    private var grainCanvasLocked = true
    private var textureDepth = 1f
    private var textureContrast = 1f
    private var shapeTextureId = 0
    private var reverseShape = false
    private var rgbToAlpha = false

    private var uploadedDabCount = 0
    private val blendController = BlendController()

    /** Number of dabs already written into the current stroke target. */
    val uploadedCount: Int get() = uploadedDabCount

    fun create() {
        GlCheck.checkOnGlThread()
        release()
        uploadedDabCount = 0
        program = GlProgram(ShaderLib.dabVertex, ShaderLib.dabFragment)
        val currentProgram = checkNotNull(program)
        currentProgram.use()
        uCanvasToClip = GLES30.glGetUniformLocation(currentProgram.id, "uCanvasToClip")
        uColorLinear = GLES30.glGetUniformLocation(currentProgram.id, "uColorLinear")
        uCanvasSize = GLES30.glGetUniformLocation(currentProgram.id, "uCanvasSize")
        uGrainTex = GLES30.glGetUniformLocation(currentProgram.id, "uGrainTex")
        uGrainActive = GLES30.glGetUniformLocation(currentProgram.id, "uGrainActive")
        uGrainScale = GLES30.glGetUniformLocation(currentProgram.id, "uGrainScale")
        uGrainCanvasLocked = GLES30.glGetUniformLocation(currentProgram.id, "uGrainCanvasLocked")
        uTextureDepth = GLES30.glGetUniformLocation(currentProgram.id, "uTextureDepth")
        uTextureContrast = GLES30.glGetUniformLocation(currentProgram.id, "uTextureContrast")
        uStrokeOpacity = GLES30.glGetUniformLocation(currentProgram.id, "uStrokeOpacity")
        uShapeTex = GLES30.glGetUniformLocation(currentProgram.id, "uShapeTex")
        uShapeActive = GLES30.glGetUniformLocation(currentProgram.id, "uShapeActive")
        uReverseShape = GLES30.glGetUniformLocation(currentProgram.id, "uReverseShape")
        uRgbToAlpha = GLES30.glGetUniformLocation(currentProgram.id, "uRgbToAlpha")

        check(uCanvasToClip >= 0 && uColorLinear >= 0 && uStrokeOpacity >= 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, DabBuffer.FLOATS_PER_DAB * Float.SIZE_BYTES, 0)
    GLES30.glVertexAttribDivisor(1, 1)
    GLES30.glEnableVertexAttribArray(2)
    GLES30.glVertexAttribPointer(2, 2, GLES30.GL_FLOAT, false, DabBuffer.FLOATS_PER_DAB * Float.SIZE_BYTES, 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,
    blendPolicy: BlendPolicy,
    strokeOpacity: Float = 1f,
) {
    if (dabs.count == 0) return
    require(colorLinear.size >= 3)

    drawRange(
        target = target,
        width = width,
        height = height,
        canvasToFbo = canvasToFbo,
        dabs = dabs,
        firstDab = 0,
        count = dabs.count,
        colorLinear = colorLinear,
        blendPolicy = blendPolicy,
        strokeOpacity = strokeOpacity,
    )
}

fun release() {
    GlCheck.checkOnGlThread()
    program?.release()
    program = null
    uploadedDabCount = 0
    grainTextureId = 0
    shapeTextureId = 0
    reverseShape = false
    rgbToAlpha = false
    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
    uStrokeOpacity = -1
}

fun beginStroke() {
    uploadedDabCount = 0
}

fun clearStrokeData() {
    uploadedDabCount = 0
}

fun setGrainTexture(
    textureId: Int,
    scale: Float,
    canvasLocked: Boolean,
    depth: Float,
    contrast: Float,
) {
    grainTextureId = textureId
    grainScale = scale.coerceAtLeast(0.0001f)
    grainCanvasLocked = canvasLocked
    textureDepth = depth.coerceIn(0f, 1f)
    textureContrast = contrast.coerceIn(0f, 2f)
}

fun clearGrainTexture() {
    grainTextureId = 0
    grainScale = 1f
    grainCanvasLocked = true
    textureDepth = 1f
    textureContrast = 1f
```

```
}

/** Sets a tip mask sampled once per dab in local dab coordinates. */
fun setShapeTexture(textureId: Int, reverse: Boolean, rgbToAlpha: Boolean) {
    shapeTextureId = textureId
    reverseShape = reverse
    this.rgbToAlpha = rgbToAlpha
}

fun clearShapeTexture() {
    shapeTextureId = 0
    reverseShape = false
    rgbToAlpha = false
}

fun drawPendingInto(
    target: RenderTarget,
    width: Int,
    height: Int,
    canvasToFbo: FloatArray,
    dabs: DabBuffer,
    colorLinear: FloatArray,
    blendPolicy: BlendPolicy,
    strokeOpacity: Float = 1f,
) {
    val first = uploadedDabCount
    val count = dabs.count - first
    if (count <= 0) return

    dabs.prepareForUpload(first, count)
    drawRange(
        target = target,
        width = width,
        height = height,
        canvasToFbo = canvasToFbo,
        dabs = dabs,
        firstDab = first,
        count = count,
        colorLinear = colorLinear,
        blendPolicy = blendPolicy,
        strokeOpacity = strokeOpacity,
    )
    uploadedDabCount = dabs.count
    dabs.finishUpload()
}

private fun drawRange(
    target: RenderTarget,
    width: Int,
    height: Int,
    canvasToFbo: FloatArray,
    dabs: DabBuffer,
    firstDab: Int,
    count: Int,
    colorLinear: FloatArray,
    blendPolicy: BlendPolicy,
    strokeOpacity: Float,
) {
    val currentProgram = program ?: return
    target.bind()
    GLES30.glViewport(0, 0, width, height)
    blendController.begin(blendPolicy)
    currentProgram.use()
    GLES30.glUniformMatrix4fv(uCanvasToClip, 1, false, canvasToFbo, 0)
    GLES30.glUniform3f(uColorLinear, colorLinear[0], colorLinear[1], colorLinear[2])

    GLES30.glUniform2f(uCanvasSize, width.toFloat(), height.toFloat())
    GLES30.glUniform1i(uGrainActive, if (grainTextureId != 0) 1 else 0)
    GLES30.glUniform1f(uGrainScale, grainScale)
    GLES30.glUniform1i(uGrainCanvasLocked, if (grainCanvasLocked) 1 else 0)
    GLES30.glUniform1f(uTextureDepth, textureDepth)
    GLES30.glUniform1f(uTextureContrast, textureContrast)
    GLES30.glUniform1f(uStrokeOpacity, strokeOpacity.coerceIn(0f, 1f))
    GLES30.glUniform1i(uShapeActive, if (shapeTextureId != 0) 1 else 0)
    GLES30.glUniform1i(uReverseShape, if (reverseShape) 1 else 0)
    GLES30.glUniform1i(uRgbToAlpha, if (rgbToAlpha) 1 else 0)

    if (grainTextureId != 0) {
        GLES30.glActiveTexture(GLES30.GL_TEXTURE2)
        GLES30.glBindTexture(GLES30.GL_TEXTURE_2D, grainTextureId)
        GLES30.glUniform1i(uGrainTex, 2)
    }
    if (shapeTextureId != 0) {
        GLES30.glActiveTexture(GLES30.GL_TEXTURE3)
        GLES30.glBindTexture(GLES30.GL_TEXTURE_2D, shapeTextureId)
        GLES30.glUniform1i(uShapeTex, 3)
    }

    GLES30.glBindVertexArray(vaoId)
    GLES30.glBindBuffer(GLES30.GL_ARRAY_BUFFER, instanceBufferId)
    GLES30.glBufferSubData(
        GLES30.GL_ARRAY_BUFFER,
        firstDab * DabBuffer.FLOATS_PER_DAB * Float.SIZE_BYTES,
```

```
            count * DabBuffer.FLOATS_PER_DAB * Float.SIZE_BYTES,
            dabs.floats,
        )

        // Reset offsets for instancing
        GLES30.glVertexAttribPointer(
            1, 4, GLES30.GL_FLOAT, false, DabBuffer.FLOATS_PER_DAB * Float.SIZE_BYTES, 0
        )
        GLES30.glVertexAttribPointer(
            2, 2, GLES30.GL_FLOAT, false, DabBuffer.FLOATS_PER_DAB * Float.SIZE_BYTES, 16
        )

        // Manual attribute offset for ES 3.0 (since BaseInstance is 3.1+)
        GLES30.glVertexAttribPointer(
            1, 4, GLES30.GL_FLOAT, false, DabBuffer.FLOATS_PER_DAB * Float.SIZE_BYTES,
            (firstDab * DabBuffer.FLOATS_PER_DAB * Float.SIZE_BYTES)
        )
        GLES30.glVertexAttribPointer(
            2, 2, GLES30.GL_FLOAT, false, DabBuffer.FLOATS_PER_DAB * Float.SIZE_BYTES,
            (firstDab * DabBuffer.FLOATS_PER_DAB * Float.SIZE_BYTES + 16)
        )

        GLES30.glDrawArraysInstanced(GLES30.GL_TRIANGLE_STRIP, 0, 4, count)

        if (grainTextureId != 0) {
            GLES30.glActiveTexture(GLES30.GL_TEXTURE2)
            GLES30.glBindTexture(GLES30.GL_TEXTURE_2D, 0)
        }
        if (shapeTextureId != 0) {
            GLES30.glActiveTexture(GLES30.GL_TEXTURE3)
            GLES30.glBindTexture(GLES30.GL_TEXTURE_2D, 0)
        }
        GLES30.glActiveTexture(GLES30.GL_TEXTURE0)

        GLES30.glBindBuffer(GLES30.GL_ARRAY_BUFFER, 0)
        GLES30.glBindVertexArray(0)
        blendController.end()
        GLES30.glBindFramebuffer(GLES30.GL_FRAMEBUFFER, 0)
    }
}
```

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 = 1
    f) {
    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/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 fil
    tered = 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 = filte
    red[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 val pressureFilter = PressureFilter()
    private val resolvedDab = ResolvedDab()
    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 lastTiltX = 0f
    private var lastTiltY = 0f
    private var carriedDistance = 0f
    private var movedDuringStroke = false
    private var pointCount = 0
    private var p0x = 0f; private var p0y = 0f; private var p0p = 0f
    private var p1x = 0f; private var p1y = 0f; private var p1p = 0f
    private var p2x = 0f; private var p2y = 0f; private var p2p = 0f
    private var p3x = 0f; private var p3y = 0f; private var p3p = 0f

    fun reset() {
        active = false
        pointerId = -1
        hasLast = false
        carriedDistance = 0f
        movedDuringStroke = false
        pointCount = 0
```

```
        stabilizer.reset()
        pressureFilter.reset()
    }

    fun begin(batch: InputBatch, out: DabBuffer) {
        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
        lastPressure = pressureFilter.filter(s.timestampNanos, s.pressure)
        lastTiltX = s.tiltX
        lastTiltY = s.tiltY
        hasLast = true
        p0x = lastX; p0y = lastY; p0p = lastPressure
        p1x = lastX; p1y = lastY; p1p = lastPressure
        p2x = lastX; p2y = lastY; p2p = lastPressure
        p3x = lastX; p3y = lastY; p3p = lastPressure
        pointCount = 1
    }

    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)
                val pressure = pressureFilter.filter(s.timestampNanos, s.pressure)
                if (settings.smoothing <= .0001f) {
                    addPoint(stabilizer.x, stabilizer.y, pressure, s.tiltX, s.tiltY, out)
                } else {
                    appendSmoothedPoint(stabilizer.x, stabilizer.y, pressure, out)
                }
            }
        }
    }

    fun finish(out: DabBuffer, cancel: Boolean) {
        if (!active) return
        if (cancel) {
            reset()
            return
        }
        if (settings.smoothing > .0001f && pointCount >= 3) {
            flushSmoothedTail(out)
        }
        if (!movedDuringStroke || carriedDistance > .001f) {
            addDab(
                x = lastX,
                y = lastY,
                pressure = lastPressure,
                tiltX = lastTiltX,
                tiltY = lastTiltY,
                out = out,
            )
        }
        reset()
    }

    private fun appendSmoothedPoint(x: Float, y: Float, pressure: Float, out: DabBuffer) {
        p0x = p1x; p0y = p1y; p0p = p1p
        p1x = p2x; p1y = p2y; p1p = p2p
        p2x = p3x; p2y = p3y; p2p = p3p
        p3x = x; p3y = y; p3p = pressure
        pointCount = (pointCount + 1).coerceAtMost(4)

        when (pointCount) {
            2 -> addPoint(x, y, pressure, lastTiltX, lastTiltY, out)
            4 -> interpolateCatmullRom(out)
        }
    }

    /** Processes the last delayed Catmull-Rom segment with a duplicated endpoint. */
    private fun flushSmoothedTail(out: DabBuffer) {
        p0x = p1x; p0y = p1y; p0p = p1p
        p1x = p2x; p1y = p2y; p1p = p2p
        p2x = p3x; p2y = p3y; p2p = p3p
        p3x = p2x; p3y = p2y; p3p = p2p
        pointCount = 4
        interpolateCatmullRom(out)
    }

    private fun interpolateCatmullRom(out: DabBuffer) {
        val dx = p2x - p1x
        val dy = p2y - p1y
        val distance = sqrt(dx * dx + dy * dy)
        if (distance < .0001f) return

        val steps = (distance / maxOf(1f, settings.baseRadiusPx * .25f))
```

```
        .toInt()
        .coerceIn(2, 64)

    for (index in 1..steps) {
        val t = index.toFloat() / steps
        val t2 = t * t
        val t3 = t2 * t
        val x = .5f * (
            2f * p1x + (-p0x + p2x) * t +
                (2f * p0x - 5f * p1x + 4f * p2x - p3x) * t2 +
                (-p0x + 3f * p1x - 3f * p2x + p3x) * t3
            )
        val y = .5f * (
            2f * p1y + (-p0y + p2y) * t +
                (2f * p0y - 5f * p1y + 4f * p2y - p3y) * t2 +
                (-p0y + 3f * p1y - 3f * p2y + p3y) * t3
            )
        val pressure = p1p + (p2p - p1p) * t
        addPoint(x, y, pressure, lastTiltX, lastTiltY, out)
    }
}

private fun addPoint(x: Float, y: Float, p: Float, tx: Float, ty: Float, out: DabBuffer) {
    val dx = x - lastX
    val dy = y - lastY
    val distance = sqrt(dx * dx + dy * dy)
    if (distance <= .0001f) return
    if (distance > 1.5f) movedDuringStroke = true

    var travelled = 0f
    while (travelled < distance) {
        val tProbe = (travelled / distance).coerceIn(0f, 1f)
        val probePressure = lastPressure + (p - lastPressure) * tProbe
        val probeTiltX = lastTiltX + (tx - lastTiltX) * tProbe
        val probeTiltY = lastTiltY + (ty - lastTiltY) * tProbe

        BrushDynamics.resolve(
            settings = settings,
            pressure = probePressure,
            tiltX = probeTiltX,
            tiltY = probeTiltY,
            out = resolvedDab,
        )

        val step = spacingForRadius(resolvedDab.radius)
        // Radius can shrink with pressure, leaving more carried distance
        // than the new step. Emit at the current point and reset it.
        val needed = (step - carriedDistance).coerceAtLeast(0f)

        if (travelled + needed > distance) {
            carriedDistance += distance - travelled
            break
        }

        travelled += needed
        val t = (travelled / distance).coerceIn(0f, 1f)
        addDab(
            x = lastX + dx * t,
            y = lastY + dy * t,
            pressure = lastPressure + (p - lastPressure) * t,
            tiltX = lastTiltX + (tx - lastTiltX) * t,
            tiltY = lastTiltY + (ty - lastTiltY) * t,
            out = out,
        )
        carriedDistance = 0f
    }
    lastX = x
    lastY = y
    lastPressure = p
    lastTiltX = tx
    lastTiltY = ty
}

private fun spacingForRadius(radius: Float): Float {
    val unit = if (settings.spacingUsesDiameter) radius * 2f else radius
    return (unit * settings.spacing).coerceIn(.25f, 256f)
}

private fun addDab(
    x: Float,
    y: Float,
    pressure: Float,
    tiltX: Float,
    tiltY: Float,
    out: DabBuffer,
) {
    BrushDynamics.resolve(
        settings = settings,
        pressure = pressure,
        tiltX = tiltX,
        tiltY = tiltY,
        out = resolvedDab,
    )
```

```
        out.add(
            x = x,
            y = y,
            radius = resolvedDab.radius,
            rotation = 0f,
            coverage = resolvedDab.coverage,
            flow = resolveFlow(pressure),
        )
    }

    private fun resolveFlow(pressure: Float): Float {
        if (!settings.pressureToOpacity) return settings.flow.coerceIn(0f, 1f)
        val p = pressure.coerceIn(0f, 1f)
            .toDouble()
            .pow(settings.pressureGamma.toDouble())
            .toFloat()
        return (settings.flow * (.04f + .96f * p)).coerceIn(0f, 1f)
    }
}
```

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

```
package com.wetinknext.engine.brush

import android.content.Context
import android.graphics.Bitmap
import android.graphics.BitmapFactory
import java.nio.ByteBuffer
import java.nio.ByteOrder
import java.util.concurrent.ExecutorService
import java.util.concurrent.Executors

data class LoadedBrushTexture(
    val path: String,
    val width: Int,
    val height: Int,
    val rgba: ByteBuffer,
)

class TextureLoader(
    private val context: Context,
    private val executor: ExecutorService =
        Executors.newSingleThreadExecutor(),
) {
    fun loadAsync(
        path: String,
        onLoaded: (LoadedBrushTexture) -> Unit,
        onError: (Throwable) -> Unit,
    ) {
        executor.execute {
            try {
                val loaded = decode(path)
                onLoaded(loaded)
            } catch (error: Throwable) {
                onError(error)
            }
        }
    }

    private fun decode(path: String): LoadedBrushTexture {
        val source = decodeBitmap(path)

        val bitmap = if (source.config == Bitmap.Config.ARGB_8888) {
            source
        } else {
            source.copy(Bitmap.Config.ARGB_8888, false)
        }

        val width = bitmap.width
        val height = bitmap.height

        val rgba = ByteBuffer
            .allocateDirect(width * height * 4)
            .order(ByteOrder.nativeOrder())

        bitmap.copyPixelsToBuffer(rgba)
        rgba.position(0)

        if (bitmap !== source && !bitmap.isRecycled) {
            bitmap.recycle()
        }

        if (!source.isRecycled && source !== bitmap) {
            source.recycle()
        }

        return LoadedBrushTexture(
            path = path,
            width = width,
            height = height,
```

```
            rgba = rgba,
        )
    }

    private fun decodeBitmap(path: String): Bitmap {
        val bitmap = if (path.startsWith("asset:")) {
            val assetPath = path.removePrefix("asset:")

            context.assets.open(assetPath).use { input ->
                BitmapFactory.decodeStream(input)
            }
        } else {
            BitmapFactory.decodeFile(path)
        }

        return requireNotNull(bitmap) {
            "Cannot decode brush texture: $path"
        }
    }

    fun shutdown() {
        executor.shutdown()
    }
}
```

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() {
        GlCheck.checkOnGlThread()
        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() {
        GlCheck.checkOnGlThread()
        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
    private var documentUsesHalfFloat = false

    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
```

```
        documentUsesHalfFloat = caps.supportsHalfFloatColorBuffer

        val background = newLayer("■■■", documentUsesHalfFloat).also {
            it.isLocked = true
            it.target.clear(1f, 1f, 1f, 1f)
        }
        layers += background

        val drawing = newLayer("■■■■ 1", documentUsesHalfFloat)
        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, documentUsesHalfFloat)
        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
        documentUsesHalfFloat = false
    }

    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.util.Log
import android.view.MotionEvent
import androidx.compose.ui.graphics.toArgb
import com.wetinknext.BuildConfig
import com.wetinknext.engine.brush.BrushSettings
import com.wetinknext.engine.brush.BrushTexture
import com.wetinknext.engine.brush.LoadedBrushTexture
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.DeflateTileCompressor
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.ConcurrentLinkedQueue
import java.util.concurrent.Executors
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 undoExecutor = Executors.newSingleThreadExecutor()
    private val tileCompressor = DeflateTileCompressor()
    private val pendingUndoEntries = ConcurrentLinkedQueue<UndoEntry>()
    private var glThread: Thread? = null

    private var geometry: CanvasGeometry? = null
    private var compositor: Compositor? = null
    private var dabRenderer: DabRenderer? = null
    private var capsuleRenderer: CapsuleStrokeRenderer? = null
    private var grainTexture: BrushTexture? = null
    private var grainPath: String? = null
    private var shapeTexture: BrushTexture? = null
    private var shapePath: String? = 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 var capsulePreviewInitialized = false
    private var stampPreviewInitialized = false
    private val cancelRequested = AtomicBoolean(false)

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

```
    fun setOnSecondaryPointerDown(listener: () -> Unit) {
        inputCapturer.onSecondaryPointerDown = listener
    }

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

    fun requestCancelFromInput() {
        cancelRequested.set(true)
    }

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

    fun redo() {
        resetStroke()
        val entry = undoManager.peekRedo() ?: return
        val layer = layerStack.findLayerById(entry.layerId) ?: return
        TileSnapshotRestore.restore(layer.target, entry.afterTiles)
        if (!undoManager.commitRedo(entry)) return
        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)))

    fun setBrushColor(color: androidx.compose.ui.graphics.Color) =
        updateBrush(brushSettings.copy(colorArgb = color.toArgb().toLong() and 0xFFFFFFFFL))

    fun applyBrush(settings: BrushSettings) {
        val nextSettings = settings
            .copy(colorArgb = brushSettings.colorArgb)
            .resolved()
        updateBrush(nextSettings)
    }

    fun setBrushSettings(settings: BrushSettings) {
        resetStroke()
        updateBrush(settings)
    }

    /** ******* ****** ** ****** ******* ******* ** ********* ******: ********** ** ********** DOWN. */
    private fun updateBrush(next: BrushSettings) {
        brushSettings = next

        ColorSpaces.srgb8ToLinear(next.colorArgb, strokeColorLinear)
```

```
    if (strokeActive) pendingBrush = next else applyBrushToEmitters(next)
    publishState()
}

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

fun requestState() {
    publishState()
}

fun checkOnGlThread() {
    val current = Thread.currentThread()
    check(current === glThread) {
        "GL resource accessed outside GLSurfaceView render thread: current=$current, expected=$glThread"
    }
}

override fun onSurfaceCreated(gl: GL10?, config: EGLConfig?) {
    val currentThread = Thread.currentThread()
    glThread = currentThread
    GlCheck.setGlThread(currentThread)
    releaseGlObjects()
    if (undoExecutor.isShutdown) {
        undoExecutor = Executors.newSingleThreadExecutor()
    }
    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()
    processPendingUndo()

    if (strokeActive && brushSettings.renderMode == BrushRenderMode.RIBBON) {
        renderCapsulePreview()
    } else if (strokeActive && brushSettings.renderMode == BrushRenderMode.STAMP && dabBuffer.count > 0) {
        // No longer clearing and drawing everything here, handled incrementally in drainInput
    }

    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 = if (brushSettings.renderMode == BrushRenderMode.STAMP) {
            brushSettings.opacity.coerceIn(0f, 1f)
        } else {
            (brushSettings.opacity * brushSettings.flow).coerceIn(0f, 1f)
        },
        canvasToClip = canvasToClipMatrix,
    )
}

private fun processPendingUndo() {
    while (true) {
        val entry = pendingUndoEntries.poll() ?: break
        undoManager.push(entry)
        publishState()
    }
}

/** 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()
    pendingUndoEntries.clear()
    dabRenderer?.release()
    dabRenderer = null
    grainTexture?.release()
    grainTexture = null
    grainPath = null
    shapeTexture?.release()
    shapeTexture = null
    shapePath = null
    capsuleRenderer?.release()
    capsuleRenderer = null
    strokeBlitter?.release()
    strokeBlitter = null
    pendingBrush = null
    compositor?.release()
    compositor = null
    geometry?.release()
    geometry = null
    undoExecutor.shutdownNow()
    pendingUndoEntries.clear()
    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 {
                            dabRenderer?.beginStroke()
                            stampEmitter.begin(batch, dabBuffer)
                            strokeTarget.clear(0f, 0f, 0f, 0f)
                            stampPreviewInitialized = true

                            drawPendingStampPreview("DOWN")
                        }
                    }
                }

                InputAction.MOVE -> if (strokeActive) {
```

```
                    if (brushSettings.renderMode == BrushRenderMode.RIBBON) {
                        capsuleEmitter.append(
                            batch = batch,
                            out = checkNotNull(capsuleRenderer),
                        )
                    } else {
                        stampEmitter.append(batch, dabBuffer)
                        drawPendingStampPreview("MOVE")
                    }
                }

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

                        // UP ■■■■■■■ ■■■■■■■■ ■■■■■■■■■■■ historical samples,
                        // ■■■■■■ ■■■■■■■■ ■■■■■■, ■■■■■■■■ ■■■■ ■■■■■■■■ 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
    val blitter = strokeBlitter ?: return

    // Complete the isolated local-coverage mask before it is composited.
    // Global brush opacity is deliberately applied only by the blit.
    drawPendingStampPreview("UP")

    // Both snapshots use the actual layer format (RGBA8 or RGBA16F).
    val beforeTilesRaw = TileSnapshotCapture.capture(layer.target, dirtyBounds)
    blitter.blit(
        layer = layer.target,
        geometry = checkNotNull(geometry),
        strokeTextureId = strokeTarget.textureId,
        canvasToFbo = canvasToFboMatrix,
        width = layerStack.canvasWidth,
        height = layerStack.canvasHeight,
        opacity = brushSettings.opacity.coerceIn(0f, 1f),
    )
    layer.version++
    val afterTilesRaw = TileSnapshotCapture.capture(layer.target, dirtyBounds)

    val layerId = layer.id
    undoExecutor.execute {
        val beforeTiles = beforeTilesRaw.map { it.compress(tileCompressor) }
        val afterTiles = afterTilesRaw.map { it.compress(tileCompressor) }
        pendingUndoEntries.add(UndoEntry(layerId, 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()
    dabBuffer.includeDirtyRect(
        rect = strokeDirtyRect,
        extraMargin = STAMP_DIRTY_MARGIN_PX,
    )
    strokeDirtyRect.clamp(layerStack.canvasWidth.toFloat(), layerStack.canvasHeight.toFloat())
    strokeDirtyRect.toPixelBounds(out)
    return out[2] > out[0] && out[3] > out[1]
}

private fun drawPendingStampPreview(phase: String) {
    val renderer = dabRenderer ?: return
    check(renderer.uploadedCount <= dabBuffer.count) {
```

```
        "Stamp preview rewound: uploaded=${renderer.uploadedCount}, dabs=${dabBuffer.count}"
    }
    if (BuildConfig.DEBUG) {
        Log.d(
            BRUSH_DIAG_TAG,
            "phase=$phase dabs=${dabBuffer.count} pending=${dabBuffer.count - renderer.uploadedCount}",
        )
    }
    renderer.drawPendingInto(
        target = strokeTarget,
        width = layerStack.canvasWidth,
        height = layerStack.canvasHeight,
        canvasToFbo = canvasToFboMatrix,
        dabs = dabBuffer,
        colorLinear = strokeColorLinear,
        blendPolicy = brushSettings.blendPolicy,
        strokeOpacity = 1f,
    )
    check(renderer.uploadedCount == dabBuffer.count) {
        "Stamp preview did not consume all dabs: uploaded=${renderer.uploadedCount}, dabs=${dabBuffer.count}"
    }
}

private fun resetStroke() {
    strokeActive = false
    stampEmitter.reset()
    dabBuffer.clear()
    capsuleEmitter.reset()
    capsuleRenderer?.clearStrokeData()
    capsulePreviewInitialized = false
    stampPreviewInitialized = false
    dabRenderer?.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,
        ),
    )
}

/**
 * ■■■■■■■■■■■ ■■■■■■■ ■■■■■■ GLSurfaceView.queueEvent,
 * ■■ ■■■■■ ■■ GL-■■■■■■■ ■ current EGL context.
 */
fun applyLoadedGrain(
    loaded: LoadedBrushTexture,
    scale: Float,
    canvasLocked: Boolean,
    depth: Float,
    contrast: Float,
) {
    val oldTexture = grainTexture
    val newTexture = BrushTexture()

    try {
        newTexture.createFromRgba(
            width = loaded.width,
            height = loaded.height,
            rgba = loaded.rgba,
        )
```

```
        capsuleRenderer?.setGrainTexture(
            textureId = newTexture.textureId,
            scale = scale,
            canvasLocked = canvasLocked,
            depth = depth,
            contrast = contrast,
        )

        dabRenderer?.setGrainTexture(
            textureId = newTexture.textureId,
            scale = scale,
            canvasLocked = canvasLocked,
            depth = depth,
            contrast = contrast,
        )

        grainTexture = newTexture
        grainPath = loaded.path

        oldTexture?.release()
    } catch (error: Throwable) {
        newTexture.release()
        throw error
    }
}

fun clearGrainTexture() {
    capsuleRenderer?.clearGrainTexture()
    dabRenderer?.clearGrainTexture()

    grainTexture?.release()
    grainTexture = null
    grainPath = null
}

/** Must run on the GL thread. Shape masks affect STAMP dabs only. */
fun applyLoadedShape(
    loaded: LoadedBrushTexture,
    reverse: Boolean,
    rgbToAlpha: Boolean,
) {
    val oldTexture = shapeTexture
    val newTexture = BrushTexture()

    try {
        newTexture.createFromRgba(loaded.width, loaded.height, loaded.rgba)
        dabRenderer?.setShapeTexture(newTexture.textureId, reverse, rgbToAlpha)
        shapeTexture = newTexture
        shapePath = loaded.path
        oldTexture?.release()
    } catch (error: Throwable) {
        newTexture.release()
        throw error
    }
}

fun clearShapeTexture() {
    dabRenderer?.clearShapeTexture()
    shapeTexture?.release()
    shapeTexture = null
    shapePath = null
}

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

private fun renderCapsulePreview() {
    val renderer = capsuleRenderer ?: return

    if (!capsulePreviewInitialized) {
        strokeTarget.clear(
            red = 0f,
            green = 0f,
            blue = 0f,
            alpha = 0f,
        )
        capsulePreviewInitialized = true
    }

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

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

    if (layer.isLocked) return
```

```
    if (!capsuleEmitter.hasStroke) return
    if (!computeCapsuleDirtyBounds(dirtyBounds)) return

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

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

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

    /*
     * ****** ****** StrokeBlitter:
     * strokeTarget -> layer.target
     *
     * ** ******* **************:
     * 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 afterRaw = TileSnapshotCapture.capture(
        layer.target,
        dirtyBounds,
    )

    val layerId = layer.id
    undoExecutor.execute {
        val beforeTiles = beforeRaw.map { it.compress(tileCompressor) }
        val afterTiles = afterRaw.map { it.compress(tileCompressor) }
        pendingUndoEntries.add(UndoEntry(layerId, beforeTiles, afterTiles))
    }

    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 {
    private const val BRUSH_DIAG_TAG = "BrushDiag"
```

```
        const val DEFAULT_CANVAS_WIDTH = 1500
        const val DEFAULT_CANVAS_HEIGHT = 2000
        private const val AA_MARGIN_PX = 4f
        private const val STAMP_DIRTY_MARGIN_PX = 4f
    }
}
```

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
import com.wetinknext.engine.brush.BrushSettings
import com.wetinknext.engine.brush.TextureLoader
import com.wetinknext.engine.brush.LoadedBrushTexture

/** 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()

    private val textureLoader = TextureLoader(context)
    private var grainGeneration = 0L
    private var shapeGeneration = 0L

    var onTextureError: ((String, Throwable) -> Unit)? = null

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

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

    override fun onTouchEvent(event: MotionEvent): Boolean {
        val handled = engineRenderer.onTouchEvent(event)
```

```
    if (handled) requestRender()
    return handled || super.onTouchEvent(event)
}

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

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

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

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

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

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

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

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

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

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

fun applyBrush(settings: BrushSettings) = queueEvent {
    engineRenderer.applyBrush(settings)
    requestRender()
}

fun setBrushSettings(settings: BrushSettings) = queueEvent {
    engineRenderer.setBrushSettings(settings)
    requestRender()
}

fun setBrushColor(color: androidx.compose.ui.graphics.Color) = queueEvent {
    engineRenderer.setBrushColor(color)
    requestRender()
}

fun loadGrainTexture(
    path: String,
    scale: Float = 1f,
    canvasLocked: Boolean = true,
    depth: Float = 1f,
    contrast: Float = 1f,
) {
    val generation = ++grainGeneration

    textureLoader.loadAsync(
        path = path,
        onLoaded = { loaded ->
            queueEvent {
                if (generation != grainGeneration) {
                    return@queueEvent
                }

                engineRenderer.applyLoadedGrain(
                    loaded = loaded,
                    scale = scale,
                    canvasLocked = canvasLocked,
                    depth = depth,
```

```
                    contrast = contrast,
                )
                requestRender()
            }
        },
        onError = { error ->
            post {
                onTextureError?.invoke(path, error)
            }
        },
    )
}

fun clearGrainTexture() {
    grainGeneration++

    queueEvent {
        engineRenderer.clearGrainTexture()
        requestRender()
    }
}

fun loadShapeTexture(
    path: String,
    reverse: Boolean = false,
    rgbToAlpha: Boolean = false,
) {
    val generation = ++shapeGeneration
    textureLoader.loadAsync(
        path = path,
        onLoaded = { loaded ->
            queueEvent {
                if (generation != shapeGeneration) return@queueEvent
                engineRenderer.applyLoadedShape(loaded, reverse, rgbToAlpha)
                requestRender()
            }
        },
        onError = { error -> post { onTextureError?.invoke(path, error) } },
    )
}

fun clearShapeTexture() {
    shapeGeneration++
    queueEvent {
        engineRenderer.clearShapeTexture()
        requestRender()
    }
}

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

override fun onDetachedFromWindow() {
    grainGeneration++
    shapeGeneration++
    textureLoader.shutdown()

    // ******** listeners, ****** ********* ******* *** ******** * ********
    onTextureError = null
    onEditorStateChange = null
    engineRenderer.onStateChange = null
    engineRenderer.setOnSecondaryPointerDown { }

    // ********* *********** GL-********, ** ** ************** ** 100% ****** ******
    queueEvent {
        // ********* ********* ****** ****** ***************
        engineRenderer.cancelActiveStroke()
        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()).asFloatBuf
    fer()
        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()).asFloatBuf
    fer()
        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_TRI
    ANGLES, 0, 3); GLES30.glBindVertexArray(0) }
    fun release() { if (vboId != 0) GLES30.glDeleteBuffers(1, intArrayOf(vboId), 0); if (vaoId != 0) GLES30.glDeleteVert
    exArrays(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 {
    @Volatile
    private var glThread: Thread? = null

    fun setGlThread(thread: Thread) {
        glThread = thread
    }

    fun checkOnGlThread() {
        val current = Thread.currentThread()
        val expected = glThread
        check(current === expected) {
            "GL resource accessed outside GLSurfaceView render thread: current=$current, expected=$expected"
        }
    }

    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) {
        GlCheck.checkOnGlThread()
        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() {
        GlCheck.checkOnGlThread()
        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_BY
    TE, 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 = ha
    lfFloat
        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 vec2 iDab1;
    uniform mat4 uCanvasToClip;
    uniform vec2 uCanvasSize;
    out vec2 vLocal;
    out vec2 vCanvasUv;
    flat out float vCoverage;
    flat out float vFlow;
    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;
        vCoverage=iDab1.x;
        vFlow=iDab1.y;
        vec2 p = iDab0.xy+r*iDab0.z;
        vCanvasUv = p / uCanvasSize;
        gl_Position=uCanvasToClip*vec4(p,0.,1.);
    }
"""
const val dabFragment = """#version 300 es
    precision highp float;
    in vec2 vLocal;
    in vec2 vCanvasUv;
    flat in float vCoverage;
    flat in float vFlow;
    uniform vec3 uColorLinear;
    uniform sampler2D uGrainTex;
    uniform int uGrainActive;
    uniform float uGrainScale;
    uniform int uGrainCanvasLocked;
    uniform float uTextureDepth;
    uniform float uTextureContrast;
    uniform float uStrokeOpacity;
    uniform sampler2D uShapeTex;
    uniform int uShapeActive;
    uniform int uReverseShape;
    uniform int uRgbToAlpha;
    out vec4 fragColor;
    float luminance(vec3 color) {
        return dot(color, vec3(0.299, 0.587, 0.114));
    }
    float applyTextureLevels(float value, float contrast, float depth) {
        float adjusted = clamp((value - 0.5) * contrast + 0.5, 0.0, 1.0);
        return mix(1.0 - depth, 1.0, adjusted);
    }
    void main(){
        float r=length(vLocal);
        float aa=max(fwidth(r),.001);
        float cov=1.-smoothstep(1.-aa,1.+aa,r);
        if (cov <= 0.0) discard;

        float shapeMask = 1.0;
        if (uShapeActive == 1) {
            vec2 shapeUv = vLocal * 0.5 + 0.5;
            vec4 shapeColor = texture(uShapeTex, shapeUv);
            shapeMask = uRgbToAlpha == 1
                ? dot(shapeColor.rgb, vec3(0.299, 0.587, 0.114))
                : shapeColor.a;
            if (uReverseShape == 1) shapeMask = 1.0 - shapeMask;
        }

        float grainFactor = 1.0;
        if (uGrainActive == 1) {
            vec2 uv = uGrainCanvasLocked == 1
                ? vCanvasUv * max(uGrainScale, 0.0001)
                : (vLocal * 0.5 + 0.5) * max(uGrainScale, 0.0001);
            vec4 grainColor = texture(uGrainTex, uv);
```

```
            grainFactor = applyTextureLevels(
                luminance(grainColor.rgb), uTextureContrast, uTextureDepth
            );
        }

        float a = vCoverage * vFlow * cov * shapeMask * grainFactor * uStrokeOpacity;
        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 uStr
okeActive; 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=uAntiAliasL
evel==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;
    uniform vec2 uCanvasSize;
    out vec2 vPos;
    out vec2 vCanvasUv;
    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;
        vCanvasUv = p / uCanvasSize;
        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;
        in vec2 vCanvasUv;
        flat in vec3 vA;
        flat in vec3 vB;
        uniform vec3 uColorLinear;
        uniform sampler2D uGrainTex;
        uniform int uGrainActive;
        uniform float uGrainScale;
        uniform int uGrainCanvasLocked;
        uniform float uTextureDepth;
        uniform float uTextureContrast;
        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: [redacted] ([redacted], [redacted]) -> [redacted].
            if (l2 < 1e-6) return length(p - a) - max(r1, r2);
            // Guard 2: [redacted] [redacted] [redacted] [redacted] ([redacted] [redacted] [redacted] [redacted]).
            // [redacted] a2 < 0 -> sqrt([redacted]) -> NaN -> [redacted].
            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;   // [redacted] [redacted] [redacted] B
            if (sign(y) * a2 * y2 < k) return sqrt(x2 + y2) * il2 - r1;   // [redacted] [redacted] [redacted] A
            return (sqrt(x2 * a2 * il2) + y * rr) * il2 - r1;             // [redacted] [redacted]
        }

        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;

            float grainFactor = 1.0;
            if (uGrainActive == 1) {
                vec2 uv = vCanvasUv * max(uGrainScale, 0.0001);
                float val = texture(uGrainTex, uv).r;
                val = clamp((val - 0.5) * uTextureContrast + 0.5, 0.0, 1.0);
                grainFactor = mix(1.0 - uTextureDepth, 1.0, val);
            }

            float finalAlpha = cov * grainFactor;
            fragColor = vec4(uColorLinear * finalAlpha, finalAlpha);
        }
    """

    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: ArrayBlocking
    Queue<InputBatch>) {
    private var activePointerId = -1
    private val canvasPoint = FloatArray(2)
    private val stylusAxes = StylusAxes()
    private val tiltOut = FloatArray(2)

    var onSecondaryPointerDown: (() -> Unit)? = null

    /** [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].
            if (activePointerId >= 0) {
                onSecondaryPointerDown?.invoke()
```

```
                activePointerId = -1
                enqueueEmpty(InputAction.CANCEL)
                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: ■■■■■ ■■■■■■■■ ■■■■■■ ■■■■■■ ■■ ■■■■■■■■.
                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)
                stylusAxes.readTilt(event, index, h, tool, tiltOut)
                val orientation = stylusAxes.readOrientation(event, index, h, tool)
                if(!batch.addSample(
                        canvasPoint[0], canvasPoint[1], pressure(event, tool, index, h),
                        tiltOut[0], tiltOut[1], orientation,
                        event.getHistoricalEventTime(h) * 1_000_000L, id, tool, true
                    )) break
            }
        }
        t.screenToCanvas(event.getX(index), event.getY(index), canvasPoint)
        stylusAxes.readTilt(event, index, -1, tool, tiltOut)
        val orientation = stylusAxes.readOrientation(event, index, -1, tool)
        batch.addSample(
            canvasPoint[0], canvasPoint[1], pressure(event, tool, index, -1),
            tiltOut[0], tiltOut[1], orientation,
            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;Motion
    Event.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
    ) SYNTHETIC_FINGER_PRESSURE else (if(h>=0)e.getHistoricalPressure(i,h) else e.getPressure(i)).coerceIn(0f,1f)

    companion object {
        private const val SYNTHETIC_FINGER_PRESSURE = 0.62f
    }
}
```

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

```
package com.wetinknext.engine.input

import android.view.MotionEvent
import kotlin.math.cos
import kotlin.math.sin

class StylusAxes {

    fun readTilt(
        event: MotionEvent,
        pointerIndex: Int,
        historicalIndex: Int,
        tool: PointerTool,
        out: FloatArray,
    ) {
        if (tool == PointerTool.FINGER ||
            tool == PointerTool.UNKNOWN
        ) {
            out[0] = 0f
            out[1] = 0f
            return
        }

        val tilt = if (historicalIndex >= 0) {
```

```
            event.getHistoricalAxisValue(
                MotionEvent.AXIS_TILT,
                pointerIndex,
                historicalIndex,
            )
        } else {
            event.getAxisValue(
                MotionEvent.AXIS_TILT,
                pointerIndex,
            )
        }

        val orientation = if (historicalIndex >= 0) {
            event.getHistoricalAxisValue(
                MotionEvent.AXIS_ORIENTATION,
                pointerIndex,
                historicalIndex,
            )
        } else {
            event.getAxisValue(
                MotionEvent.AXIS_ORIENTATION,
                pointerIndex,
            )
        }

        val normalizedTilt =
            (tilt / (Math.PI.toFloat() * 0.5f))
                .coerceIn(0f, 1f)

        out[0] =
            (cos(orientation) * normalizedTilt)
                .coerceIn(-1f, 1f)

        out[1] =
            (sin(orientation) * normalizedTilt)
                .coerceIn(-1f, 1f)
    }

    fun readOrientation(
        event: MotionEvent,
        pointerIndex: Int,
        historicalIndex: Int,
        tool: PointerTool,
    ): Float {
        if (tool == PointerTool.FINGER || tool == PointerTool.UNKNOWN) return 0f
        return if (historicalIndex >= 0) {
            event.getHistoricalAxisValue(MotionEvent.AXIS_ORIENTATION, pointerIndex, historicalIndex)
        } else {
            event.getAxisValue(MotionEvent.AXIS_ORIENTATION, pointerIndex)
        }
    }
}
```

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

```
package com.wetinknext.engine.undo

import java.io.ByteArrayOutputStream
import java.util.zip.Deflater
import java.util.zip.Inflater

class DeflateTileCompressor(
    private val level: Int = Deflater.BEST_SPEED,
) : TileCompressor {

    override fun compress(data: ByteArray): ByteArray {
        val deflater = Deflater(level)
        return try {
            deflater.setInput(data)
            deflater.finish()

            val output = ByteArrayOutputStream(data.size)
            val buffer = ByteArray(16 * 1024)

            while (!deflater.finished()) {
                val count = deflater.deflate(buffer)
                output.write(buffer, 0, count)
            }

            output.toByteArray()
        } finally {
            deflater.end()
        }
    }

    override fun decompress(
        data: ByteArray,
```

```
        expectedSize: Int,
    ): ByteArray {
        val inflater = Inflater()
        return try {
            inflater.setInput(data)

            val output = ByteArrayOutputStream(expectedSize)
            val buffer = ByteArray(16 * 1024)

            while (!inflater.finished()) {
                val count = inflater.inflate(buffer)
                if (count == 0 && inflater.finished()) break
                check(count > 0) { "Decompression stalled or data corrupted" }
                output.write(buffer, 0, count)
            }

            output.toByteArray()
        } finally {
            inflater.end()
        }
    }
}
```

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

```
package com.wetinknext.engine.undo

/** Uncompressed tile data captured on the GL thread. */
class RawTileSnapshot(
    val coord: TileCoord,
    val pixelLeft: Int,
    val pixelTop: Int,
    val pixelWidth: Int,
    val pixelHeight: Int,
    val bytesPerPixel: Int,
    val rawBytes: ByteArray,
) {
    val rawSize: Int get() = rawBytes.size

    fun compress(compressor: TileCompressor): TileSnapshot {
        return TileSnapshot(
            coord = coord,
            pixelLeft = pixelLeft,
            pixelTop = pixelTop,
            pixelWidth = pixelWidth,
            pixelHeight = pixelHeight,
            bytesPerPixel = bytesPerPixel,
            storedData = compressor.compress(rawBytes),
            compressor = compressor,
        )
    }
}
```

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. */
    fun decompress(): ByteArray = compressor.decompress(storedData, rawSize)

    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<RawTileSnapshot> {
        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<RawTileSnapshot>()
        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 += RawTileSnapshot(
                    coord = TileCoord(tileX, tileY),
                    pixelLeft = pixelLeft,
                    pixelTop = pixelTop,
                    pixelWidth = pixelWidth,
                    pixelHeight = pixelHeight,
                    bytesPerPixel = bytesPerPixel,
                    rawBytes = rawBytes,
                )
            }
        }
        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.decompress()
            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()
    }

    /** Returns the candidate without changing history; restore it before committing. */
    fun peekUndo(): UndoEntry? = undoStack.lastOrNull()

    /** Moves the exact current undo candidate to redo after a successful restore. */
    fun commitUndo(entry: UndoEntry): Boolean {
        if (undoStack.lastOrNull() !== entry) return false
        undoStack.removeLast()
        undoMemoryBytes -= entry.memorySize
        redoStack.addLast(entry)
        redoMemoryBytes += entry.memorySize
        trimToLimits()
        return true
    }

    /** Returns the candidate without changing history; restore it before committing. */
    fun peekRedo(): UndoEntry? = redoStack.lastOrNull()

    /** Moves the exact current redo candidate to undo after a successful restore. */
    fun commitRedo(entry: UndoEntry): Boolean {
        if (redoStack.lastOrNull() !== entry) return false
        redoStack.removeLast()
        redoMemoryBytes -= entry.memorySize
        undoStack.addLast(entry)
        undoMemoryBytes += entry.memorySize
        trimToLimits()
        return true
    }

    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/animation/AnimationSettingsMenu.kt

```
package com.wetinknext.ui.animation

import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.foundation.verticalScroll
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material3.*
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.toArgb
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.wetinknext.domain.animation.AnimationDocument
import com.wetinknext.domain.animation.AnimationPlaybackMode
import com.wetinknext.domain.animation.OnionSkinSettings
import com.wetinknext.ui.theme.AppTheme

@Composable
fun AnimationSettingsMenu(
    theme: AppTheme,
    document: AnimationDocument,
    onDocumentChange: (AnimationDocument) -> Unit,
    onDismiss: () -> Unit,
    modifier: Modifier = Modifier
) {
    Box(
        modifier = modifier
            .width(280.dp)
            .heightIn(max = 360.dp)
            .clip(RoundedCornerShape(24.dp))
            .background(theme.panelBg.copy(alpha = 0.98f))
            .border(1.dp, theme.panelStroke, RoundedCornerShape(24.dp))
            .padding(16.dp)
    ) {
        Column(
            modifier = Modifier.verticalScroll(rememberScrollState()),
            verticalArrangement = Arrangement.spacedBy(10.dp),
        ) {
            Text("■■■■■■■■■■ ■■■■■■■■■", color = theme.textPrimary, fontSize = 16.sp, fontWeight = FontWeight.Bold)

            // Playback Mode
            Column(verticalArrangement = Arrangement.spacedBy(8.dp)) {
                Text("■■■■■■", color = theme.textSecondary, fontSize = 12.sp)
                Row(
                    modifier = Modifier.fillMaxWidth(),
                    horizontalArrangement = Arrangement.spacedBy(8.dp)
                ) {
                    PlaybackModeButton("■■■■", AnimationPlaybackMode.LOOP, document.playbackMode, theme) {
                        onDocumentChange(document.copy(playbackMode = it))
                    }
                    PlaybackModeButton("■■■■-■■■■", AnimationPlaybackMode.PING_PONG, document.playbackMode, theme) {
                        onDocumentChange(document.copy(playbackMode = it))
                    }
                    PlaybackModeButton("■■■■ ■■■", AnimationPlaybackMode.ONE_SHOT, document.playbackMode, theme) {
                        onDocumentChange(document.copy(playbackMode = it))
                    }
                }
            }

            // FPS
            Column(verticalArrangement = Arrangement.spacedBy(4.dp)) {
                Row(horizontalArrangement = Arrangement.SpaceBetween, modifier = Modifier.fillMaxWidth()) {
                    Text("■■■■■■■■■ (FPS)", color = theme.textSecondary, fontSize = 12.sp)
```

```
                Text("${document.framesPerSecond}", color = theme.accent, fontSize = 12.sp, fontWeight = FontWeight.
    Bold)
            }
            Slider(
                value = document.framesPerSecond.toFloat(),
                onValueChange = { onDocumentChange(document.copy(framesPerSecond = it.toInt())) },
                valueRange = 1f..60f,
                colors = SliderDefaults.colors(thumbColor = theme.accent, activeTrackColor = theme.accent)
            )
        }

        HorizontalDivider(color = theme.panelStroke)

        // Onion Skin Settings
        Row(
            modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.SpaceBetween,
            verticalAlignment = Alignment.CenterVertically
        ) {
            Text("[redacted] (Onion Skin)", color = theme.textPrimary, fontSize = 14.sp)
            Switch(
                checked = document.onionSkin.enabled,
                onCheckedChange = { onDocumentChange(document.copy(onionSkin = document.onionSkin.copy(enabled = it)
    )) },
                colors = SwitchDefaults.colors(checkedThumbColor = theme.accent, checkedTrackColor = theme.accent.co
    py(alpha = 0.5f))
            )
        }

        if (document.onionSkin.enabled) {
            OnionCountSlider("[redacted]", document.onionSkin.previousFrames, theme) {
                onDocumentChange(document.copy(onionSkin = document.onionSkin.copy(previousFrames = it)))
            }
            OnionColorPicker(
                label = "[redacted] [redacted]",
                selectedArgb = document.onionSkin.previousColorArgb,
                theme = theme,
            ) { color ->
                onDocumentChange(document.copy(onionSkin = document.onionSkin.copy(previousColorArgb = color.toArgb(
    ))))
            }
            OnionCountSlider("[redacted]", document.onionSkin.nextFrames, theme) {
                onDocumentChange(document.copy(onionSkin = document.onionSkin.copy(nextFrames = it)))
            }
            OnionColorPicker(
                label = "[redacted] [redacted]",
                selectedArgb = document.onionSkin.nextColorArgb,
                theme = theme,
            ) { color ->
                onDocumentChange(document.copy(onionSkin = document.onionSkin.copy(nextColorArgb = color.toArgb())))
            }

            Column(verticalArrangement = Arrangement.spacedBy(4.dp)) {
                Text("[redacted]", color = theme.textSecondary, fontSize = 12.sp)
                Slider(
                    value = document.onionSkin.opacity,
                    onValueChange = { onDocumentChange(document.copy(onionSkin = document.onionSkin.copy(opacity = i
    t))) },
                    valueRange = 0f..1f,
                    colors = SliderDefaults.colors(thumbColor = theme.accent, activeTrackColor = theme.accent)
                )
            }
        }
    }
    }
}

@Composable
private fun RowScope.PlaybackModeButton(
    label: String,
    mode: AnimationPlaybackMode,
    current: AnimationPlaybackMode,
    theme: AppTheme,
    onClick: (AnimationPlaybackMode) -> Unit
) {
    val selected = mode == current
    Box(
        modifier = Modifier
            .weight(1f)
            .height(32.dp)
            .clip(RoundedCornerShape(8.dp))
            .background(if (selected) theme.accent else theme.panelInsetSoft)
            .clickable { onClick(mode) },
        contentAlignment = Alignment.Center
    ) {
        Text(label, color = if (selected) Color.White else theme.textSecondary, fontSize = 10.sp)
    }
}

@Composable
private fun OnionCountSlider(
    label: String,
    value: Int,
```

```
    theme: AppTheme,
    onValueChange: (Int) -> Unit
) {
    Column(verticalArrangement = Arrangement.spacedBy(2.dp)) {
        Row(horizontalArrangement = Arrangement.SpaceBetween, modifier = Modifier.fillMaxWidth()) {
            Text(label, color = theme.textSecondary, fontSize = 11.sp)
            Text("$value", color = theme.textPrimary, fontSize = 11.sp)
        }
        Slider(
            value = value.toFloat(),
            onValueChange = { onValueChange(it.toInt()) },
            valueRange = 0f..12f,
            colors = SliderDefaults.colors(thumbColor = theme.accent, activeTrackColor = theme.accent)
        )
    }
}

@Composable
private fun OnionColorPicker(
    label: String,
    selectedArgb: Int,
    theme: AppTheme,
    onColorChange: (Color) -> Unit,
) {
    val palette = listOf(
        Color(0xFFFF4FA3),
        Color(0xFFFF7A59),
        Color(0xFFFFC857),
        Color(0xFF57D68D),
        Color(0xFF4A8DFF),
        Color(0xFFB079FF),
    )
    Column(verticalArrangement = Arrangement.spacedBy(6.dp)) {
        Text(label, color = theme.textSecondary, fontSize = 12.sp)
        Row(horizontalArrangement = Arrangement.spacedBy(8.dp)) {
            palette.forEach { color ->
                val selected = color.toArgb() == selectedArgb
                Box(
                    modifier = Modifier
                        .size(24.dp)
                        .clip(CircleShape)
                        .background(color)
                        .border(
                            width = if (selected) 2.dp else 1.dp,
                            color = if (selected) Color.White else theme.panelStroke,
                            shape = CircleShape,
                        )
                        .clickable { onColorChange(color) }
                )
            }
        }
    }
}
```

app/src/main/java/com/wetinknext/ui/animation/AnimationTimelineToolbar.kt

```
package com.wetinknext.ui.animation

import androidx.compose.foundation.Image
import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.clickable
import androidx.compose.foundation.gestures.detectDragGesturesAfterLongPress
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.lazy.LazyRow
import androidx.compose.foundation.lazy.itemsIndexed
import androidx.compose.foundation.lazy.rememberLazyListState
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.*
import androidx.compose.material3.*
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.ImageBitmap
import androidx.compose.ui.graphics.graphicsLayer
import androidx.compose.ui.graphics.vector.ImageVector
import androidx.compose.ui.input.pointer.pointerInput
import androidx.compose.ui.layout.ContentScale
import androidx.compose.ui.platform.LocalDensity
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.IntOffset
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
```

```
import androidx.compose.ui.window.Popup
import com.wetinknext.domain.animation.AnimationDocument
import com.wetinknext.domain.animation.AnimationFrame
import com.wetinknext.domain.animation.AnimationFrameRole
import com.wetinknext.ui.theme.AppTheme
import kotlin.math.roundToInt

data class FrameThumbnailLayer(
    val image: ImageBitmap,
    val opacity: Float,
)

@Composable
fun AnimationTimelineToolbar(
    theme: AppTheme,
    document: AnimationDocument,
    currentFrameId: Long,
    frameThumbnails: Map<Long, List<FrameThumbnailLayer>> = emptyMap(),
    onFrameSelect: (Long) -> Unit,
    onFrameMove: (frameId: Long, direction: Int) -> Unit,
    onAddFrame: () -> Unit,
    onDuplicateFrame: (Long) -> Unit,
    onDeleteFrame: (Long) -> Unit,
    onHoldChange: (Long, Int) -> Unit,
    onRoleToggle: (Long, AnimationFrameRole) -> Unit,
    onAssembleLayers: () -> Unit,
    onLayersClick: () -> Unit,
    onClose: () -> Unit,
    onSettingsClick: () -> Unit = {},
    isPlaying: Boolean = false,
    onPlayToggle: () -> Unit = {},
    modifier: Modifier = Modifier
) {
    val listState = rememberLazyListState()
    val frames = document.frames
    var showContextMenuForId by remember { mutableStateOf<Long?>(null) }

    // Auto-scroll to current frame
    LaunchedEffect(currentFrameId) {
        val index = frames.indexOfFirst { it.id == currentFrameId }
        if (index >= 0) {
            listState.animateScrollToItem(index)
        }
    }

    Column(
        modifier = modifier
            .fillMaxWidth()
            .padding(bottom = 16.dp),
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Row(
            modifier = Modifier
                .widthIn(min = 400.dp, max = 820.dp)
                .height(86.dp)
                .clip(RoundedCornerShape(28.dp))
                .background(theme.panelBg.copy(alpha = 0.94f))
                .border(1.2.dp, theme.panelStroke, RoundedCornerShape(28.dp))
                .padding(horizontal = 14.dp, vertical = 8.dp),
            verticalAlignment = Alignment.CenterVertically
        ) {
            // Play toggle
            Column(
                horizontalAlignment = Alignment.CenterHorizontally,
                modifier = Modifier.padding(end = 4.dp)
            ) {
                Box(
                    modifier = Modifier
                        .size(44.dp)
                        .clip(CircleShape)
                        .background(if (isPlaying) theme.accent.copy(alpha = 0.4f) else theme.accent)
                        .clickable { onPlayToggle() },
                    contentAlignment = Alignment.Center
                ) {
                    Icon(
                        if (isPlaying) Icons.Default.Pause else Icons.Default.PlayArrow,
                        null,
                        tint = Color.White
                    )
                }
                Text(if (isPlaying) "■■■■■" else "■■■■", color = theme.textSecondary, fontSize = 9.sp)
            }

            Spacer(Modifier.width(10.dp))
            Box(Modifier.width(1.dp).fillMaxHeight(0.6f).background(theme.panelStroke))
            Spacer(Modifier.width(10.dp))

            // Frames timeline
            LazyRow(
                state = listState,
                modifier = Modifier.weight(1f),
                horizontalArrangement = Arrangement.spacedBy(8.dp),
                verticalAlignment = Alignment.CenterVertically
```

```
            ) {
                itemsIndexed(frames, key = { _, f -> f.id }) { _, frame ->
                    TimelineFrameItem(
                        frame = frame,
                        isSelected = frame.id == currentFrameId,
                        thumbnailLayers = frameThumbnails[frame.id].orEmpty(),
                        theme = theme,
                        canMoveLeft = frames.indexOf(frame) > 0,
                        canMoveRight = frames.indexOf(frame) < frames.lastIndex,
                        onMove = { direction -> onFrameMove(frame.id, direction) },
                        onClick = {
                            if (currentFrameId == frame.id) {
                                showContextMenuForId = frame.id
                            } else {
                                onFrameSelect(frame.id)
                            }
                        }
                    )
                }
            }

            Spacer(Modifier.width(10.dp))
            Box(Modifier.width(1.dp).fillMaxHeight(0.6f).background(theme.panelStroke))
            Spacer(Modifier.width(10.dp))

            // Actions
            Row(
                verticalAlignment = Alignment.CenterVertically,
                horizontalArrangement = Arrangement.spacedBy(12.dp)
            ) {
                TimelineActionIcon(Icons.Default.Add, "[redacted]", theme, onClick = onAddFrame)
                TimelineActionIcon(Icons.Default.Layers, "[redacted]", theme, onClick = onAssembleLayers)
                TimelineActionIcon(Icons.Default.ContentCopy, "[redacted]", theme, onClick = { onDuplicateFrame(currentFrameI
    d) })
                TimelineActionIcon(Icons.Default.Delete, "[redacted]", theme, enabled = frames.size > 1, onClick = { onDele
    teFrame(currentFrameId) })
                TimelineActionIcon(Icons.Default.Settings, "[redacted].", theme, onClick = onSettingsClick)
                TimelineActionIcon(Icons.Default.Layers, "[redacted]", theme, onClick = onLayersClick)
                TimelineActionIcon(Icons.Default.Close, "[redacted]", theme, onClick = onClose)
            }
        }

        // Context Menu
        showContextMenuForId?.let { frameId ->
            val frame = frames.find { it.id == frameId }
            if (frame != null) {
                Popup(
                    alignment = Alignment.TopCenter,
                    offset = IntOffset(0, -220),
                    onDismissRequest = { showContextMenuForId = null }
                ) {
                    FrameContextMenu(
                        frame = frame,
                        theme = theme,
                        onDuplicate = { onDuplicateFrame(frameId); showContextMenuForId = null },
                        onDelete = { onDeleteFrame(frameId); showContextMenuForId = null },
                        onHoldChange = { onHoldChange(frameId, it) },
                        onRoleToggle = { onRoleToggle(frameId, it); showContextMenuForId = null },
                        canDelete = frames.size > 1
                    )
                }
            }
        }
    }
}

@Composable
private fun FrameContextMenu(
    frame: AnimationFrame,
    theme: AppTheme,
    onDuplicate: () -> Unit,
    onDelete: () -> Unit,
    onHoldChange: (Int) -> Unit,
    onRoleToggle: (AnimationFrameRole) -> Unit,
    canDelete: Boolean
) {
    Surface(
        modifier = Modifier
            .width(220.dp)
            .clip(RoundedCornerShape(16.dp))
            .border(1.dp, theme.panelStroke, RoundedCornerShape(16.dp)),
        color = theme.panelBg,
        tonalElevation = 8.dp
    ) {
        Column(modifier = Modifier.padding(12.dp)) {
            ContextMenuItem("[redacted]", Icons.Default.ContentCopy, theme, onClick = onDuplicate)
            ContextMenuItem("[redacted]", Icons.Default.Delete, theme, enabled = canDelete, onClick = onDelete)

            HorizontalDivider(modifier = Modifier.padding(vertical = 8.dp), color = theme.panelStroke)

            Text("[redacted]: ${frame.holdFrames}", color = theme.textSecondary, fontSize = 11.sp, modifier = Modifier.pad
    ding(horizontal = 8.dp))
            Slider(
```

```
                value = frame.holdFrames.toFloat(),
                onValueChange = { onHoldChange(it.toInt()) },
                valueRange = 1f..120f,
                colors = SliderDefaults.colors(thumbColor = theme.accent, activeTrackColor = theme.accent)
            )

            HorizontalDivider(modifier = Modifier.padding(vertical = 8.dp), color = theme.panelStroke)

            val isBg = frame.role == AnimationFrameRole.BACKGROUND
            val isFg = frame.role == AnimationFrameRole.FOREGROUND

            Row(verticalAlignment = Alignment.CenterVertically, modifier = Modifier.fillMaxWidth().clickable { onRoleTog
    gle(if (isBg) AnimationFrameRole.NORMAL else AnimationFrameRole.BACKGROUND) }.padding(8.dp)) {
                Checkbox(checked = isBg, onCheckedChange = null, colors = CheckboxDefaults.colors(checkedColor = theme.a
    ccent))
                Spacer(Modifier.width(8.dp))
                Text("[redacted]", color = theme.textPrimary, fontSize = 13.sp)
            }
            Row(verticalAlignment = Alignment.CenterVertically, modifier = Modifier.fillMaxWidth().clickable { onRoleTog
    gle(if (isFg) AnimationFrameRole.NORMAL else AnimationFrameRole.FOREGROUND) }.padding(8.dp)) {
                Checkbox(checked = isFg, onCheckedChange = null, colors = CheckboxDefaults.colors(checkedColor = theme.a
    ccent))
                Spacer(Modifier.width(8.dp))
                Text("[redacted]", color = theme.textPrimary, fontSize = 13.sp)
            }
        }
    }
}

@Composable
private fun ContextMenuItem(
    label: String,
    icon: ImageVector,
    theme: AppTheme,
    enabled: Boolean = true,
    onClick: () -> Unit
) {
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .clickable(enabled = enabled) { onClick() }
            .padding(8.dp)
            .alphaIfDisabled(!enabled),
        verticalAlignment = Alignment.CenterVertically
    ) {
        Icon(icon, null, tint = if (enabled) theme.textPrimary else theme.textSecondary.copy(alpha = 0.5f), modifier = M
    odifier.size(18.dp))
        Spacer(Modifier.width(12.dp))
        Text(label, color = if (enabled) theme.textPrimary else theme.textSecondary.copy(alpha = 0.5f), fontSize = 14.sp
    )
    }
}

@Composable
private fun TimelineFrameItem(
    frame: AnimationFrame,
    isSelected: Boolean,
    thumbnailLayers: List<FrameThumbnailLayer>,
    theme: AppTheme,
    canMoveLeft: Boolean,
    canMoveRight: Boolean,
    onMove: (Int) -> Unit,
    onClick: () -> Unit
) {
    val moveThresholdPx = with(LocalDensity.current) { 44.dp.toPx() }
    var accumulatedDragX by remember(frame.id) { mutableFloatStateOf(0f) }
    Column(
        horizontalAlignment = Alignment.CenterHorizontally,
        modifier = Modifier
            .pointerInput(frame.id, canMoveLeft, canMoveRight) {
                detectDragGesturesAfterLongPress(
                    onDragStart = { accumulatedDragX = 0f },
                    onDrag = { change, dragAmount ->
                        accumulatedDragX += dragAmount.x
                        val direction = when {
                            accumulatedDragX <= -moveThresholdPx && canMoveLeft -> -1
                            accumulatedDragX >= moveThresholdPx && canMoveRight -> 1
                            else -> 0
                        }
                        if (direction != 0) {
                            change.consume()
                            onMove(direction)
                            accumulatedDragX = 0f
                        }
                    }
                )
            }
            .clickable { onClick() }
    ) {
        Box(
            modifier = Modifier
                .size(54.dp, 40.dp)
                .clip(RoundedCornerShape(4.dp))
```

```
                .background(if (isSelected) theme.accent.copy(alpha = 0.2f) else theme.panelInsetSoft)
                .border(
                    width = if (isSelected) 2.dp else 1.dp,
                    color = if (isSelected) theme.accent else theme.panelStroke,
                    shape = RoundedCornerShape(4.dp)
                ),
            contentAlignment = Alignment.Center
        ) {
            if (thumbnailLayers.isNotEmpty()) {
                thumbnailLayers.forEach { layer ->
                    Image(
                        bitmap = layer.image,
                        contentDescription = null,
                        contentScale = ContentScale.Fit,
                        modifier = Modifier
                            .fillMaxSize()
                            .padding(2.dp)
                            .graphicsLayer(alpha = layer.opacity.coerceIn(0f, 1f)),
                    )
                }
            } else {
                Icon(Icons.Default.Image, null, tint = theme.textSecondary.copy(alpha = 0.3f), modifier = Modifier.size(
    20.dp))
            }
        }
        if (frame.holdFrames > 1) {
            Text("+${frame.holdFrames - 1}", color = theme.accent, fontSize = 9.sp, fontWeight = FontWeight.Bold)
        } else {
            Spacer(Modifier.height(11.dp))
        }
    }
}

@Composable
private fun TimelineActionIcon(
    icon: ImageVector,
    label: String,
    theme: AppTheme,
    enabled: Boolean = true,
    onClick: () -> Unit
) {
    Column(
        horizontalAlignment = Alignment.CenterHorizontally,
        modifier = Modifier
            .alphaIfDisabled(!enabled)
            .clickable(enabled = enabled) { onClick() }
    ) {
        Icon(icon, null, tint = if (enabled) theme.textPrimary else theme.textSecondary.copy(alpha = 0.5f), modifier = M
    odifier.size(22.dp))
        Text(label, color = theme.textSecondary, fontSize = 9.sp)
    }
}

private fun Modifier.alphaIfDisabled(disabled: Boolean): Modifier = if (disabled) this.then(Modifier.graphicsLayer { alp
    ha = 0.5f }) else this
```

app/src/main/java/com/wetinknext/ui/color/ColorCollection.kt

```
package com.wetinknext.ui.color

import androidx.compose.ui.graphics.Color

data class ColorCollection(
    val id: Long,
    val name: String,
    val colors: List<Int>,
    val builtin: Boolean = false,
)

fun ColorCollection.composeColors(): List<Color> = colors.map { Color(it) }

private fun hex(s: String): Int {
    val v = s.removePrefix("#").trim()
    val r = v.substring(0, 2).toInt(16)
    val g = v.substring(2, 4).toInt(16)
    val b = v.substring(4, 6).toInt(16)
    return (0xFF shl 24) or (r shl 16) or (g shl 8) or b
}

private fun palette(vararg hex: String): List<Int> = hex.map { hex(it) }

object BuiltinCollections {
    val all: List<ColorCollection> = listOf(
        ColorCollection(
            id = -1L,
            name = "[redacted]",
            colors = palette(
                "#FFF3B0", "#FFD6A5", "#FFC2B0", "#FFCCD5", "#F1C6F2",
                "#D8C7F5", "#C5D7FF", "#B8E5FF", "#B6F0EE", "#C9F2D8",
                "#FFE066", "#FFB87A", "#FF9F8B", "#FF9CB0", "#E59FE6",
```

```
                "#B89BE8", "#9CB6F5", "#86CFF2", "#7FD9D5", "#8AD9A8",
                "#F7C548", "#F09870", "#E07A6E", "#D9748E", "#C383C6",
                "#9678D0", "#7798E0", "#5DB1E6", "#5CB8B5", "#65BD8A",
            ),
            builtin = true,
        ),
        ColorCollection(
            id = -2L,
            name = "[redacted]",
            colors = palette(
                "#F9E79F", "#F7D08A", "#F5B187", "#F0998A", "#EC9090",
                "#E68F9C", "#D88BAE", "#C58CC0", "#B391CD", "#9C95D6",
                "#A8D5B6", "#9FD2C9", "#9CCFD8", "#A0C9E0", "#A8C0E3",
                "#B5BBE3", "#C0B8DE", "#CCB5D3", "#D6B3C5", "#DBB3B6",
                "#C7E1A1", "#B7DAA3", "#A8D2AE", "#9CCABD", "#94C0CB",
                "#94B5D2", "#9DA9D1", "#B0A0CB", "#C19BBE", "#CC9AAB",
            ),
            builtin = true,
        ),
        ColorCollection(
            id = -3L,
            name = "[redacted]",
            colors = palette(
                "#F4C9B5", "#F0D4BC", "#E8B89A", "#D89C7E", "#C0805F",
                "#9C5F3F", "#74442B", "#4F2C1C", "#311A11", "#1B0F0A",
                "#FBE3D4", "#F8DCC9", "#F2C5A5", "#E2A77F", "#C58557",
                "#A36739", "#7A4824", "#552E14", "#321A0A", "#FFFAF0",
                "#E8C7C1", "#D49C9C", "#B97070", "#8C4F4F", "#5C3030",
                "#C7BFB1", "#9C9484", "#6E6757", "#A0B0C4", "#5A6878",
            ),
            builtin = true,
        ),
        ColorCollection(
            id = -4L,
            name = "[redacted]",
            colors = palette(
                "#AEDCF0", "#7FBFE1", "#56A0CB", "#3D80B0", "#2E608E",
                "#244670", "#1B335A", "#142545", "#0D1830", "#070D1E",
                "#CFE3B0", "#A8CF8C", "#7FB36A", "#5C9651", "#3F7A3F",
                "#2E5E33", "#214728", "#16321C", "#5C7B4A", "#3F5732",
                "#F4D58D", "#E3B26E", "#D08A4A", "#B5642E", "#923F1C",
                "#6E2A14", "#8B5A2B", "#5C3A1A", "#A38560", "#6F5B3D",
            ),
            builtin = true,
        ),
        ColorCollection(
            id = -5L,
            name = "[redacted]",
            colors = palette(
                "#111318", "#3B414A", "#7B8491", "#B8C0CC", "#FFFFFF",
                "#4A90E2", "#35B9D5", "#42B883", "#A5D646", "#F4D44D",
                "#F5A23D", "#F26B4F", "#E6536F", "#E66FAD", "#A36BDB",
                "#6D8CFF", "#64D4E8", "#76D7A6", "#D8EA72", "#FFE58A",
                "#245EA8", "#167E98", "#20794F", "#6B8F20", "#B58B08",
                "#B65A12", "#B53832", "#A62E58", "#A23E86", "#633B9B",
            ),
            builtin = true,
        ),
    )
}
```

app/src/main/java/com/wetinknext/ui/color/ColorMath.kt

```
package com.wetinknext.ui.color

import androidx.compose.ui.graphics.Color
import kotlin.math.abs

data class Hsl(val h: Float, val s: Float, val l: Float)

fun rgbToHsl(c: Color): Hsl {
    val r = c.red
    val g = c.green
    val b = c.blue
    val max = maxOf(r, g, b)
    val min = minOf(r, g, b)
    val l = (max + min) / 2f
    val d = max - min
    if (d == 0f) return Hsl(0f, 0f, l)
    val s = if (l > 0.5f) d / (2f - max - min) else d / (max + min)
    val h = when (max) {
        r -> 60f * (((g - b) / d) % 6f)
        g -> 60f * ((b - r) / d + 2f)
        else -> 60f * ((r - g) / d + 4f)
    }
    return Hsl(((h % 360f) + 360f) % 360f, s, l)
}

fun hslToColor(h: Float, s: Float, l: Float): Color {
    val hh = ((h % 360f) + 360f) % 360f
```

```
    val c = (1f - abs(2f * l - 1f)) * s
    val x = c * (1f - abs((hh / 60f) % 2f - 1f))
    val m = l - c / 2f
    val (r1, g1, b1) = when ((hh / 60f).toInt()) {
        0 -> Triple(c, x, 0f)
        1 -> Triple(x, c, 0f)
        2 -> Triple(0f, c, x)
        3 -> Triple(0f, x, c)
        4 -> Triple(x, 0f, c)
        else -> Triple(c, 0f, x)
    }
    return Color(
        (r1 + m).coerceIn(0f, 1f),
        (g1 + m).coerceIn(0f, 1f),
        (b1 + m).coerceIn(0f, 1f),
        1f,
    )
}

fun hslToColor(hsl: Hsl): Color = hslToColor(hsl.h, hsl.s, hsl.l)

fun Color.toHex6(): String {
    val r = (red * 255f).toInt().coerceIn(0, 255)
    val g = (green * 255f).toInt().coerceIn(0, 255)
    val b = (blue * 255f).toInt().coerceIn(0, 255)
    return "#%02X%02X%02X".format(r, g, b)
}

enum class HarmonyType(
    val displayName: String,
    val offsets: FloatArray,
) {
    Complementary("■■■■■■■■■■■■■■■■", floatArrayOf(0f, 180f)),
    Analogous("■■■■■■■■■■■", floatArrayOf(-30f, 0f, 30f)),
    Triadic("■■■■■■■■■", floatArrayOf(0f, 120f, 240f)),
    SplitComplementary(
        "■■■■■■-■■■■■■■■■■■■■■■■",
        floatArrayOf(0f, 150f, 210f),
    ),
    Tetradic("■■■■■■■■■■", floatArrayOf(0f, 60f, 180f, 240f)),
    Square("■■■■■■■■■■", floatArrayOf(0f, 90f, 180f, 270f)),
    ;

    fun derive(base: Hsl): List<Hsl> {
        val h = base.h
        return offsets.map { off ->
            Hsl(((h + off) % 360f + 360f) % 360f, base.s, base.l)
        }
    }
}
```

app/src/main/java/com/wetinknext/ui/color/ColorPanel.kt

```
package com.wetinknext.ui.color

import androidx.compose.animation.core.animateFloatAsState
import androidx.compose.animation.core.tween
import androidx.compose.foundation.Canvas
import androidx.compose.foundation.ExperimentalFoundationApi
import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.clickable
import androidx.compose.foundation.combinedClickable
import androidx.compose.foundation.gestures.detectDragGestures
import androidx.compose.foundation.gestures.detectHorizontalDragGestures
import androidx.compose.foundation.gestures.detectTapGestures
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.foundation.text.BasicTextField
import androidx.compose.foundation.verticalScroll
import androidx.compose.material3.*
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.draw.shadow
import androidx.compose.ui.geometry.Offset
import androidx.compose.ui.geometry.Size
import androidx.compose.ui.graphics.Brush
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.SolidColor
import androidx.compose.ui.graphics.drawscope.DrawScope
import androidx.compose.ui.graphics.drawscope.Stroke
import androidx.compose.ui.input.pointer.pointerInput
import androidx.compose.ui.platform.LocalDensity
import androidx.compose.ui.text.TextStyle
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.style.TextAlign
```

```
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.IntOffset
import androidx.compose.ui.unit.sp
import androidx.compose.ui.window.Dialog
import com.wetinknext.ui.components.PanelSurface
import com.wetinknext.ui.theme.AppTheme
import kotlinx.coroutines.delay
import kotlin.math.atan2
import kotlin.math.cos
import kotlin.math.hypot
import kotlin.math.sin
import kotlin.math.roundToInt

private enum class ColorTab { Wheel, Map }

@Composable
fun ColorPanel(
    state: GlesColorState,
    color: Color,
    theme: AppTheme,
    onColorChange: (Color) -> Unit,
    onAddFromPhoto: () -> Unit,
    title: String = "■■■■■",
    recordRecent: Boolean = true,
    allowCollectionEditing: Boolean = true,
    onPinCollection: (ColorCollection) -> Unit = {},
    modifier: Modifier = Modifier,
    onDismiss: () -> Unit,
) {
    var tab by remember { mutableStateOf(ColorTab.Wheel) }
    var hsl by remember { mutableStateOf(rgbToHsl(color)) }
    var harmonyType by remember { mutableStateOf(HarmonyType.Triadic) }
    val latestHsl by rememberUpdatedState(hsl)

    val wheelScrollState = rememberScrollState()
    val mapScrollState = rememberScrollState()
    val tabScrollState = when (tab) {
        ColorTab.Wheel -> wheelScrollState
        ColorTab.Map -> mapScrollState
    }

    LaunchedEffect(hsl, recordRecent) {
        val selectedColor = hslToColor(hsl)
        onColorChange(selectedColor)
        if (recordRecent) {
            delay(350)
            state.pushRecentColor(selectedColor)
        }
    }

    val dismissAndSave = {
        if (recordRecent) state.pushRecentColor(hslToColor(hsl))
        onDismiss()
    }

    DisposableEffect(Unit) {
        onDispose {
            if (recordRecent) state.pushRecentColor(hslToColor(latestHsl))
        }
    }

    PanelSurface(
        modifier = modifier.width(320.dp),
    ) {
        Column(
            modifier = Modifier.padding(14.dp),
        ) {
            ColorHeader(
                preview = hslToColor(hsl),
                title = title,
                theme = theme,
                onConfirm = dismissAndSave,
            )
            Spacer(Modifier.height(10.dp))
            TabBar(tab, theme) { tab = it }
            Spacer(Modifier.height(12.dp))
            Box(
                modifier = Modifier
                    .height(TabContentHeight)
                    .fillMaxWidth(),
            ) {
                Column(
                    modifier = Modifier
                        .fillMaxSize()
                        .verticalScroll(tabScrollState),
                ) {
                    when (tab) {
                        ColorTab.Wheel -> WheelTab(
                            hsl = hsl,
                            theme = theme,
                            onHslChange = { hsl = it },
                            harmonyType = harmonyType,
                            onHarmonyTypeChange = { harmonyType = it },
```

```
                            recent = state.recentColors.toList(),
                            onPickRecent = { picked ->
                                val p = rgbToHsl(picked)
                                hsl = hsl.copy(h = p.h, s = p.s)
                                if (recordRecent) state.pushRecentColor(hslToColor(hsl))
                            },
                        )
                        ColorTab.Map -> MapTab(
                            userCollections = state.userCollections.toList(),
                            currentColor = hslToColor(hsl),
                            theme = theme,
                            onPick = { picked ->
                                hsl = rgbToHsl(picked)
                                if (recordRecent) state.pushRecentColor(picked)
                            },
                            onDeleteUserCollection = { id ->
                                state.deleteUserCollection(id)
                            },
                            onAddFromPhoto = onAddFromPhoto,
                            allowCollectionEditing = allowCollectionEditing,
                            onCreatePalette = { name ->
                                state.addUserCollection(
                                    ColorCollection(
                                        id = System.currentTimeMillis(),
                                        name = name.ifBlank { "[redacted] [redacted]" },
                                        colors = emptyList(),
                                        builtin = false,
                                    ),
                                )
                            },
                            onAddColorToCollection = { id, color ->
                                state.addColorToCollection(id, color)
                            },
                            onReplaceColorInCollection = { id, slot, color ->
                                state.replaceColorInCollection(id, slot, color)
                            },
                            onRemoveColorFromCollection = { id, slot ->
                                state.removeColorFromCollection(id, slot)
                            },
                            onPinCollection = onPinCollection,
                        )
                    }
                }
            }
        }
    }
}

private val TabContentHeight = 440.dp

@Composable
private fun ColorHeader(preview: Color, onConfirm: () -> Unit, title: String, theme: AppTheme) {
    Row(verticalAlignment = Alignment.CenterVertically) {
        Text(
            title,
            color = theme.textPrimary,
            fontWeight = FontWeight.Bold,
            fontSize = 18.sp,
        )
        Spacer(Modifier.weight(1f))
        Box(
            modifier = Modifier
                .size(28.dp)
                .clip(CircleShape)
                .background(preview)
                .border(1.dp, theme.panelStroke, CircleShape)
                .clickable(onClick = onConfirm),
        )
    }
}

@Composable
private fun TabBar(current: ColorTab, theme: AppTheme, onChange: (ColorTab) -> Unit) {
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .clip(RoundedCornerShape(10.dp))
            .background(theme.panelInset)
            .padding(3.dp),
        horizontalArrangement = Arrangement.spacedBy(3.dp),
    ) {
        TabChip(label = "[redacted]", selected = current == ColorTab.Wheel, theme = theme) { onChange(ColorTab.Wheel) }
        TabChip(label = "[redacted]", selected = current == ColorTab.Map, theme = theme) { onChange(ColorTab.Map) }
    }
}

@Composable
private fun RowScope.TabChip(
    label: String,
    selected: Boolean,
    theme: AppTheme,
    onClick: () -> Unit,
) {
```

```
    Box(
        modifier = Modifier
            .weight(1f)
            .clip(RoundedCornerShape(8.dp))
            .background(if (selected) theme.accent.copy(alpha = 0.25f) else Color.Transparent)
            .clickable(onClick = onClick)
            .padding(vertical = 7.dp),
        contentAlignment = Alignment.Center,
    ) {
        Text(
            label,
            color = if (selected) theme.textPrimary else theme.textSecondary,
            fontSize = 13.sp,
            fontWeight = if (selected) FontWeight.SemiBold else FontWeight.Normal,
        )
    }
}

@Composable
private fun WheelTab(
    hsl: Hsl,
    theme: AppTheme,
    onHslChange: (Hsl) -> Unit,
    harmonyType: HarmonyType,
    onHarmonyTypeChange: (HarmonyType) -> Unit,
    recent: List<Color>,
    onPickRecent: (Color) -> Unit,
) {
    HueSatWheel(
        hsl = hsl,
        harmonyType = harmonyType,
        onHsChange = { newH, newS -> onHslChange(hsl.copy(h = newH, s = newS)) },
        modifier = Modifier
            .fillMaxWidth()
            .height(160.dp),
    )
    Spacer(Modifier.height(8.dp))
    LightnessSlider(
        hsl = hsl,
        theme = theme,
        onChange = { newL -> onHslChange(hsl.copy(l = newL)) },
    )
    Spacer(Modifier.height(8.dp))
    HexRow(color = hslToColor(hsl), theme = theme)
    Spacer(Modifier.height(10.dp))
    HarmonySection(
        hsl = hsl,
        theme = theme,
        harmonyType = harmonyType,
        onHarmonyTypeChange = onHarmonyTypeChange,
        onPick = onHslChange,
    )
    Spacer(Modifier.height(10.dp))
    SectionTitle("RECENT", theme)
    Spacer(Modifier.height(6.dp))
    RecentRow(colors = recent.take(GRID_COLUMNS), theme = theme, onPick = onPickRecent)
}

@Composable
private fun HueSatWheel(
    hsl: Hsl,
    harmonyType: HarmonyType,
    onHsChange: (Float, Float) -> Unit,
    modifier: Modifier = Modifier,
) {
    val latestOnHsChange by rememberUpdatedState(onHsChange)
    Canvas(
        modifier = modifier
            .clip(RoundedCornerShape(16.dp))
            .pointerInput(Unit) {
                detectDragGestures { change, _ ->
                    val (nh, ns) = pickHueSat(change.position, size.width.toFloat(), size.height.toFloat())
                    latestOnHsChange(nh, ns)
                }
            }
            .pointerInput(Unit) {
                detectTapGestures(onTap = { offset ->
                    val (nh, ns) = pickHueSat(offset, size.width.toFloat(), size.height.toFloat())
                    latestOnHsChange(nh, ns)
                })
            },
    ) {
        val w = size.width
        val hgt = size.height
        val cx = w / 2f
        val cy = hgt / 2f
        val outer = minOf(w, hgt) / 2f - 4f
        if (outer <= 0f) return@Canvas

        val steps = 120
        for (i in 0 until steps) {
            val a0 = i * 360f / steps
            val a1 = (i + 1) * 360f / steps
```

```
            val midA = (a0 + a1) / 2f
            val hueColor = hslToColor(midA, 1f, 0.5f)
            drawArc(
                color = hueColor,
                startAngle = a0 - 90f,
                sweepAngle = (a1 - a0) + 0.6f,
                useCenter = true,
                topLeft = Offset(cx - outer, cy - outer),
                size = Size(outer * 2, outer * 2),
            )
        }
        drawCircle(
            brush = Brush.radialGradient(
                colors = listOf(Color.White, Color.White.copy(alpha = 0f)),
                center = Offset(cx, cy),
                radius = outer,
            ),
            radius = outer,
            center = Offset(cx, cy),
        )

        val derived = harmonyType.derive(hsl)
        val verts = derived.map { d -> hslToWheelPoint(d, cx, cy, outer) }
        drawHarmonyShape(harmonyType, verts)

        val current = hslToWheelPoint(hsl, cx, cy, outer)
        drawCircle(color = Color.White, radius = 7f, center = current, style = Stroke(width = 2f))
        drawCircle(color = Color.Black, radius = 7f, center = current, style = Stroke(width = 1f))
    }
}

private fun hslToWheelPoint(h: Hsl, cx: Float, cy: Float, outer: Float): Offset {
    val a = Math.toRadians((h.h - 90.0))
    val r = h.s.coerceIn(0f, 1f) * outer
    return Offset(cx + (r * cos(a)).toFloat(), cy + (r * sin(a)).toFloat())
}

private fun DrawScope.drawHarmonyShape(type: HarmonyType, verts: List<Offset>) {
    if (verts.size < 2) return
    val edges: List<Pair<Int, Int>> = when (type) {
        HarmonyType.Complementary -> listOf(0 to 1)
        HarmonyType.Analogous -> listOf(0 to 1, 1 to 2)
        HarmonyType.Triadic -> listOf(0 to 1, 1 to 2, 2 to 0)
        HarmonyType.SplitComplementary -> listOf(0 to 1, 0 to 2, 1 to 2)
        HarmonyType.Tetradic,
        HarmonyType.Square -> listOf(0 to 1, 1 to 2, 2 to 3, 3 to 0)
    }
    edges.forEach { (a, b) ->
        drawLine(color = Color.White.copy(alpha = 0.85f), start = verts[a], end = verts[b], strokeWidth = 2.5f)
        drawLine(color = Color.Black.copy(alpha = 0.55f), start = verts[a], end = verts[b], strokeWidth = 1f)
    }
    verts.forEach { v ->
        drawCircle(color = Color.White, radius = 4f, center = v)
        drawCircle(color = Color.Black, radius = 4f, center = v, style = Stroke(width = 1f))
    }
}

private fun pickHueSat(p: Offset, w: Float, h: Float): Pair<Float, Float> {
    val cx = w / 2f
    val cy = h / 2f
    val outer = minOf(w, h) / 2f - 4f
    val dx = p.x - cx
    val dy = p.y - cy
    val r = hypot(dx, dy)
    val ang = Math.toDegrees(atan2(dy, dx).toDouble()) + 90.0
    val nh = ((ang + 360.0) % 360.0).toFloat()
    val ns = (r / outer).coerceIn(0f, 1f)
    return nh to ns
}

@Composable
private fun LightnessSlider(hsl: Hsl, theme: AppTheme, onChange: (Float) -> Unit) {
    val hue = hsl.h
    val sat = hsl.s
    val gradient = remember(hue, sat) {
        Brush.horizontalGradient(
            colors = listOf(
                hslToColor(hue, sat, 0f),
                hslToColor(hue, sat, 0.5f),
                hslToColor(hue, sat, 1f),
            ),
        )
    }
    val latestOnChange by rememberUpdatedState(onChange)
    Row(verticalAlignment = Alignment.CenterVertically) {
        Text("L", color = theme.textSecondary, fontSize = 12.sp, modifier = Modifier.width(16.dp))
        Spacer(Modifier.width(8.dp))
        Box(
            modifier = Modifier
                .height(20.dp)
                .fillMaxWidth()
                .clip(RoundedCornerShape(10.dp))
                .background(gradient)
```

```
                .border(1.5.dp, theme.panelStroke, RoundedCornerShape(10.dp))
                .pointerInput(Unit) {
                    detectDragGestures { change, _ ->
                        val f = (change.position.x / size.width).coerceIn(0f, 1f)
                        latestOnChange(f)
                    }
                }
                .pointerInput(Unit) {
                    detectTapGestures(onTap = { offset ->
                        val f = (offset.x / size.width).coerceIn(0f, 1f)
                        latestOnChange(f)
                    })
                },
        ) {
            Canvas(modifier = Modifier.fillMaxSize()) {
                val knobR = (size.height / 2f - 2f).coerceAtMost(size.width / 2f).coerceAtLeast(0f)
                if (knobR <= 0f) return@Canvas
                val knobX = (hsl.l.coerceIn(0f, 1f) * size.width).coerceIn(knobR + 2f, size.width - knobR - 2f)

                drawCircle(color = Color.White, radius = knobR, center = Offset(knobX, size.height / 2f), style = Stroke
    (width = 2.5f))
                drawCircle(color = theme.panelStroke, radius = knobR, center = Offset(knobX, size.height / 2f), style =
    Stroke(width = 1f))
            }
        }
    }
}

@Composable
private fun HexRow(color: Color, theme: AppTheme) {
    Row(verticalAlignment = Alignment.CenterVertically) {
        Text("Hex", color = theme.textSecondary, fontSize = 11.sp)
        Spacer(Modifier.weight(1f))
        Box(
            modifier = Modifier
                .clip(RoundedCornerShape(8.dp))
                .background(theme.panelInset)
                .border(1.dp, theme.panelStroke, RoundedCornerShape(8.dp))
                .padding(horizontal = 10.dp, vertical = 4.dp),
        ) {
            Text(color.toHex6(), color = theme.textPrimary, fontSize = 12.sp, fontWeight = FontWeight.Medium)
        }
    }
}

@Composable
private fun HarmonySection(
    hsl: Hsl,
    theme: AppTheme,
    harmonyType: HarmonyType,
    onHarmonyTypeChange: (HarmonyType) -> Unit,
    onPick: (Hsl) -> Unit,
) {
    SectionTitle("[redacted]", theme)
    Spacer(Modifier.height(6.dp))
    Row(
        horizontalArrangement = Arrangement.spacedBy(6.dp),
        modifier = Modifier.fillMaxWidth(),
    ) {
        HarmonyType.entries.forEach { type ->
            HarmonyIcon(
                type = type,
                theme = theme,
                selected = type == harmonyType,
                modifier = Modifier.weight(1f),
            ) { onHarmonyTypeChange(type) }
        }
    }
    Spacer(Modifier.height(6.dp))
    Text(harmonyType.displayName, color = theme.textSecondary, fontSize = 11.sp, textAlign = TextAlign.Center, modifier
    = Modifier.fillMaxWidth())
    Spacer(Modifier.height(8.dp))
    val derived = remember(harmonyType, hsl) { harmonyType.derive(hsl) }
    Row(
        horizontalArrangement = Arrangement.spacedBy(6.dp),
        verticalAlignment = Alignment.CenterVertically,
        modifier = Modifier.fillMaxWidth(),
    ) {
        derived.forEach { d ->
            Box(
                modifier = Modifier
                    .weight(1f)
                    .height(32.dp)
                    .clip(RoundedCornerShape(8.dp))
                    .background(hslToColor(d))
                    .border(1.dp, theme.panelInset, RoundedCornerShape(8.dp))
                    .clickable { onPick(d) },
            )
        }
        repeat(4 - derived.size) { Box(modifier = Modifier.weight(1f)) }
    }
}
```

```
@Composable
private fun HarmonyIcon(
    type: HarmonyType,
    theme: AppTheme,
    selected: Boolean,
    modifier: Modifier = Modifier,
    onClick: () -> Unit,
) {
    Box(
        modifier = modifier
            .aspectRatio(1f)
            .clip(RoundedCornerShape(8.dp))
            .background(if (selected) theme.accent.copy(alpha = 0.25f) else theme.panelInset)
            .border(1.dp, if (selected) theme.accent else theme.panelInset, RoundedCornerShape(8.dp))
            .clickable(onClick = onClick)
            .padding(6.dp),
        contentAlignment = Alignment.Center,
    ) {
        Canvas(modifier = Modifier.fillMaxSize()) {
            val cx = size.width / 2f
            val cy = size.height / 2f
            val r = minOf(size.width, size.height) / 2f - 1f
            val ringColor = (if (selected) Color.White else theme.textSecondary).copy(alpha = 0.6f)
            val shapeColor = if (selected) Color.White else theme.textSecondary
            drawCircle(color = ringColor, radius = r, center = Offset(cx, cy), style = Stroke(width = 1.2f))
            val verts = type.offsets.map { off ->
                val a = Math.toRadians((off - 90.0).toDouble())
                Offset(cx + (r * 0.78f * cos(a)).toFloat(), cy + (r * 0.78f * sin(a)).toFloat())
            }
            val edges: List<Pair<Int, Int>> = when (type) {
                HarmonyType.Complementary -> listOf(0 to 1)
                HarmonyType.Analogous -> listOf(0 to 1, 1 to 2)
                HarmonyType.Triadic -> listOf(0 to 1, 1 to 2, 2 to 0)
                HarmonyType.SplitComplementary -> listOf(0 to 1, 0 to 2, 1 to 2)
                HarmonyType.Tetradic,
                HarmonyType.Square -> listOf(0 to 1, 1 to 2, 2 to 3, 3 to 0)
            }
            edges.forEach { (a, b) -> drawLine(color = shapeColor, start = verts[a], end = verts[b], strokeWidth = 1.8f)
     }
            verts.forEach { v -> drawCircle(color = shapeColor, radius = 2.2f, center = v) }
        }
    }
}

private const val GRID_COLUMNS = 10

@Composable
private fun MapTab(
    userCollections: List<ColorCollection>,
    currentColor: Color,
    theme: AppTheme,
    onPick: (Color) -> Unit,
    onDeleteUserCollection: (Long) -> Unit,
    onAddFromPhoto: () -> Unit,
    allowCollectionEditing: Boolean,
    onCreatePalette: (String) -> Unit,
    onAddColorToCollection: (Long, Color) -> Unit,
    onReplaceColorInCollection: (Long, Int, Color) -> Unit,
    onRemoveColorFromCollection: (Long, Int) -> Unit,
    onPinCollection: (ColorCollection) -> Unit,
) {
    var showCreate by remember { mutableStateOf(false) }
    Column(verticalArrangement = Arrangement.spacedBy(10.dp)) {
        if (allowCollectionEditing) {
            Row(horizontalArrangement = Arrangement.spacedBy(8.dp)) {
                MapActionButton(label = "+ ■■■■■■■", theme = theme, modifier = Modifier.weight(1f), onClick = { showCrea
    te = true })
                MapActionButton(label = "+ ■■ ■■■■", theme = theme, modifier = Modifier.weight(1f), onClick = onAddFromP
    hoto)
            }
        }
        userCollections.forEach { collection ->
            CollectionCard(collection = collection, theme = theme, onPick = onPick, onDelete = { onDeleteUserCollection(
    collection.id) },
                onAddCurrentColor = { onAddColorToCollection(collection.id, currentColor) },
                onReplaceAt = { slot -> onReplaceColorInCollection(collection.id, slot, currentColor) },
                onRemoveAt = { slot -> onRemoveColorFromCollection(collection.id, slot) },
                onPin = { onPinCollection(collection) },
            )
        }
        BuiltinCollections.all.forEach { collection ->
            CollectionCard(collection = collection, theme = theme, onPick = onPick, onDelete = null, onAddCurrentColor =
     null, onReplaceAt = null, onRemoveAt = null, onPin = { onPinCollection(collection) })
        }
    }
    if (showCreate && allowCollectionEditing) {
        CreatePaletteDialog(theme = theme, onConfirm = { name -> onCreatePalette(name); showCreate = false }, onDismiss
    = { showCreate = false })
    }
}

@Composable
private fun MapActionButton(label: String, theme: AppTheme, modifier: Modifier = Modifier, onClick: () -> Unit) {
```

```
    Box(modifier = modifier.fillMaxWidth().clip(RoundedCornerShape(10.dp)).background(theme.accent.copy(alpha = 0.15f)).
    border(1.2.dp, theme.accentSoft, RoundedCornerShape(10.dp)).clickable(onClick = onClick).padding(vertical = 10.dp, horiz
    ontal = 10.dp), contentAlignment = Alignment.Center) {
        Text(label, color = theme.textPrimary, fontSize = 13.sp, fontWeight = FontWeight.SemiBold)
    }
}

@Composable
private fun CollectionCard(collection: ColorCollection, theme: AppTheme, onPick: (Color) -> Unit, onDelete: (() -> Unit)
    ?, onAddCurrentColor: (() -> Unit)?, onReplaceAt: ((Int) -> Unit)?, onRemoveAt: ((Int) -> Unit)?, onPin: () -> Unit) {
    var editingIdx by remember { mutableStateOf<Int?>(null) }
    val revealPx = with(LocalDensity.current) { 96.dp.toPx() }
    var rawOffset by remember(collection.id) { mutableFloatStateOf(0f) }
    val animatedOffset by animateFloatAsState(rawOffset, animationSpec = tween(140), label = "paletteSwipe")
    val cardShape = RoundedCornerShape(10.dp)
    Box(modifier = Modifier.fillMaxWidth().clip(cardShape)) {
        Box(
            modifier = Modifier.matchParentSize().background(theme.accent.copy(alpha = 0.9f)),
            contentAlignment = Alignment.CenterEnd,
        ) {
            Text(
                "■■■■■■■■■",
                color = Color.White,
                fontSize = 12.sp,
                fontWeight = FontWeight.SemiBold,
                modifier = Modifier.width(96.dp).fillMaxHeight().clickable { onPin(); rawOffset = 0f }.wrapContentHeight
    (Alignment.CenterVertically),
                textAlign = TextAlign.Center,
            )
        }
        Column(
            modifier = Modifier
                .fillMaxWidth()
                .offset { IntOffset(animatedOffset.roundToInt(), 0) }
                .background(theme.panelInset)
                .border(1.dp, theme.panelStroke, cardShape)
                .pointerInput(collection.id, revealPx) {
                    detectHorizontalDragGestures(
                        onHorizontalDrag = { change, dx ->
                            if (dx < 0f || rawOffset < 0f) {
                                change.consume()
                                rawOffset = (rawOffset + dx).coerceIn(-revealPx, 0f)
                            }
                        },
                        onDragEnd = { rawOffset = if (rawOffset < -revealPx * 0.4f) -revealPx else 0f },
                        onDragCancel = { rawOffset = 0f },
                    )
                }
                .padding(horizontal = 8.dp, vertical = 8.dp),
            verticalArrangement = Arrangement.spacedBy(6.dp),
        ) {
        Row(verticalAlignment = Alignment.CenterVertically) {
            Text(collection.name, color = theme.textPrimary, fontSize = 13.sp, fontWeight = FontWeight.SemiBold)
            Spacer(Modifier.weight(1f))
            if (onDelete != null) {
                Box(modifier = Modifier.size(22.dp).clip(CircleShape).background(theme.panelBg.copy(alpha = 0.7f)).borde
    r(1.dp, theme.panelStroke, CircleShape).clickable(onClick = onDelete), contentAlignment = Alignment.Center) {
                    Text("×", color = theme.textSecondary, fontSize = 14.sp, fontWeight = FontWeight.Bold)
                }
            }
        }
        SwatchGrid(colors = collection.composeColors(), theme = theme, onPick = onPick, trailingPlusOnClick = onAddCurre
    ntColor, onLongPressIndex = if (onReplaceAt != null && onRemoveAt != null) { { idx -> editingIdx = idx } } else null)
        }
    }
    val activeIdx = editingIdx
    if (activeIdx != null && onReplaceAt != null && onRemoveAt != null) {
        val swatchColor = collection.composeColors().getOrNull(activeIdx)
        if (swatchColor != null) {
            EditSwatchSheet(swatchColor = swatchColor, theme = theme, onReplace = { onReplaceAt(activeIdx); editingIdx =
     null }, onRemove = { onRemoveAt(activeIdx); editingIdx = null }, onDismiss = { editingIdx = null })
        } else editingIdx = null
    }
}

@OptIn(ExperimentalFoundationApi::class)
@Composable
private fun SwatchGrid(colors: List<Color>, theme: AppTheme, onPick: (Color) -> Unit, trailingPlusOnClick: (() -> Unit)?
     = null, onLongPressIndex: ((Int) -> Unit)? = null) {
    val total = colors.size + if (trailingPlusOnClick != null) 1 else 0
    if (total == 0) return
    val itemRows = (0 until total).toList().chunked(GRID_COLUMNS)
    Column(verticalArrangement = Arrangement.spacedBy(4.dp)) {
        itemRows.forEach { row ->
            Row(horizontalArrangement = Arrangement.spacedBy(4.dp), modifier = Modifier.fillMaxWidth()) {
                row.forEach { i ->
                    if (i < colors.size) {
                        val c = colors[i]
                        val cellModifier = Modifier.weight(1f).aspectRatio(1f).clip(RoundedCornerShape(5.dp)).background
    (c).border(1.dp, theme.panelInset, RoundedCornerShape(5.dp))
                        if (onLongPressIndex != null) Box(modifier = cellModifier.combinedClickable(onClick = { onPick(c
    ) }, onLongClick = { onLongPressIndex(i) }))
                        else Box(modifier = cellModifier.clickable { onPick(c) })
```

```
                    } else PlusTile(theme = theme, modifier = Modifier.weight(1f).aspectRatio(1f), onClick = trailingPlu
    sOnClick!!)
                }
                repeat(GRID_COLUMNS - row.size) { Box(modifier = Modifier.weight(1f)) }
            }
        }
    }
}

@Composable
private fun EditSwatchSheet(swatchColor: Color, theme: AppTheme, onReplace: () -> Unit, onRemove: () -> Unit, onDismiss:
     () -> Unit) {
    Dialog(onDismissRequest = onDismiss) {
        Column(modifier = Modifier.width(280.dp).clip(RoundedCornerShape(16.dp)).background(theme.panelBg).padding(16.dp
    ), verticalArrangement = Arrangement.spacedBy(12.dp)) {
            Row(verticalAlignment = Alignment.CenterVertically) {
                Box(modifier = Modifier.size(28.dp).clip(RoundedCornerShape(6.dp)).background(swatchColor).border(1.dp,
    theme.panelInset, RoundedCornerShape(6.dp)))
                Spacer(Modifier.width(10.dp)); Text("[redacted]", color = theme.textPrimary, fontSize = 16.sp, fontWeight = Fo
    ntWeight.SemiBold)
            }
            DialogButton(label = "[redacted] [redacted]", theme = theme, primary = true, modifier = Modifier.fillMaxWidth(),
    onClick = onReplace)
            DialogButton(label = "[redacted]", theme = theme, primary = false, modifier = Modifier.fillMaxWidth(), onClick
    = onRemove)
            DialogButton(label = "[redacted]", theme = theme, primary = false, modifier = Modifier.fillMaxWidth(), onClick =
     onDismiss)
        }
    }
}

@Composable
private fun PlusTile(theme: AppTheme, modifier: Modifier = Modifier, onClick: () -> Unit) {
    Box(modifier = modifier.clip(RoundedCornerShape(5.dp)).background(theme.panelBg.copy(alpha = 0.5f)).border(1.dp, the
    me.accent.copy(alpha = 0.7f), RoundedCornerShape(5.dp)).clickable(onClick = onClick), contentAlignment = Alignment.Cente
    r) {
        Text("+", color = theme.textPrimary, fontSize = 16.sp, fontWeight = FontWeight.Bold)
    }
}

@Composable
private fun CreatePaletteDialog(theme: AppTheme, onConfirm: (String) -> Unit, onDismiss: () -> Unit) {
    var name by remember { mutableStateOf("") }
    Dialog(onDismissRequest = onDismiss) {
        Column(modifier = Modifier.width(280.dp).clip(RoundedCornerShape(16.dp)).background(theme.panelBg).padding(16.dp
    ), verticalArrangement = Arrangement.spacedBy(12.dp)) {
            Text("[redacted] [redacted]", color = theme.textPrimary, fontSize = 16.sp, fontWeight = FontWeight.SemiBold)
            Box(modifier = Modifier.fillMaxWidth().clip(RoundedCornerShape(8.dp)).background(theme.panelInset).border(1.
    dp, theme.panelInset, RoundedCornerShape(8.dp)).padding(horizontal = 10.dp, vertical = 8.dp)) {
                BasicTextField(value = name, onValueChange = { name = it.take(40) }, singleLine = true, textStyle = Text
    Style(color = theme.textPrimary, fontSize = 14.sp), cursorBrush = SolidColor(theme.accent), decorationBox = { inner -> i
    f (name.isEmpty()) Text("[redacted]", color = theme.textSecondary, fontSize = 14.sp); inner() }, modifier = Modifier.fillM
    axWidth())
            }
            Row(horizontalArrangement = Arrangement.spacedBy(8.dp), modifier = Modifier.fillMaxWidth()) {
                DialogButton(label = "[redacted]", theme = theme, primary = false, modifier = Modifier.weight(1f), onClick =
     onDismiss)
                DialogButton(label = "[redacted]", theme = theme, primary = true, modifier = Modifier.weight(1f), onClick =
     { onConfirm(name) })
            }
        }
    }
}

@Composable
private fun DialogButton(label: String, theme: AppTheme, primary: Boolean, modifier: Modifier = Modifier, onClick: () ->
     Unit) {
    Box(modifier = modifier.clip(RoundedCornerShape(8.dp)).background(if (primary) theme.accent.copy(alpha = 0.35f) else
     theme.panelInset).border(1.dp, if (primary) theme.accent else theme.panelInset, RoundedCornerShape(8.dp)).clickable(onC
    lick = onClick).padding(vertical = 10.dp), contentAlignment = Alignment.Center) {
        Text(label, color = theme.textPrimary, fontSize = 13.sp, fontWeight = FontWeight.SemiBold)
    }
}

@Composable
private fun SectionTitle(text: String, theme: AppTheme) {
    Text(text, color = theme.textSecondary, fontSize = 11.sp, fontWeight = FontWeight.Medium)
}

@Composable
private fun RecentRow(colors: List<Color>, theme: AppTheme, onPick: (Color) -> Unit) {
    Row(horizontalArrangement = Arrangement.spacedBy(4.dp), verticalAlignment = Alignment.CenterVertically, modifier = M
    odifier.fillMaxWidth()) {
        repeat(GRID_COLUMNS) { i ->
            val c = colors.getOrNull(i)
            if (c != null) Box(modifier = Modifier.weight(1f).aspectRatio(1f).clip(RoundedCornerShape(5.dp)).background(
    c).border(1.dp, theme.panelInset, RoundedCornerShape(5.dp)).clickable { onPick(c) })
            else Box(modifier = Modifier.weight(1f).aspectRatio(1f).clip(RoundedCornerShape(5.dp)).background(theme.pane
    lInset).border(1.dp, theme.panelInset, RoundedCornerShape(5.dp)))
        }
    }
}
```

app/src/main/java/com/wetinknext/ui/color/GlesColorState.kt

```
package com.wetinknext.ui.color

import android.content.Context
import androidx.compose.runtime.mutableStateListOf
import androidx.compose.runtime.mutableLongStateOf
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.toArgb
import androidx.compose.ui.geometry.Offset
import org.json.JSONArray
import org.json.JSONObject

class GlesColorState(context: Context? = null) {
    private val prefs = context?.applicationContext?.getSharedPreferences(PrefsName, Context.MODE_PRIVATE)

    val recentColors = mutableStateListOf<Color>()
    val userCollections = mutableStateListOf<ColorCollection>()
    var pinnedCollectionId by mutableLongStateOf(prefs?.getLong(PinnedCollectionKey, NoPinnedCollection) ?: NoPinnedColl
    ection)
        private set
    var pinnedPaletteOffset by mutableStateOf(
        Offset(
            prefs?.getFloat(PinnedPaletteOffsetXKey, DefaultPinnedOffsetX) ?: DefaultPinnedOffsetX,
            prefs?.getFloat(PinnedPaletteOffsetYKey, DefaultPinnedOffsetY) ?: DefaultPinnedOffsetY,
        ),
    )
        private set

    init {
        loadRecentColors()
        loadUserCollections()
    }

    fun pushRecentColor(color: Color) {
        val argb = color.toArgb()
        val existingIndex = recentColors.indexOfFirst { it.toArgb() == argb }
        if (existingIndex == 0) return
        if (existingIndex > 0) recentColors.removeAt(existingIndex)
        recentColors.add(0, color)
        while (recentColors.size > MaxRecentColors) {
            recentColors.removeAt(recentColors.lastIndex)
        }
        saveRecentColors()
    }

    private fun loadRecentColors() {
        val encoded = prefs?.getString(RecentColorsKey, null) ?: return
        encoded.split(',')
            .mapNotNull { it.toIntOrNull() }
            .take(MaxRecentColors)
            .forEach { recentColors.add(Color(it)) }
    }

    private fun saveRecentColors() {
        prefs?.edit()
            ?.putString(RecentColorsKey, recentColors.joinToString(",") { it.toArgb().toString() })
            ?.apply()
    }

    fun addUserCollection(collection: ColorCollection) {
        userCollections.add(0, collection.copy(builtin = false))
        saveUserCollections()
    }

    fun deleteUserCollection(id: Long) {
        userCollections.removeAll { it.id == id }
        if (pinnedCollectionId == id) unpinCollection()
        saveUserCollections()
    }

    fun pinCollection(id: Long) {
        pinnedCollectionId = id
        prefs?.edit()?.putLong(PinnedCollectionKey, id)?.apply()
    }

    fun unpinCollection() {
        pinnedCollectionId = NoPinnedCollection
        prefs?.edit()?.remove(PinnedCollectionKey)?.apply()
    }

    fun persistPinnedPaletteOffset(offset: Offset) {
        pinnedPaletteOffset = offset
        prefs?.edit()
            ?.putFloat(PinnedPaletteOffsetXKey, offset.x)
            ?.putFloat(PinnedPaletteOffsetYKey, offset.y)
            ?.apply()
```

```
    }

    fun addColorToCollection(id: Long, color: Color) {
        updateUserCollection(id) { collection ->
            collection.copy(colors = collection.colors + color.toArgb())
        }
    }

    fun replaceColorInCollection(id: Long, slot: Int, color: Color) {
        updateUserCollection(id) { collection ->
            if (slot !in collection.colors.indices) return@updateUserCollection collection
            val colors = collection.colors.toMutableList()
            colors[slot] = color.toArgb()
            collection.copy(colors = colors)
        }
    }

    fun removeColorFromCollection(id: Long, slot: Int) {
        updateUserCollection(id) { collection ->
            if (slot !in collection.colors.indices) return@updateUserCollection collection
            val colors = collection.colors.toMutableList()
            colors.removeAt(slot)
            collection.copy(colors = colors)
        }
    }

    private fun updateUserCollection(id: Long, transform: (ColorCollection) -> ColorCollection) {
        val index = userCollections.indexOfFirst { it.id == id }
        if (index < 0) return
        userCollections[index] = transform(userCollections[index]).copy(builtin = false)
        saveUserCollections()
    }

    private fun loadUserCollections() {
        val encoded = prefs?.getString(UserCollectionsKey, null) ?: return
        runCatching {
            val array = JSONArray(encoded)
            for (i in 0 until array.length()) {
                val item = array.optJSONObject(i) ?: continue
                val colorsJson = item.optJSONArray("colors") ?: JSONArray()
                val colors = buildList {
                    for (j in 0 until colorsJson.length()) {
                        add(colorsJson.getInt(j))
                    }
                }
                userCollections.add(
                    ColorCollection(
                        id = item.optLong("id", System.currentTimeMillis() + i),
                        name = item.optString("name", "[redacted]"),
                        colors = colors,
                        builtin = false,
                    ),
                )
            }
        }
    }

    private fun saveUserCollections() {
        val array = JSONArray()
        userCollections.filterNot { it.builtin }.forEach { collection ->
            val colors = JSONArray()
            collection.colors.forEach { colors.put(it) }
            array.put(
                JSONObject()
                    .put("id", collection.id)
                    .put("name", collection.name)
                    .put("colors", colors),
            )
        }
        prefs?.edit()
            ?.putString(UserCollectionsKey, array.toString())
            ?.apply()
    }

    private companion object {
        const val PrefsName = "wetinknext_color_state"
        const val RecentColorsKey = "recent_colors"
        const val UserCollectionsKey = "user_collections"
        const val PinnedCollectionKey = "pinned_collection"
        const val PinnedPaletteOffsetXKey = "pinned_palette_offset_x"
        const val PinnedPaletteOffsetYKey = "pinned_palette_offset_y"
        const val NoPinnedCollection = Long.MIN_VALUE
        const val DefaultPinnedOffsetX = 84f
        const val DefaultPinnedOffsetY = 128f
        const val MaxRecentColors = 14
    }
}
```

app/src/main/java/com/wetinknext/ui/components/AdjustmentsPanel.kt

```
package com.wetinknext.ui.components

import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.foundation.verticalScroll
import androidx.compose.material3.HorizontalDivider
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.draw.shadow
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.wetinknext.ui.theme.AppTheme

enum class AdjustmentAction(val title: String) {
    BRIGHTNESS_CONTRAST("[redacted]"),
    THRESHOLD("[redacted]"),
    BRIGHTNESS_TO_OPACITY("[redacted]"),
}

@Composable
fun AdjustmentsPanel(
    theme: AppTheme,
    onClose: () -> Unit,
    onAction: (AdjustmentAction) -> Unit,
    modifier: Modifier = Modifier
) {
    Column(
        modifier = modifier
            .width(280.dp)
            .shadow(8.dp, RoundedCornerShape(16.dp))
            .clip(RoundedCornerShape(16.dp))
            .background(theme.panelBg.copy(alpha = 0.98f))
            .border(1.dp, theme.panelStroke, RoundedCornerShape(16.dp))
            .padding(vertical = 8.dp)
    ) {
        Text(
            text = "[redacted]",
            color = theme.textPrimary,
            fontSize = 16.sp,
            fontWeight = FontWeight.Bold,
            modifier = Modifier.padding(horizontal = 16.dp, vertical = 8.dp)
        )

        Column(
            modifier = Modifier
                .fillMaxWidth()
                .verticalScroll(rememberScrollState())
        ) {
            AdjustmentAction.entries.forEachIndexed { index, action ->
                AdjustmentItem(
                    title = action.title,
                    theme = theme,
                    onClick = {
                        onClose()
                        onAction(action)
                    }
                )
                if (index < AdjustmentAction.entries.size - 1) {
                    HorizontalDivider(
                        modifier = Modifier.padding(horizontal = 16.dp),
                        thickness = 0.5.dp,
                        color = theme.panelStroke.copy(alpha = 0.5f)
                    )
                }
            }
        }
    }
}

@Composable
private fun AdjustmentItem(
    title: String,
    theme: AppTheme,
    onClick: () -> Unit
) {
    Row(
        modifier = Modifier
            .fillMaxWidth()
```

```
            .clickable { onClick() }
            .padding(horizontal = 16.dp, vertical = 12.dp),
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement = Arrangement.SpaceBetween
    ) {
        Text(
            text = title,
            color = theme.textPrimary,
            fontSize = 14.sp,
            modifier = Modifier.weight(1f)
        )
    }
}
```

app/src/main/java/com/wetinknext/ui/components/BrightnessContrastPanel.kt

```
package com.wetinknext.ui.components

import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.Check
import androidx.compose.material.icons.filled.Close
import androidx.compose.material3.Icon
import androidx.compose.material3.Slider
import androidx.compose.material3.SliderDefaults
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.wetinknext.ui.theme.AppTheme
import kotlin.math.roundToInt

@Composable
fun BrightnessContrastPanel(
    theme: AppTheme,
    brightness: Float, // -1..1
    contrast: Float,   // -1..1
    onBrightnessChange: (Float) -> Unit,
    onContrastChange: (Float) -> Unit,
    onApply: () -> Unit,
    onCancel: () -> Unit,
    isApplyEnabled: Boolean = true,
    modifier: Modifier = Modifier
) {
    Column(
        modifier = modifier
            .width(360.dp)
            .clip(RoundedCornerShape(28.dp))
            .background(theme.panelBg.copy(alpha = 0.95f))
            .border(1.2.dp, theme.panelStroke, RoundedCornerShape(28.dp))
            .padding(horizontal = 20.dp, vertical = 16.dp),
        verticalArrangement = Arrangement.spacedBy(16.dp)
    ) {
        Row(
            modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.SpaceBetween,
            verticalAlignment = Alignment.CenterVertically
        ) {
            Text(
                text = "[redacted] [redacted] [redacted]",
                color = theme.textPrimary,
                fontSize = 15.sp,
                fontWeight = FontWeight.Bold
            )

            Row(
                verticalAlignment = Alignment.CenterVertically,
                horizontalArrangement = Arrangement.spacedBy(12.dp)
            ) {
                Box(
                    modifier = Modifier
                        .size(36.dp)
                        .clip(CircleShape)
                        .background(theme.panelInset)
                        .clickable { onCancel() },
                    contentAlignment = Alignment.Center
                ) {
```

```
                    Icon(Icons.Default.Close, null, tint = theme.textPrimary, modifier = Modifier.size(20.dp))
                }

                Box(
                    modifier = Modifier
                        .size(36.dp)
                        .clip(CircleShape)
                        .background(if (isApplyEnabled) theme.accent else theme.panelInset)
                        .clickable(enabled = isApplyEnabled) { onApply() },
                    contentAlignment = Alignment.Center
                ) {
                    Icon(Icons.Default.Check, null, tint = Color.White, modifier = Modifier.size(20.dp))
                }
            }
        }

        // Brightness
        Column(verticalArrangement = Arrangement.spacedBy(4.dp)) {
            Row(
                modifier = Modifier.fillMaxWidth(),
                horizontalArrangement = Arrangement.SpaceBetween
            ) {
                Text("■■■■■■■■", color = theme.textSecondary, fontSize = 13.sp)
                Text("${(brightness * 100).roundToInt()}", color = theme.accent, fontSize = 13.sp, fontWeight = FontWeig
    ht.Bold)
            }
            Slider(
                value = brightness,
                onValueChange = onBrightnessChange,
                valueRange = -1f..1f,
                colors = SliderDefaults.colors(
                    thumbColor = theme.accent,
                    activeTrackColor = theme.accent,
                    inactiveTrackColor = theme.panelInset
                )
            )
        }

        // Contrast
        Column(verticalArrangement = Arrangement.spacedBy(4.dp)) {
            Row(
                modifier = Modifier.fillMaxWidth(),
                horizontalArrangement = Arrangement.SpaceBetween
            ) {
                Text("■■■■■■■■■", color = theme.textSecondary, fontSize = 13.sp)
                Text("${(contrast * 100).roundToInt()}", color = theme.accent, fontSize = 13.sp, fontWeight = FontWeight
    .Bold)
            }
            Slider(
                value = contrast,
                onValueChange = onContrastChange,
                valueRange = -1f..1f,
                colors = SliderDefaults.colors(
                    thumbColor = theme.accent,
                    activeTrackColor = theme.accent,
                    inactiveTrackColor = theme.panelInset
                )
            )
        }
    }
}
```

app/src/main/java/com/wetinknext/ui/components/BrightnessToOpacityPanel.kt

```
package com.wetinknext.ui.components

import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.Check
import androidx.compose.material.icons.filled.Close
import androidx.compose.material3.Icon
import androidx.compose.material3.Switch
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.wetinknext.ui.theme.AppTheme
```

```
@Composable
fun BrightnessToOpacityPanel(
    theme: AppTheme,
    isInverted: Boolean,
    onInvertChange: (Boolean) -> Unit,
    onApply: () -> Unit,
    onCancel: () -> Unit,
    isApplyEnabled: Boolean = true,
    modifier: Modifier = Modifier
) {
    Column(
        modifier = modifier
            .width(360.dp)
            .clip(RoundedCornerShape(28.dp))
            .background(theme.panelBg.copy(alpha = 0.95f))
            .border(1.2.dp, theme.panelStroke, RoundedCornerShape(28.dp))
            .padding(horizontal = 20.dp, vertical = 16.dp),
        verticalArrangement = Arrangement.spacedBy(16.dp)
    ) {
        Row(
            modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.SpaceBetween,
            verticalAlignment = Alignment.CenterVertically
        ) {
            Text(
                text = "[redacted]",
                color = theme.textPrimary,
                fontSize = 15.sp,
                fontWeight = FontWeight.Bold
            )

            Row(
                verticalAlignment = Alignment.CenterVertically,
                horizontalArrangement = Arrangement.spacedBy(12.dp)
            ) {
                Box(
                    modifier = Modifier
                        .size(36.dp)
                        .clip(CircleShape)
                        .background(theme.panelInset)
                        .clickable { onCancel() },
                    contentAlignment = Alignment.Center
                ) {
                    Icon(Icons.Default.Close, null, tint = theme.textPrimary, modifier = Modifier.size(20.dp))
                }

                Box(
                    modifier = Modifier
                        .size(36.dp)
                        .clip(CircleShape)
                        .background(if (isApplyEnabled) theme.accent else theme.panelInset)
                        .clickable(enabled = isApplyEnabled) { onApply() },
                    contentAlignment = Alignment.Center
                ) {
                    Icon(Icons.Default.Check, null, tint = Color.White, modifier = Modifier.size(20.dp))
                }
            }
        }

        Text(
            text = "[redacted], [redacted] — [redacted]",
            color = theme.textSecondary,
            fontSize = 12.sp,
            lineHeight = 16.sp
        )

        Row(
            modifier = Modifier.fillMaxWidth(),
            verticalAlignment = Alignment.CenterVertically,
            horizontalArrangement = Arrangement.SpaceBetween
        ) {
            Text(
                text = "[redacted]",
                color = theme.textPrimary,
                fontSize = 14.sp
            )
            Switch(
                checked = isInverted,
                onCheckedChange = onInvertChange
            )
        }
    }
}
```

app/src/main/java/com/wetinknext/ui/components/BrushPanel.kt

```
package com.wetinknext.ui.components

import androidx.compose.foundation.Canvas
import androidx.compose.foundation.Image
```

```
import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.clickable
import androidx.compose.foundation.gestures.detectTapGestures
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.*
import androidx.compose.material3.Icon
import androidx.compose.material3.Text
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.draw.drawBehind
import androidx.compose.ui.geometry.Offset
import androidx.compose.ui.geometry.Size
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.ImageBitmap
import androidx.compose.ui.graphics.vector.ImageVector
import androidx.compose.ui.input.pointer.pointerInput
import androidx.compose.ui.layout.ContentScale
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.wetinknext.engine.brush.BrushLibrary
import com.wetinknext.engine.brush.BrushPreset
import com.wetinknext.engine.brush.BrushUiCategory
import com.wetinknext.ui.theme.AppTheme

private val BRUSH_SWIPE_REVEAL_WIDTH = 88.dp

@Composable
fun BrushPanel(
    currentBrush: BrushPreset,
    onBrushSelect: (BrushPreset) -> Unit,
    onBrushStudioOpen: (BrushPreset) -> Unit,
    theme: AppTheme,
    onDismiss: () -> Unit,
    onDuplicate: (BrushPreset) -> Unit,
    onDelete: (BrushPreset) -> Unit,
    modifier: Modifier = Modifier,
    categories: List<BrushUiCategory> = BrushLibrary.allCategories,
    previews: Map<String, ImageBitmap> = emptyMap(),
    previewKey: (BrushPreset) -> String = { it.id },
    onRequestPreview: (BrushPreset) -> Unit = {},
    title: String = "[redacted]",
) {
    fun categoryIndexForCurrentBrush(): Int {
        return categories.indexOfFirst { category -> category.brushes.any { it.id == currentBrush.id } }
            .takeIf { it >= 0 }
            ?: 0
    }
    var selectedCategoryIndex by remember(categories, currentBrush.id) {
        mutableIntStateOf(categoryIndexForCurrentBrush())
    }
    LaunchedEffect(categories, currentBrush.id) {
        selectedCategoryIndex = categoryIndexForCurrentBrush()
    }
    val selectedCategory = categories.getOrNull(selectedCategoryIndex)

    PanelSurface(
        modifier = modifier
            .width(420.dp)
            .height(560.dp),
    ) {
        Row(Modifier.fillMaxSize()) {
            Column(
                modifier = Modifier
                    .width(70.dp)
                    .fillMaxHeight()
                    .clip(RoundedCornerShape(topStart = 22.dp, bottomStart = 22.dp))
                    .background(theme.panelInset)
                    .drawBehind {
                        drawLine(
                            color = theme.panelStroke,
                            start = Offset(size.width, 0f),
                            end = Offset(size.width, size.height),
                            strokeWidth = 1.dp.toPx(),
                        )
                    }
                    .padding(vertical = 16.dp),
                horizontalAlignment = Alignment.CenterHorizontally,
                verticalArrangement = Arrangement.spacedBy(20.dp),
            ) {
                categories.forEachIndexed { index, category ->
                    val isSelected = index == selectedCategoryIndex
                    Box(
                        modifier = Modifier
                            .size(44.dp)
```

```
                    .clip(CircleShape)
                    .background(if (isSelected) theme.accent.copy(alpha = 0.3f) else Color.Transparent)
                    .border(
                        if (isSelected) 1.dp else 0.dp,
                        if (isSelected) theme.accentSoft else Color.Transparent,
                        CircleShape,
                    )
                    .clickable { selectedCategoryIndex = index },
                contentAlignment = Alignment.Center,
            ) {
                Icon(
                    category.icon,
                    contentDescription = category.name,
                    tint = if (isSelected) theme.accent else theme.iconInactive,
                    modifier = Modifier.size(24.dp),
                )
            }
        }
    }

    Column(
        modifier = Modifier
            .weight(1f)
            .fillMaxHeight()
            .padding(14.dp),
    ) {
        Row(
            verticalAlignment = Alignment.CenterVertically,
            horizontalArrangement = Arrangement.spacedBy(12.dp)
        ) {
            Column(verticalArrangement = Arrangement.spacedBy(1.dp)) {
                Text(
                    text = title,
                    color = theme.textPrimary,
                    fontSize = 18.sp,
                    fontWeight = FontWeight.Bold,
                )
                selectedCategory?.let { category ->
                    Text(
                        text = category.name,
                        color = theme.accent,
                        fontSize = 11.sp,
                        fontWeight = FontWeight.SemiBold,
                    )
                }
            }
            Spacer(Modifier.weight(1f))

            Icon(
                imageVector = Icons.Default.Close,
                contentDescription = "[redacted]",
                tint = theme.textSecondary,
                modifier = Modifier.size(24.dp).clickable { onDismiss() }
            )
        }

        Spacer(Modifier.height(14.dp))

        LazyColumn(
            verticalArrangement = Arrangement.spacedBy(8.dp),
            modifier = Modifier.fillMaxSize(),
        ) {
            items(selectedCategory?.brushes.orEmpty(), key = { it.id }) { brush ->
                val isSelected = brush.id == currentBrush.id
                BrushItem(
                    brush = brush,
                    isSelected = isSelected,
                    theme = theme,
                    preview = previews[previewKey(brush)],
                    onRequestPreview = { onRequestPreview(brush) },
                    onStudioOpen = { onBrushStudioOpen(brush) },
                    onDuplicate = { onDuplicate(brush) },
                    onDelete = { onDelete(brush) },
                    onClick = {
                        if (isSelected) onBrushStudioOpen(brush)
                        else onBrushSelect(brush)
                    },
                )
            }

            if (selectedCategory?.brushes.isNullOrEmpty()) {
                item {
                    Text(
                        text = "[redacted]",
                        color = theme.textSecondary,
                        fontSize = 14.sp,
                        modifier = Modifier.padding(top = 20.dp),
                    )
                }
            }
        }
    }
}
```

```
    }
}

@Composable
private fun BrushItem(
    brush: BrushPreset,
    isSelected: Boolean,
    theme: AppTheme,
    preview: ImageBitmap?,
    onRequestPreview: () -> Unit,
    onStudioOpen: () -> Unit,
    onDuplicate: () -> Unit,
    onDelete: () -> Unit,
    onClick: () -> Unit,
) {
    val contentBg = if (isSelected) theme.panelBgSolid else theme.panelInset
    LaunchedEffect(brush.id, brush.settings, preview) {
        if (preview == null) {
            onRequestPreview()
        }
    }

    SwipeRevealRow(
        revealWidth = BRUSH_SWIPE_REVEAL_WIDTH,
        onSwipeRight = onStudioOpen,
        modifier = Modifier
            .fillMaxWidth()
            .clip(RoundedCornerShape(8.dp))
            .background(theme.panelInset),
        actions = {
            BrushSwipeActions(
                theme = theme,
                onDuplicate = onDuplicate,
                onDelete = onDelete,
                canDelete = true // Simplified for now
            )
        },
    ) {
        Box(
            modifier = Modifier
                .fillMaxWidth()
                .background(contentBg)
                .border(1.dp, if (isSelected) theme.accentSoft else theme.panelStroke, RoundedCornerShape(8.dp))
                .pointerInput(brush.id) {
                    detectTapGestures(
                        onTap = { onClick() },
                    )
                }
                .padding(10.dp),
        ) {
            Row(
                verticalAlignment = Alignment.CenterVertically,
                horizontalArrangement = Arrangement.spacedBy(12.dp),
            ) {
                BrushPreviewStrip(
                    preview = preview,
                    brush = brush,
                    theme = theme,
                    modifier = Modifier
                        .width(142.dp)
                        .height(46.dp),
                )
                Column(Modifier.weight(1f)) {
                    Text(
                        text = brush.settings.name,
                        color = if (isSelected) theme.accent else theme.textPrimary,
                        fontSize = 15.sp,
                        fontWeight = if (isSelected) FontWeight.SemiBold else FontWeight.Medium,
                        maxLines = 1,
                    )
                    Text(
                        text = "■■■■■■: ${brush.settings.baseRadiusPx.toInt()} px",
                        color = theme.textSecondary,
                        fontSize = 12.sp,
                        maxLines = 1,
                    )
                }
                if (isSelected) {
                    Box(
                        Modifier
                            .size(8.dp)
                            .clip(CircleShape)
                            .background(theme.accent),
                    )
                }
            }
        }
    }
}

@Composable
private fun BrushSwipeActions(
    theme: AppTheme,
```

```
    onDuplicate: () -> Unit,
    onDelete: () -> Unit,
    canDelete: Boolean
) {
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .fillMaxHeight(),
        horizontalArrangement = Arrangement.End,
        verticalAlignment = Alignment.CenterVertically,
    ) {
        BrushSwipeButton(
            icon = Icons.Default.ContentCopy,
            label = "[redacted]",
            color = theme.accent,
            onClick = onDuplicate
        )
        BrushSwipeButton(
            icon = Icons.Default.Delete,
            label = "[redacted].",
            color = theme.danger,
            enabled = canDelete,
            onClick = onDelete
        )
    }
}

@Composable
private fun BrushSwipeButton(
    icon: ImageVector,
    label: String,
    color: Color,
    enabled: Boolean = true,
    onClick: () -> Unit,
) {
    Column(
        modifier = Modifier
            .width(44.dp)
            .fillMaxHeight()
            .background(if (enabled) color else color.copy(alpha = 0.3f))
            .clickable(enabled = enabled, onClick = onClick)
            .padding(vertical = 4.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center,
    ) {
        Icon(icon, contentDescription = label, tint = Color.White, modifier = Modifier.size(18.dp))
        Text(label, color = Color.White, fontSize = 8.sp, fontWeight = FontWeight.Medium)
    }
}

@Composable
private fun BrushPreviewStrip(
    preview: ImageBitmap?,
    brush: BrushPreset,
    theme: AppTheme,
    modifier: Modifier = Modifier,
) {
    Box(
        modifier = modifier
            .clip(RoundedCornerShape(4.dp))
            .background(Color.White)
            .border(1.dp, theme.panelStroke, RoundedCornerShape(4.dp)),
        contentAlignment = Alignment.Center,
    ) {
        PreviewCheckerboard(Modifier.fillMaxSize(), cellSize = 7f)
        if (preview != null) {
            Image(
                bitmap = preview,
                contentDescription = null,
                contentScale = ContentScale.Fit,
                modifier = Modifier.fillMaxSize(),
            )
        } else {
            Canvas(Modifier.fillMaxSize()) {
                drawLine(
                    color = theme.accent.copy(alpha = brush.settings.opacity.coerceIn(0.25f, 1f)),
                    start = Offset(size.width * 0.12f, size.height * 0.5f),
                    end = Offset(size.width * 0.88f, size.height * 0.5f),
                    strokeWidth = (brush.settings.baseRadiusPx * 2f).coerceIn(2f, 12f),
                )
            }
        }
    }
}

@Composable
private fun PreviewCheckerboard(
    modifier: Modifier = Modifier,
    cellSize: Float,
) {
    Canvas(modifier) {
        val light = Color(0xFFE9E9E9)
        val dark = Color(0xFFD0D0D0)
```

```
        var y = 0f
        var row = 0
        while (y < size.height) {
            var x = 0f
            var col = 0
            while (x < size.width) {
                drawRect(
                    color = if ((row + col) % 2 == 0) light else dark,
                    topLeft = Offset(x, y),
                    size = Size(cellSize, cellSize),
                )
                x += cellSize
                col += 1
            }
            y += cellSize
            row += 1
        }
    }
}
```

app/src/main/java/com/wetinknext/ui/components/LayersPanel.kt

```
package com.wetinknext.ui.components

import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.*
import androidx.compose.material3.*
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.vector.ImageVector
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.wetinknext.ui.state.LayerItem
import com.wetinknext.ui.state.LayerState
import com.wetinknext.ui.theme.AppTheme
import kotlin.math.roundToInt

private val PANEL_WIDTH = 320.dp

@Composable
fun LayersPanel(
    state: LayerState,
    theme: AppTheme,
    modifier: Modifier = Modifier,
    onDismiss: () -> Unit,
    onAddLayer: () -> Unit,
    onSelectLayer: (Long) -> Unit,
    onVisibleChange: (Long, Boolean) -> Unit,
    onOpacityChange: (Long, Float) -> Unit,
    onRemoveLayer: (Long) -> Unit,
) {
    PanelSurface(
        modifier = modifier.width(PANEL_WIDTH),
    ) {
        Column(
            modifier = Modifier.padding(10.dp),
            verticalArrangement = Arrangement.spacedBy(8.dp),
        ) {
            Row(
                verticalAlignment = Alignment.CenterVertically,
                horizontalArrangement = Arrangement.spacedBy(6.dp),
            ) {
                Text(
                    "[redacted]",
                    color = theme.textPrimary,
                    fontWeight = FontWeight.Bold,
                    fontSize = 18.sp,
                )
                Spacer(Modifier.weight(1f))
                HeaderIconButton(
                    theme = theme,
                    icon = Icons.Default.Add,
                    description = "[redacted] [redacted]",
                    onClick = onAddLayer,
                )
                HeaderIconButton(
                    theme = theme,
```

```
                    icon = Icons.Default.Close,
                    description = "[redacted]",
                    onClick = onDismiss,
                )
            }

            LazyColumn(
                modifier = Modifier
                    .fillMaxWidth()
                    .heightIn(max = 400.dp),
                verticalArrangement = Arrangement.spacedBy(4.dp),
            ) {
                items(
                    items = state.layers.asReversed(),
                    key = { it.id },
                ) { layerItem ->
                    LayerRow(
                        layer = layerItem,
                        theme = theme,
                        active = layerItem.id == state.activeLayerId,
                        onSelect = { onSelectLayer(layerItem.id) },
                        onVisibleChange = { onVisibleChange(layerItem.id, it) },
                        onOpacityChange = { onOpacityChange(layerItem.id, it) },
                        onDelete = { onRemoveLayer(layerItem.id) },
                        canDelete = state.layers.size > 1 && !layerItem.locked
                    )
                }
            }
        }
    }
}

@Composable
private fun LayerRow(
    layer: LayerItem,
    theme: AppTheme,
    active: Boolean,
    onSelect: () -> Unit,
    onVisibleChange: (Boolean) -> Unit,
    onOpacityChange: (Float) -> Unit,
    onDelete: () -> Unit,
    canDelete: Boolean,
) {
    val cardBg = if (active) theme.panelBgSolid else theme.panelBg

    Column(
        modifier = Modifier
            .fillMaxWidth()
            .clip(RoundedCornerShape(12.dp))
            .background(cardBg)
            .border(
                width = if (active) 1.2.dp else 0.dp,
                color = if (active) theme.accent else Color.Transparent,
                shape = RoundedCornerShape(12.dp),
            )
            .clickable { onSelect() }
            .padding(8.dp),
        verticalArrangement = Arrangement.spacedBy(4.dp)
    ) {
        Row(
            modifier = Modifier.fillMaxWidth(),
            verticalAlignment = Alignment.CenterVertically,
        ) {
            HeaderIconButton(
                theme = theme,
                icon = if (layer.visible) Icons.Default.Visibility else Icons.Default.VisibilityOff,
                description = "[redacted]",
                onClick = { onVisibleChange(!layer.visible) },
                tint = if (layer.visible) (if (active) theme.accent else theme.textPrimary) else theme.iconInactive,
            )

            Spacer(Modifier.width(8.dp))

            Text(
                layer.name,
                modifier = Modifier.weight(1f),
                color = theme.textPrimary,
                fontSize = 14.sp,
                fontWeight = if (active) FontWeight.SemiBold else FontWeight.Normal,
                maxLines = 1,
                overflow = TextOverflow.Ellipsis,
            )

            if (canDelete) {
                HeaderIconButton(
                    theme = theme,
                    icon = Icons.Default.Delete,
                    description = "[redacted]",
                    onClick = onDelete,
                    tint = theme.danger
                )
            }
        }
```

```
        if (active) {
            Column(modifier = Modifier.fillMaxWidth().padding(horizontal = 4.dp)) {
                Row(verticalAlignment = Alignment.CenterVertically) {
                    Text("■■■■■■■■■■■■■■■", color = theme.textSecondary, fontSize = 11.sp)
                    Spacer(Modifier.weight(1f))
                    Text(
                        "${(layer.opacity * 100f).roundToInt()}%",
                        color = theme.textSecondary,
                        fontSize = 11.sp,
                    )
                }
                Slider(
                    value = layer.opacity.coerceIn(0f, 1f),
                    onValueChange = onOpacityChange,
                    valueRange = 0f..1f,
                    colors = SliderDefaults.colors(
                        thumbColor = theme.accent,
                        activeTrackColor = theme.accent,
                        inactiveTrackColor = theme.accentMuted,
                    ),
                )
            }
        }
    }
}

@Composable
private fun HeaderIconButton(
    theme: AppTheme,
    icon: ImageVector,
    description: String,
    onClick: () -> Unit,
    enabled: Boolean = true,
    tint: Color = theme.textPrimary,
) {
    Box(
        modifier = Modifier
            .size(32.dp)
            .clip(CircleShape)
            .background(if (enabled) theme.panelInsetSoft else theme.panelInset)
            .clickable(enabled = enabled, onClick = onClick),
        contentAlignment = Alignment.Center,
    ) {
        Icon(
            icon,
            contentDescription = description,
            tint = if (enabled) tint else theme.iconInactive,
            modifier = Modifier.size(18.dp),
        )
    }
}
```

app/src/main/java/com/wetinknext/ui/components/PanelSurface.kt

```
package com.wetinknext.ui.components

import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.BoxScope
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.draw.shadow
import androidx.compose.ui.input.pointer.PointerEventPass
import androidx.compose.ui.input.pointer.pointerInput
import androidx.compose.ui.unit.dp
import com.wetinknext.ui.theme.LocalAppTheme

@Composable
fun PanelSurface(
    modifier: Modifier = Modifier,
    content: @Composable BoxScope.() -> Unit,
) {
    val theme = LocalAppTheme.current
    val shape = RoundedCornerShape(22.dp)
    Box(
        modifier = modifier
            .shadow(12.dp, shape, ambientColor = theme.panelInset.copy(alpha = 0.5f), spotColor = theme.panelInset.copy(
    alpha = 0.5f))
            .clip(shape)
            .background(theme.panelBg)
            .border(1.2.dp, theme.panelStroke, shape)
    ) {
        Box(modifier = Modifier.matchParentSize().swallowChrome())
        content()
    }
}
```

```
private fun Modifier.swallowChrome(): Modifier = pointerInput(Unit) {
    awaitPointerEventScope {
        while (true) {
            val event = awaitPointerEvent(PointerEventPass.Initial)
            for (change in event.changes) {
                if (change.pressed && !change.previousPressed) {
                    change.consume()
                }
            }
        }
    }
}
```

app/src/main/java/com/wetinknext/ui/components/SelectionToolbar.kt

```
package com.wetinknext.ui.components

import androidx.compose.foundation.BorderStroke
import androidx.compose.foundation.background
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.filled.CallMerge
import androidx.compose.material.icons.filled.*
import androidx.compose.material3.Icon
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.vector.ImageVector
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.wetinknext.ui.theme.AppTheme

enum class SelectionShapeUi {
    FREEHAND,
    RECTANGLE,
    ELLIPSE,
}

@Composable
fun SelectionToolbar(
    theme: AppTheme,
    currentShape: SelectionShapeUi,
    onShapeChange: (SelectionShapeUi) -> Unit,
    onDone: () -> Unit,
    modifier: Modifier = Modifier,
) {
    val panelBorder = BorderStroke(1.dp, theme.panelStroke)

    Column(
        modifier = modifier
            .fillMaxWidth()
            .navigationBarsPadding()
            .padding(horizontal = 16.dp, vertical = 20.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.spacedBy(12.dp),
    ) {
        // Row 1 — shape modes
        Surface(
            color = theme.panelBg.copy(alpha = 0.94f),
            shape = RoundedCornerShape(24.dp),
            border = panelBorder,
        ) {
            Row(
                modifier = Modifier
                    .width(340.dp)
                    .padding(4.dp)
                    .height(32.dp)
                    .clip(RoundedCornerShape(20.dp))
                    .background(theme.panelInsetSoft),
                verticalAlignment = Alignment.CenterVertically,
            ) {
                val shapes = listOf(
                    SelectionShapeUi.FREEHAND to "■■ ■■■■",
                    SelectionShapeUi.RECTANGLE to "■■■■■■■■■■■■■■",
                    SelectionShapeUi.ELLIPSE to "■■■■■■",
                )
                shapes.forEach { (shape, label) ->
                    val isSelected = currentShape == shape
                    Box(
                        modifier = Modifier
                            .weight(1f)
                            .fillMaxHeight()
```

```
                            .clip(RoundedCornerShape(20.dp))
                            .background(if (isSelected) theme.accent else Color.Transparent)
                            .clickable { onShapeChange(shape) },
                        contentAlignment = Alignment.Center,
                    ) {
                        Text(
                            text = label,
                            color = if (isSelected) Color.White else theme.textPrimary,
                            fontSize = 13.sp,
                            fontWeight = if (isSelected) FontWeight.Bold else FontWeight.Normal,
                            maxLines = 1,
                        )
                    }
                }
            }
        }

        // Row 2 — Action icons (mostly disabled for now)
        Surface(
            color = theme.panelBg.copy(alpha = 0.94f),
            shape = RoundedCornerShape(24.dp),
            border = panelBorder,
        ) {
            Row(
                modifier = Modifier.padding(horizontal = 4.dp, vertical = 4.dp),
                horizontalArrangement = Arrangement.spacedBy(4.dp),
                verticalAlignment = Alignment.CenterVertically,
            ) {
                SelectionIconButton(Icons.Default.Add, "[redacted]", theme, isSelected = true, enabled = false)
                SelectionIconButton(Icons.Default.Delete, "[redacted]", theme, enabled = false)
                SelectionIconButton(Icons.AutoMirrored.Filled.CallMerge, "[redacted]", theme, enabled = false)
                SelectionIconButton(Icons.Default.Flip, "[redacted]", theme, enabled = false)
                SelectionIconButton(Icons.Default.FormatColorFill, "[redacted]", theme, enabled = false)
                SelectionIconButton(Icons.Default.ContentCopy, "[redacted]", theme, enabled = false)
                SelectionIconButton(Icons.Default.ContentCut, "[redacted]", theme, enabled = false)
                SelectionIconButton(Icons.Default.AddToPhotos, "[redacted]", theme, enabled = false)
            }
        }

        // Row 3 — Reset / Delete / Done
        Row(
            horizontalArrangement = Arrangement.spacedBy(16.dp),
            verticalAlignment = Alignment.CenterVertically,
        ) {
            SelectionRoundButton(Icons.Default.Refresh, "[redacted]", theme, enabled = false)
            SelectionRoundButton(Icons.Default.Delete, "[redacted]", theme, enabled = false)
            Surface(
                color = theme.accent,
                shape = CircleShape,
                modifier = Modifier.size(48.dp),
            ) {
                Box(
                    modifier = Modifier
                        .fillMaxSize()
                        .clickable { onDone() },
                    contentAlignment = Alignment.Center,
                ) {
                    Icon(Icons.Default.Check, "[redacted]", tint = Color.White)
                }
            }
        }
    }
}

@Composable
private fun SelectionIconButton(
    icon: ImageVector,
    description: String,
    theme: AppTheme,
    isSelected: Boolean = false,
    enabled: Boolean = true,
    onClick: () -> Unit = {},
) {
    Box(
        modifier = Modifier
            .size(48.dp)
            .clip(RoundedCornerShape(8.dp))
            .background(if (isSelected && enabled) theme.accent.copy(alpha = 0.30f) else Color.Transparent)
            .clickable(enabled = enabled) { onClick() },
        contentAlignment = Alignment.Center,
    ) {
        Icon(
            imageVector = icon,
            contentDescription = description,
            tint = if (!enabled) theme.iconInactive.copy(alpha = 0.5f) else if (isSelected) theme.accent else theme.text
    Primary,
            modifier = Modifier.size(24.dp),
        )
    }
}

@Composable
private fun SelectionRoundButton(
```

```
    icon: ImageVector,
    description: String,
    theme: AppTheme,
    enabled: Boolean = true,
    onClick: () -> Unit = {},
) {
    Surface(
        color = if (enabled) theme.panelBg.copy(alpha = 0.9f) else theme.panelBg.copy(alpha = 0.4f),
        shape = CircleShape,
        border = BorderStroke(1.dp, if (enabled) theme.panelStroke else theme.panelStroke.copy(alpha = 0.4f)),
        modifier = Modifier.size(48.dp),
    ) {
        Box(
            modifier = Modifier
                .fillMaxSize()
                .clickable(enabled = enabled) { onClick() },
            contentAlignment = Alignment.Center,
        ) {
            Icon(
                icon,
                description,
                tint = if (enabled) theme.textPrimary else theme.textPrimary.copy(alpha = 0.4f)
            )
        }
    }
}
```

app/src/main/java/com/wetinknext/ui/components/SwipeRevealRow.kt

```
package com.wetinknext.ui.components

import androidx.compose.animation.core.animateFloatAsState
import androidx.compose.foundation.gestures.detectHorizontalDragGestures
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.RowScope
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.offset
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.Modifier
import androidx.compose.ui.input.pointer.pointerInput
import androidx.compose.ui.platform.LocalDensity
import androidx.compose.ui.unit.Dp
import androidx.compose.ui.unit.IntOffset
import androidx.compose.ui.unit.dp
import kotlin.math.roundToInt

@Composable
fun SwipeRevealRow(
    revealWidth: Dp,
    actions: @Composable RowScope.() -> Unit,
    modifier: Modifier = Modifier,
    rightSwipeThreshold: Dp = 56.dp,
    onSwipeRight: (() -> Unit)? = null,
    content: @Composable () -> Unit,
) {
    var rawOffset by remember { mutableStateOf(0f) }
    var rightDragOffset by remember { mutableStateOf(0f) }
    val revealPx = with(LocalDensity.current) { revealWidth.toPx() }
    val rightSwipeThresholdPx = with(LocalDensity.current) { rightSwipeThreshold.toPx() }
    val animatedOffset by animateFloatAsState(
        targetValue = rawOffset.coerceIn(-revealPx, 0f),
        label = "swipeRevealRow",
    )

    Box(modifier = modifier.fillMaxWidth()) {
        Row(
            modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.End,
            content = actions,
        )

        Box(
            modifier = Modifier
                .fillMaxWidth()
                .offset { IntOffset(animatedOffset.roundToInt(), 0) }
                .pointerInput(revealWidth) {
                    detectHorizontalDragGestures(
                        onHorizontalDrag = { change, dx ->
                            if (dx < 0f || rawOffset < 0f) {
                                change.consume()
                                rawOffset += dx
```

```
                                rightDragOffset = 0f
                            } else if (dx > 0f && rawOffset == 0f && onSwipeRight != null) {
                                change.consume()
                                rightDragOffset += dx
                            }
                        },
                        onDragEnd = {
                            if (rightDragOffset > rightSwipeThresholdPx) {
                                onSwipeRight?.invoke()
                            }
                            rawOffset = if (rawOffset < -revealPx * 0.4f) -revealPx else 0f
                            rightDragOffset = 0f
                        },
                        onDragCancel = {
                            rawOffset = 0f
                            rightDragOffset = 0f
                        },
                    )
                },
        ) {
            content()
        }
    }
}
```

app/src/main/java/com/wetinknext/ui/components/ThresholdPanel.kt

```
package com.wetinknext.ui.components

import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.Check
import androidx.compose.material.icons.filled.Close
import androidx.compose.material3.Icon
import androidx.compose.material3.Slider
import androidx.compose.material3.SliderDefaults
import androidx.compose.material3.Switch
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.wetinknext.ui.theme.AppTheme
import kotlin.math.roundToInt

@Composable
fun ThresholdPanel(
    theme: AppTheme,
    threshold: Float, // 0..1
    preserveTransparency: Boolean,
    onThresholdChange: (Float) -> Unit,
    onPreserveTransparencyChange: (Boolean) -> Unit,
    onApply: () -> Unit,
    onCancel: () -> Unit,
    isApplyEnabled: Boolean = true,
    modifier: Modifier = Modifier
) {
    Column(
        modifier = modifier
            .width(360.dp)
            .clip(RoundedCornerShape(28.dp))
            .background(theme.panelBg.copy(alpha = 0.95f))
            .border(1.2.dp, theme.panelStroke, RoundedCornerShape(28.dp))
            .padding(horizontal = 20.dp, vertical = 16.dp),
        verticalArrangement = Arrangement.spacedBy(16.dp)
    ) {
        Row(
            modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.SpaceBetween,
            verticalAlignment = Alignment.CenterVertically
        ) {
            Text(
                text = "[redacted]",
                color = theme.textPrimary,
                fontSize = 15.sp,
                fontWeight = FontWeight.Bold
            )

            Row(
                verticalAlignment = Alignment.CenterVertically,
                horizontalArrangement = Arrangement.spacedBy(12.dp)
```

```
            ) {
                Box(
                    modifier = Modifier
                        .size(36.dp)
                        .clip(CircleShape)
                        .background(theme.panelInset)
                        .clickable { onCancel() },
                    contentAlignment = Alignment.Center
                ) {
                    Icon(Icons.Default.Close, null, tint = theme.textPrimary, modifier = Modifier.size(20.dp))
                }

                Box(
                    modifier = Modifier
                        .size(36.dp)
                        .clip(CircleShape)
                        .background(if (isApplyEnabled) theme.accent else theme.panelInset)
                        .clickable(enabled = isApplyEnabled) { onApply() },
                    contentAlignment = Alignment.Center
                ) {
                    Icon(Icons.Default.Check, null, tint = Color.White, modifier = Modifier.size(20.dp))
                }
            }
        }

        Text(
            text = "[redacted], [redacted] — [redacted]",
            color = theme.textSecondary,
            fontSize = 12.sp,
            lineHeight = 16.sp
        )

        // Threshold
        Column(verticalArrangement = Arrangement.spacedBy(4.dp)) {
            Row(
                modifier = Modifier.fillMaxWidth(),
                horizontalArrangement = Arrangement.SpaceBetween
            ) {
                Text("[redacted]", color = theme.textSecondary, fontSize = 13.sp)
                Text("${(threshold * 255).roundToInt()}", color = theme.accent, fontSize = 13.sp, fontWeight = FontWeigh
    t.Bold)
            }
            Slider(
                value = threshold,
                onValueChange = onThresholdChange,
                valueRange = 0f..1f,
                colors = SliderDefaults.colors(
                    thumbColor = theme.accent,
                    activeTrackColor = theme.accent,
                    inactiveTrackColor = theme.panelInset
                )
            )
        }

        Row(
            modifier = Modifier.fillMaxWidth(),
            verticalAlignment = Alignment.CenterVertically,
            horizontalArrangement = Arrangement.SpaceBetween
        ) {
            Text(
                text = "[redacted]",
                color = theme.textPrimary,
                fontSize = 14.sp
            )
            Switch(
                checked = preserveTransparency,
                onCheckedChange = onPreserveTransparencyChange
            )
        }
    }
}
```

app/src/main/java/com/wetinknext/ui/components/Toolbars.kt

```
package com.wetinknext.ui.components

import androidx.compose.animation.core.Spring
import androidx.compose.animation.core.animateFloatAsState
import androidx.compose.animation.core.spring
import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.clickable
import androidx.compose.foundation.gestures.detectDragGestures
import androidx.compose.foundation.gestures.detectTapGestures
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.filled.Redo
import androidx.compose.material.icons.automirrored.filled.Undo
```

```
import androidx.compose.material.icons.filled.*
import androidx.compose.material3.Icon
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Brush
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.graphicsLayer
import androidx.compose.ui.graphics.vector.ImageVector
import androidx.compose.ui.input.pointer.pointerInput
import androidx.compose.ui.layout.onSizeChanged
import androidx.compose.ui.platform.LocalDensity
import androidx.compose.ui.res.vectorResource
import androidx.compose.ui.unit.IntOffset
import androidx.compose.ui.unit.dp
import com.wetinknext.R
import com.wetinknext.ui.theme.AppTheme

@Composable
fun TopToolbar(
    theme: AppTheme,
    currentColor: Color,
    canUndo: Boolean,
    canRedo: Boolean,
    isSelectionActive: Boolean,
    isTransformActive: Boolean,
    isAnimationActive: Boolean,
    isAdjustmentsActive: Boolean = false,
    onUndoClick: () -> Unit,
    onRedoClick: () -> Unit,
    onSelectionClick: () -> Unit,
    onTransformClick: () -> Unit,
    onAnimationClick: () -> Unit,
    onAdjustmentsClick: () -> Unit = {},
    onLayersClick: () -> Unit,
    onColorClick: () -> Unit,
) {
    Row(
        modifier = Modifier
            .padding(top = 12.dp, end = 12.dp)
            .clip(RoundedCornerShape(25.dp))
            .background(theme.panelBg.copy(alpha = 0.9f))
            .border(1.2.dp, theme.panelStroke, RoundedCornerShape(25.dp))
            .padding(horizontal = 8.dp, vertical = 6.dp),
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement = Arrangement.spacedBy(10.dp)
    ) {
        ToolbarIcon(Icons.AutoMirrored.Filled.Undo, theme, isSelected = false, enabled = canUndo, onClick = onUndoClick)
        ToolbarIcon(Icons.AutoMirrored.Filled.Redo, theme, isSelected = false, enabled = canRedo, onClick = onRedoClick)

        ToolbarIcon(Icons.Default.SelectAll, theme, isSelected = isSelectionActive, onClick = onSelectionClick)
        ToolbarIcon(Icons.Default.Movie, theme, isSelected = isAnimationActive, onClick = onAnimationClick)
        ToolbarIcon(Icons.Default.Fullscreen, theme, isSelected = isTransformActive, onClick = onTransformClick)
        ToolbarIcon(Icons.Default.Tune, theme, isSelected = isAdjustmentsActive, onClick = onAdjustmentsClick)
        ToolbarIcon(Icons.Default.Layers, theme, onClick = onLayersClick)

        ColorPickerDot(
            color = currentColor,
            theme = theme,
            onClick = onColorClick
        )
    }
}

@Composable
fun ColorPickerDot(
    color: Color,
    theme: AppTheme,
    onClick: () -> Unit
) {
    var isPressed by remember { mutableStateOf(false) }

    val scale by animateFloatAsState(
        targetValue = if (isPressed) 1.15f else 1f,
        animationSpec = spring(dampingRatio = Spring.DampingRatioMediumBouncy, stiffness = Spring.StiffnessLow),
        label = "scale"
    )

    Box(
        modifier = Modifier
            .graphicsLayer {
                scaleX = scale
                scaleY = scale
            }
            .size(30.dp)
            .clip(CircleShape)
            .background(color)
            .border(1.5.dp, theme.panelStroke, CircleShape)
            .clickable { onClick() }
    )
}
```

```
@Composable
fun SideToolbar(
    theme: AppTheme,
    brushSize: Float,
    brushOpacity: Float,
    isEraser: Boolean,
    onBrushSizeChange: (Float) -> Unit,
    onBrushOpacityChange: (Float) -> Unit,
    onBrushClick: () -> Unit,
    onEraserClick: (Boolean) -> Unit,
) {
    Column(
        modifier = Modifier
            .padding(start = 12.dp)
            .width(44.dp)
            .clip(RoundedCornerShape(22.dp))
            .background(theme.panelBg.copy(alpha = 0.9f))
            .border(1.2.dp, theme.panelStroke, RoundedCornerShape(22.dp))
            .padding(vertical = 10.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.spacedBy(12.dp)
    ) {
        ToolbarIcon(Icons.Default.Brush, theme, isSelected = !isEraser, onClick = {
            onEraserClick(false)
            onBrushClick()
        })
        ToolbarIcon(ImageVector.vectorResource(R.drawable.ic_eraser), theme, isSelected = isEraser, onClick = { onEraser
    Click(true) })

        Spacer(Modifier.height(4.dp))

        VerticalSlider(
            value = brushSize,
            onValueChange = onBrushSizeChange,
            valueRange = 2f..400f,
            theme = theme
        )

        VerticalSlider(
            value = brushOpacity,
            onValueChange = onBrushOpacityChange,
            valueRange = 0f..1f,
            theme = theme
        )
    }
}

@Composable
private fun VerticalSlider(
    value: Float,
    onValueChange: (Float) -> Unit,
    valueRange: ClosedFloatingPointRange<Float> = 0f..1f,
    theme: AppTheme,
    modifier: Modifier = Modifier
) {
    val range = valueRange.endInclusive - valueRange.start
    var trackHeightPx by remember { mutableFloatStateOf(1f) }
    val density = LocalDensity.current

    val norm = ((value - valueRange.start) / range).coerceIn(0f, 1f)

    fun updateFromY(y: Float) {
        val n = (1f - (y / trackHeightPx)).coerceIn(0f, 1f)
        onValueChange(valueRange.start + n * range)
    }

    Box(
        modifier = modifier
            .width(28.dp)
            .height(120.dp)
            .clip(RoundedCornerShape(14.dp))
            .background(theme.accentMuted.copy(alpha = 0.15f))
            .border(1.dp, theme.panelStroke, RoundedCornerShape(14.dp))
            .onSizeChanged { trackHeightPx = it.height.toFloat() }
            .pointerInput(valueRange) {
                detectTapGestures { offset -> updateFromY(offset.y) }
            }
            .pointerInput(valueRange) {
                detectDragGestures { change, _ ->
                    change.consume()
                    updateFromY(change.position.y)
                }
            }
    ) {
        Box(
            Modifier
                .align(Alignment.BottomCenter)
                .fillMaxWidth()
                .fillMaxHeight(norm)
                .background(Brush.verticalGradient(listOf(theme.accentSoft, theme.accent)))
        )
        Box(
            Modifier
```

```
                .align(Alignment.TopCenter)
                .offset {
                    val thumbSizePx = with(density) { 22.dp.toPx() }
                    IntOffset(0, ((1f - norm) * (trackHeightPx - thumbSizePx)).toInt())
                }
                .padding(2.dp)
                .size(22.dp)
                .clip(CircleShape)
                .background(theme.textPrimary)
                .border(1.dp, theme.panelStroke, CircleShape)
        )
    }
}

@Composable
private fun ToolbarIcon(
    icon: ImageVector,
    theme: AppTheme,
    isSelected: Boolean = false,
    enabled: Boolean = true,
    contentDescription: String? = null,
    onClick: () -> Unit = {}
) {
    Box(
        modifier = Modifier
            .size(34.dp)
            .clip(CircleShape)
            .background(if (isSelected) theme.accentMuted.copy(alpha = 0.3f) else Color.Transparent)
            .border(if (isSelected) 1.dp else 0.dp, if (isSelected) theme.accentSoft else Color.Transparent, CircleShape
    )
            .clickable(enabled = enabled) { onClick() },
        contentAlignment = Alignment.Center
    ) {
        Icon(
            icon,
            contentDescription = contentDescription,
            tint = if (!enabled) theme.iconInactive.copy(alpha = 0.5f) else if (isSelected) theme.accent else theme.text
    Primary,
            modifier = Modifier.size(20.dp)
        )
    }
}
```

app/src/main/java/com/wetinknext/ui/components/TransformMenuView.kt

```
package com.wetinknext.ui.components

import androidx.compose.foundation.BorderStroke
import androidx.compose.foundation.background
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.*
import androidx.compose.material3.*
import androidx.compose.runtime.Composable
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.draw.rotate
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.vector.ImageVector
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.wetinknext.ui.theme.AppTheme

enum class TransformModeUi {
    FREEFORM,
    UNIFORM,
    DISTORT,
    WARP
}

interface TransformActions {
    fun reset()
    fun cancel()
    fun apply()
    fun flipHorizontal()
    fun flipVertical()
    fun setMode(mode: TransformModeUi)
}

@Composable
fun TransformMenuView(
    theme: AppTheme,
```

```
    actions: TransformActions,
    currentMode: TransformModeUi,
    modifier: Modifier = Modifier
) {
    val panelBorder = BorderStroke(1.dp, theme.panelStroke)

    Column(
        modifier = modifier
            .fillMaxWidth()
            .navigationBarsPadding()
            .padding(horizontal = 16.dp, vertical = 20.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.spacedBy(12.dp)
    ) {
        // Row 1: Modes
        Surface(
            color = theme.panelBg.copy(alpha = 0.94f),
            shape = RoundedCornerShape(24.dp),
            border = panelBorder
        ) {
            Row(
                modifier = Modifier.padding(4.dp),
                horizontalArrangement = Arrangement.spacedBy(4.dp)
            ) {
                TransformModeButton("[redacted]", TransformModeUi.FREEFORM, currentMode, theme, actions::setMode)
                TransformModeButton("[redacted].", TransformModeUi.UNIFORM, currentMode, theme, actions::setMode)
                TransformModeButton("[redacted]", TransformModeUi.DISTORT, currentMode, theme, actions::setMode)
                TransformModeButton("[redacted].", TransformModeUi.WARP, currentMode, theme, actions::setMode)
            }
        }

        // Row 2: Actions
        Surface(
            color = theme.panelBg.copy(alpha = 0.94f),
            shape = RoundedCornerShape(24.dp),
            border = panelBorder
        ) {
            Row(
                modifier = Modifier.padding(horizontal = 4.dp, vertical = 4.dp),
                horizontalArrangement = Arrangement.spacedBy(4.dp),
                verticalAlignment = Alignment.CenterVertically
            ) {
                ActionIconButton(Icons.Default.Flip, "[redacted] [redacted]", theme, enabled = false) { actions.flipHorizontal() }
                ActionIconButton(Icons.Default.Flip, "[redacted] [redacted]", theme, iconModifier = Modifier.rotate(90f), enabled =
 false) { actions.flipVertical() }
                ActionIconButton(Icons.Default.RotateRight, "[redacted] 45", theme, enabled = false) { /* ... */ }
                ActionIconButton(Icons.Default.FitScreen, "[redacted]", theme, enabled = false) { actions.reset() }

                VerticalDivider(modifier = Modifier.height(24.dp).padding(horizontal = 4.dp), color = theme.panelStroke)

                // Placeholder for interpolation/snapping
                Box(modifier = Modifier.size(48.dp), contentAlignment = Alignment.Center) {
                    Icon(Icons.Default.LinkOff, null, tint = theme.textSecondary.copy(alpha = 0.4f), modifier = Modifier
.size(24.dp))
                }
            }
        }

        // Row 3: Main Controls
        Row(
            modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.spacedBy(16.dp, Alignment.CenterHorizontally),
            verticalAlignment = Alignment.CenterVertically
        ) {
            // Reset
            TransformRoundButton(Icons.Default.Refresh, "[redacted]", theme, enabled = false) { actions.reset() }

            // Cancel
            TransformRoundButton(Icons.Default.Close, "[redacted]", theme) { actions.cancel() }

            // Apply
            Surface(
                color = theme.accent.copy(alpha = 0.4f),
                shape = CircleShape,
                modifier = Modifier.size(48.dp)
            ) {
                Box(
                    modifier = Modifier.fillMaxSize(),
                    contentAlignment = Alignment.Center
                ) {
                    Icon(Icons.Default.Check, "[redacted]", tint = Color.White.copy(alpha = 0.4f))
                }
            }
        }
    }
}

@Composable
private fun TransformModeButton(
    label: String,
    mode: TransformModeUi,
    currentMode: TransformModeUi,
    theme: AppTheme,
```

```
    onClick: (TransformModeUi) -> Unit
) {
    val isSelected = mode == currentMode
    val enabled = false // Transform engine not ready
    Box(
        modifier = Modifier
            .clip(RoundedCornerShape(20.dp))
            .background(if (isSelected) theme.accent.copy(alpha = 0.4f) else Color.Transparent)
            .clickable(enabled = enabled) { onClick(mode) }
            .padding(horizontal = 14.dp, vertical = 8.dp),
        contentAlignment = Alignment.Center
    ) {
        Text(
            text = label,
            color = if (isSelected) Color.White.copy(alpha = 0.4f) else theme.textPrimary.copy(alpha = 0.4f),
            fontSize = 13.sp,
            fontWeight = if (isSelected) FontWeight.Bold else FontWeight.Normal
        )
    }
}

@Composable
private fun ActionIconButton(
    icon: ImageVector,
    description: String,
    theme: AppTheme,
    iconModifier: Modifier = Modifier,
    enabled: Boolean = true,
    onClick: () -> Unit
) {
    Box(
        modifier = Modifier
            .size(48.dp)
            .clickable(enabled = enabled) { onClick() },
        contentAlignment = Alignment.Center
    ) {
        Icon(
            imageVector = icon,
            contentDescription = description,
            tint = if (enabled) theme.textPrimary else theme.textPrimary.copy(alpha = 0.4f),
            modifier = iconModifier.size(24.dp)
        )
    }
}

@Composable
private fun TransformRoundButton(
    icon: ImageVector,
    description: String,
    theme: AppTheme,
    enabled: Boolean = true,
    onClick: () -> Unit
) {
    Surface(
        color = if (enabled) theme.panelBg.copy(alpha = 0.9f) else theme.panelBg.copy(alpha = 0.4f),
        shape = CircleShape,
        border = BorderStroke(1.dp, if (enabled) theme.panelStroke else theme.panelStroke.copy(alpha = 0.4f)),
        modifier = Modifier.size(48.dp)
    ) {
        Box(
            modifier = Modifier.fillMaxSize().clickable(enabled = enabled) { onClick() },
            contentAlignment = Alignment.Center
        ) {
            Icon(icon, description, tint = if (enabled) theme.textPrimary else theme.textPrimary.copy(alpha = 0.4f))
        }
    }
}
```

app/src/main/java/com/wetinknext/ui/home/HomeScreen.kt

```
package com.wetinknext.ui.home

import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.clickable
import androidx.compose.foundation.Image
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.lazy.grid.GridCells
import androidx.compose.foundation.lazy.grid.LazyVerticalGrid
import androidx.compose.foundation.lazy.grid.items
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.foundation.verticalScroll
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.outlined.Add
import androidx.compose.material.icons.outlined.Delete
import androidx.compose.material.icons.outlined.Edit
import androidx.compose.material.icons.outlined.MoreVert
import androidx.compose.material3.*
import androidx.compose.runtime.*
```

```
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.asImageBitmap
import androidx.compose.ui.graphics.graphicsLayer
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.wetinknext.ui.theme.AppTheme
import java.text.SimpleDateFormat
import java.util.Date
import java.util.Locale

// Minimal models for Home UI shell
data class HomeProject(
    val id: String,
    val name: String,
    val canvasWidth: Int,
    val canvasHeight: Int,
    val updatedAtMs: Long,
)

data class DeletedHomeProject(
    val project: HomeProject,
    val deletedAtMs: Long,
)

@Composable
fun HomeScreen(
    theme: AppTheme,
    projects: List<HomeProject>,
    deletedProjects: List<DeletedHomeProject> = emptyList(),
    onOpenProject: (HomeProject) -> Unit,
    onCreateProject: (name: String, width: Int, height: Int, dpi: Int, colorProfile: String) -> Unit,
    onRenameProject: (HomeProject, String) -> Unit,
    onDeleteProject: (HomeProject) -> Unit,
    onRestoreProject: (DeletedHomeProject) -> Unit,
    onDeleteForever: (DeletedHomeProject) -> Unit,
) {
    var showNewDialog by remember { mutableStateOf(false) }
    var renaming by remember { mutableStateOf<HomeProject?>(null) }
    var deleting by remember { mutableStateOf<HomeProject?>(null) }
    var showTrash by remember { mutableStateOf(false) }

    Box(
        modifier = Modifier
            .fillMaxSize()
            .background(theme.panelBgSolid),
    ) {
        Column(modifier = Modifier.fillMaxSize()) {
            HomeTopBar(
                theme = theme,
                total = projects.size,
                onCreate = { showNewDialog = true },
                onTrash = { showTrash = true },
            )
            LazyVerticalGrid(
                columns = GridCells.Adaptive(minSize = 180.dp),
                contentPadding = PaddingValues(16.dp),
                horizontalArrangement = Arrangement.spacedBy(12.dp),
                verticalArrangement = Arrangement.spacedBy(12.dp),
                modifier = Modifier.fillMaxSize(),
            ) {
                items(projects, key = { it.id }) { project ->
                    ProjectTile(
                        project = project,
                        theme = theme,
                        onOpen = { onOpenProject(project) },
                        onRename = { renaming = project },
                        onDelete = { deleting = project },
                    )
                }
            }
        }
    }

    if (showNewDialog) {
        NewCanvasDialog(
            theme = theme,
            defaultWidth = 2800,
            defaultHeight = 1840,
            onDismiss = { showNewDialog = false },
            onConfirm = { name, width, height, dpi, colorProfile ->
                showNewDialog = false
                onCreateProject(name, width, height, dpi, colorProfile)
            },
        )
    }

    renaming?.let { project ->
```

```
        RenameDialog(
            theme = theme,
            initial = project.name,
            onDismiss = { renaming = null },
            onConfirm = { newName ->
                onRenameProject(project, newName)
                renaming = null
            },
        )
    }

    deleting?.let { project ->
        AlertDialog(
            onDismissRequest = { deleting = null },
            confirmButton = {
                TextButton(
                    onClick = {
                        onDeleteProject(project)
                        deleting = null
                    },
                ) {
                    Text("[redacted]", color = theme.danger, fontWeight = FontWeight.SemiBold)
                }
            },
            dismissButton = {
                TextButton(onClick = { deleting = null }) {
                    Text("[redacted]", color = theme.textSecondary)
                }
            },
            title = { Text("[redacted] [redacted]?", color = theme.textPrimary) },
            text = { Text(project.name, color = theme.textSecondary) },
            containerColor = theme.panelBgSolid,
        )
    }

    if (showTrash) {
        TrashDialog(
            theme = theme,
            deletedProjects = deletedProjects,
            onRestore = { deleted ->
                onRestoreProject(deleted)
            },
            onDeleteForever = { deleted ->
                onDeleteForever(deleted)
            },
            onDismiss = { showTrash = false },
        )
    }
}

@Composable
private fun HomeTopBar(
    theme: AppTheme,
    total: Int,
    onCreate: () -> Unit,
    onTrash: () -> Unit,
) {
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(horizontal = 16.dp, vertical = 14.dp),
        verticalAlignment = Alignment.CenterVertically,
    ) {
        Column(modifier = Modifier.weight(1f)) {
            Text(
                "[redacted] [redacted]",
                color = theme.textPrimary,
                fontSize = 22.sp,
                fontWeight = FontWeight.SemiBold,
            )
            Spacer(Modifier.height(3.dp))
            Text(
                "[redacted]: $total",
                color = theme.textSecondary,
                fontSize = 13.sp,
                fontWeight = FontWeight.Medium,
            )
        }
        TopIconButton(theme = theme, onClick = onCreate) {
            Icon(Icons.Outlined.Add, null, tint = theme.textPrimary, modifier = Modifier.size(22.dp))
        }
        Spacer(Modifier.size(10.dp))
        TopIconButton(theme = theme, onClick = onTrash) {
            Icon(Icons.Outlined.Delete, null, tint = theme.textPrimary, modifier = Modifier.size(22.dp))
        }
    }
}

@Composable
private fun TopIconButton(
    theme: AppTheme,
    onClick: () -> Unit,
    content: @Composable () -> Unit,
```

```
) {
    Box(
        modifier = Modifier
            .size(44.dp)
            .clip(RoundedCornerShape(14.dp))
            .background(theme.panelBg)
            .border(1.dp, theme.panelStroke, RoundedCornerShape(14.dp))
            .clickable(onClick = onClick),
        contentAlignment = Alignment.Center,
    ) {
        content()
    }
}

@Composable
private fun ProjectTile(
    project: HomeProject,
    theme: AppTheme,
    onOpen: () -> Unit,
    onRename: () -> Unit,
    onDelete: () -> Unit,
) {
    var menuExpanded by remember { mutableStateOf(false) }

    Column(
        modifier = Modifier
            .fillMaxWidth()
            .clip(RoundedCornerShape(8.dp))
            .background(theme.panelBg)
            .border(1.2.dp, theme.panelStroke, RoundedCornerShape(8.dp))
            .clickable(onClick = onOpen)
            .padding(10.dp),
    ) {
        ProjectPreview(project = project, theme = theme)
        Spacer(Modifier.height(10.dp))
        Row(verticalAlignment = Alignment.CenterVertically) {
            Column(modifier = Modifier.weight(1f)) {
                Text(
                    project.name,
                    color = theme.textPrimary,
                    fontSize = 14.sp,
                    fontWeight = FontWeight.SemiBold,
                    maxLines = 1,
                    overflow = TextOverflow.Ellipsis,
                )
                Text(
                    "${project.canvasWidth} x ${project.canvasHeight} px · ${formatDate(project.updatedAtMs)}",
                    color = theme.textSecondary,
                    fontSize = 11.sp,
                    maxLines = 1,
                    overflow = TextOverflow.Ellipsis,
                )
            }
            Box {
                Box(
                    modifier = Modifier
                        .size(34.dp)
                        .clip(RoundedCornerShape(12.dp))
                        .clickable { menuExpanded = true },
                    contentAlignment = Alignment.Center,
                ) {
                    Icon(Icons.Outlined.MoreVert, null, tint = theme.textSecondary, modifier = Modifier.size(20.dp))
                }
                DropdownMenu(
                    expanded = menuExpanded,
                    onDismissRequest = { menuExpanded = false },
                    modifier = Modifier.background(theme.panelBg)
                ) {
                    DropdownMenuItem(
                        text = { Text("[redacted]", color = theme.textPrimary) },
                        leadingIcon = { Icon(Icons.Outlined.Edit, null, tint = theme.textSecondary) },
                        onClick = {
                            menuExpanded = false
                            onRename()
                        },
                    )
                    DropdownMenuItem(
                        text = { Text("[redacted]", color = theme.danger) },
                        leadingIcon = { Icon(Icons.Outlined.Delete, null, tint = theme.danger) },
                        onClick = {
                            menuExpanded = false
                            onDelete()
                        },
                    )
                }
            }
        }
    }
}

@Composable
private fun ProjectPreview(
    project: HomeProject,
```

```
    theme: AppTheme,
) {
    // Placeholder for actual preview loading
    Box(
        modifier = Modifier
            .fillMaxWidth()
            .aspectRatio(1.15f)
            .clip(RoundedCornerShape(6.dp))
            .background(theme.panelInset)
            .border(1.dp, theme.panelStroke, RoundedCornerShape(6.dp)),
        contentAlignment = Alignment.Center,
    ) {
        val ratio = project.canvasWidth.toFloat() / project.canvasHeight.coerceAtLeast(1).toFloat()
        val swatchModifier = if (ratio >= 1f) {
            Modifier
                .fillMaxWidth(0.74f)
                .aspectRatio(ratio)
        } else {
            Modifier
                .fillMaxWidth((0.54f * ratio).coerceIn(0.24f, 0.54f))
                .aspectRatio(ratio)
        }

        Box(
            modifier = swatchModifier
                .clip(RoundedCornerShape(4.dp))
                .background(Color(0xFFFAFAFA))
                .border(1.dp, theme.panelStroke, RoundedCornerShape(4.dp)),
        )
    }
}

@Composable
private fun RenameDialog(
    theme: AppTheme,
    initial: String,
    onDismiss: () -> Unit,
    onConfirm: (String) -> Unit,
) {
    var name by remember(initial) { mutableStateOf(initial) }
    AlertDialog(
        onDismissRequest = onDismiss,
        confirmButton = {
            TextButton(
                onClick = {
                    val cleaned = name.trim()
                    if (cleaned.isNotEmpty()) onConfirm(cleaned)
                },
            ) {
                Text("[redacted]", color = theme.accent, fontWeight = FontWeight.SemiBold)
            }
        },
        dismissButton = {
            TextButton(onClick = onDismiss) {
                Text("[redacted]", color = theme.textSecondary)
            }
        },
        title = { Text("[redacted]", color = theme.textPrimary) },
        text = {
            OutlinedTextField(
                value = name,
                onValueChange = { name = it.take(48) },
                singleLine = true,
                colors = OutlinedTextFieldDefaults.colors(
                    focusedBorderColor = theme.accent,
                    unfocusedBorderColor = theme.panelStroke,
                    focusedTextColor = theme.textPrimary,
                    unfocusedTextColor = theme.textPrimary
                )
            )
        },
        containerColor = theme.panelBgSolid,
    )
}

@Composable
private fun TrashDialog(
    theme: AppTheme,
    deletedProjects: List<DeletedHomeProject>,
    onRestore: (DeletedHomeProject) -> Unit,
    onDeleteForever: (DeletedHomeProject) -> Unit,
    onDismiss: () -> Unit,
) {
    AlertDialog(
        onDismissRequest = onDismiss,
        confirmButton = {
            TextButton(onClick = onDismiss) {
                Text("[redacted]", color = theme.accent, fontWeight = FontWeight.SemiBold)
            }
        },
        title = { Text("[redacted]", color = theme.textPrimary) },
        text = {
            if (deletedProjects.isEmpty()) {
```

```
                Text("[redacted] [redacted] [redacted] [redacted] 3 [redacted].", color = theme.textSecondary, fontSize = 13.sp)
            } else {
                Column(
                    modifier = Modifier
                        .fillMaxWidth()
                        .height(360.dp)
                        .verticalScroll(rememberScrollState()),
                    verticalArrangement = Arrangement.spacedBy(10.dp),
                ) {
                    Text(
                        "[redacted] [redacted] 3 [redacted].",
                        color = theme.textSecondary,
                        fontSize = 12.sp,
                    )
                    deletedProjects.sortedByDescending { it.deletedAtMs }.forEach { deleted ->
                        DeletedProjectRow(
                            deleted = deleted,
                            theme = theme,
                            onRestore = { onRestore(deleted) },
                            onDeleteForever = { onDeleteForever(deleted) },
                        )
                    }
                }
            }
        },
        containerColor = theme.panelBgSolid,
    )
}

@Composable
private fun DeletedProjectRow(
    deleted: DeletedHomeProject,
    theme: AppTheme,
    onRestore: () -> Unit,
    onDeleteForever: () -> Unit,
) {
    val project = deleted.project
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .clip(RoundedCornerShape(12.dp))
            .background(theme.panelBg)
            .border(1.dp, theme.panelStroke, RoundedCornerShape(12.dp))
            .padding(10.dp),
        verticalAlignment = Alignment.CenterVertically,
    ) {
        Box(
            modifier = Modifier
                .size(54.dp)
                .clip(RoundedCornerShape(6.dp))
                .background(theme.panelInset)
                .border(1.dp, theme.panelStroke, RoundedCornerShape(6.dp)),
            contentAlignment = Alignment.Center,
        ) {
            Box(
                modifier = Modifier
                    .fillMaxWidth(0.5f)
                    .aspectRatio(project.canvasWidth.toFloat() / project.canvasHeight.coerceAtLeast(1).toFloat())
                    .background(Color(0xFFFAFAFA)),
            )
        }
        Column(
            modifier = Modifier
                .weight(1f)
                .padding(start = 10.dp),
        ) {
            Text(
                project.name,
                color = theme.textPrimary,
                fontSize = 13.sp,
                fontWeight = FontWeight.SemiBold,
                maxLines = 1,
                overflow = TextOverflow.Ellipsis,
            )
            Text(
                "[redacted]: ${trashDaysLeft(deleted.deletedAtMs)} [redacted].",
                color = theme.textSecondary,
                fontSize = 11.sp,
            )
            Row(horizontalArrangement = Arrangement.spacedBy(8.dp)) {
                TextButton(onClick = onRestore) {
                    Text("[redacted]", color = theme.accent, fontSize = 12.sp)
                }
                TextButton(onClick = onDeleteForever) {
                    Text("[redacted]", color = theme.textSecondary, fontSize = 12.sp)
                }
            }
        }
    }
}

private fun formatDate(timeMs: Long): String =
    SimpleDateFormat("d MMM, HH:mm", Locale.forLanguageTag("ru")).format(Date(timeMs))
```

```
private fun trashDaysLeft(deletedAtMs: Long): Int {
    val retentionMs = 3L * 24L * 60L * 60L * 1000L
    val remainingMs = (retentionMs - (System.currentTimeMillis() - deletedAtMs)).coerceAtLeast(0L)
    return ((remainingMs + 24L * 60L * 60L * 1000L - 1L) / (24L * 60L * 60L * 1000L)).toInt().coerceAtLeast(1)
}
```

app/src/main/java/com/wetinknext/ui/home/NewCanvasDialog.kt

```
package com.wetinknext.ui.home

import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.clickable
import androidx.compose.foundation.horizontalScroll
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.foundation.text.KeyboardOptions
import androidx.compose.material3.*
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.input.KeyboardType
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.wetinknext.ui.theme.AppTheme
import kotlin.math.max

private data class CanvasPreset(
    val label: String,
    val width: Int,
    val height: Int,
    val description: String,
)

private val QuickPresets = listOf(
    CanvasPreset("[redacted]", 2800, 1840, "2800 x 1840 px"),
    CanvasPreset("[redacted]", 2048, 2048, "2048 x 2048 px"),
    CanvasPreset("4K", 4096, 2160, "4096 x 2160 px"),
    CanvasPreset("A4", 2480, 3508, "210 x 297 [redacted]"),
    CanvasPreset("1080P", 1920, 1080, "1920 x 1080 px"),
    CanvasPreset("9:16", 1080, 1920, "1080 x 1920 px"),
)

private val DpiOptions = listOf(72, 144, 300, 350, 600, 1200)
private val ColorProfiles = listOf("sRGB", "Display P3")

@Composable
fun NewCanvasDialog(
    theme: AppTheme,
    defaultWidth: Int,
    defaultHeight: Int,
    onDismiss: () -> Unit,
    onConfirm: (name: String, width: Int, height: Int, dpi: Int, colorProfile: String) -> Unit,
) {
    val initialIndex = QuickPresets.indexOfFirst {
        it.width == defaultWidth && it.height == defaultHeight
    }
    var selected by remember { mutableIntStateOf(initialIndex) }
    var customW by remember { mutableStateOf(defaultWidth.toString()) }
    var customH by remember { mutableStateOf(defaultHeight.toString()) }
    var name by remember { mutableStateOf("[redacted] [redacted]") }
    var dpi by remember { mutableIntStateOf(300) }
    var colorProfile by remember { mutableStateOf("sRGB") }
    var dpiExpanded by remember { mutableStateOf(false) }
    var profileExpanded by remember { mutableStateOf(false) }
    val isCustom = selected !in QuickPresets.indices

    AlertDialog(
        onDismissRequest = onDismiss,
        title = {
            Text("[redacted] [redacted]", color = theme.textPrimary, fontWeight = FontWeight.SemiBold)
        },
        text = {
            Column {
                OutlinedTextField(
                    value = name,
                    onValueChange = { name = it.take(48) },
                    label = { Text("[redacted]") },
                    singleLine = true,
                    modifier = Modifier.fillMaxWidth(),
                    colors = OutlinedTextFieldDefaults.colors(
                        focusedBorderColor = theme.accent,
                        unfocusedBorderColor = theme.panelStroke,
                        focusedLabelColor = theme.accent,
                        unfocusedLabelColor = theme.textSecondary
```

```
            )
        )
        Spacer(Modifier.height(12.dp))
        Text("[redacted]", color = theme.textSecondary, fontSize = 13.sp)
        Spacer(Modifier.height(8.dp))
        Row(
            horizontalArrangement = Arrangement.spacedBy(8.dp),
            modifier = Modifier.horizontalScroll(rememberScrollState()),
        ) {
            QuickPresets.forEachIndexed { index, preset ->
                QuickPresetCard(
                    preset = preset,
                    selected = selected == index,
                    theme = theme,
                    onClick = {
                        selected = index
                        customW = preset.width.toString()
                        customH = preset.height.toString()
                    },
                )
            }
        }

        Spacer(Modifier.height(12.dp))
        TextButton(onClick = { selected = -1 }) {
            Text(
                "[redacted] [redacted]",
                color = if (isCustom) theme.accent else theme.textSecondary,
                fontWeight = if (isCustom) FontWeight.SemiBold else FontWeight.Normal,
                fontSize = 13.sp,
            )
        }

        if (isCustom) {
            Spacer(Modifier.height(4.dp))
            Row(verticalAlignment = Alignment.CenterVertically) {
                OutlinedTextField(
                    value = customW,
                    onValueChange = { v -> customW = v.filter { it.isDigit() }.take(5) },
                    label = { Text("[redacted]") },
                    singleLine = true,
                    keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number),
                    modifier = Modifier.weight(1f),
                    colors = OutlinedTextFieldDefaults.colors(
                        focusedBorderColor = theme.accent,
                        unfocusedBorderColor = theme.panelStroke,
                        focusedLabelColor = theme.accent,
                        unfocusedLabelColor = theme.textSecondary
                    )
                )
                Spacer(Modifier.width(8.dp))
                Text("x", color = theme.textSecondary)
                Spacer(Modifier.width(8.dp))
                OutlinedTextField(
                    value = customH,
                    onValueChange = { v -> customH = v.filter { it.isDigit() }.take(5) },
                    label = { Text("[redacted]") },
                    singleLine = true,
                    keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number),
                    modifier = Modifier.weight(1f),
                    colors = OutlinedTextFieldDefaults.colors(
                        focusedBorderColor = theme.accent,
                        unfocusedBorderColor = theme.panelStroke,
                        focusedLabelColor = theme.accent,
                        unfocusedLabelColor = theme.textSecondary
                    )
                )
            }
        }

        Spacer(Modifier.height(14.dp))
        OptionRow(label = "DPI", value = "$dpi dpi", theme = theme, expanded = dpiExpanded, onExpand = { dpiExpa
nded = true }) {
            DpiOptions.forEach { option ->
                DropdownMenuItem(
                    text = { Text("$option dpi") },
                    onClick = {
                        dpi = option
                        dpiExpanded = false
                    },
                )
            }
        }
        OptionRow(label = "[redacted] [redacted]", value = colorProfile, theme = theme, expanded = profileExpanded, o
nExpand = { profileExpanded = true }) {
            ColorProfiles.forEach { option ->
                DropdownMenuItem(
                    text = { Text(option) },
                    onClick = {
                        colorProfile = option
                        profileExpanded = false
                    },
                )
```

```
                    }
                }
            }
        },
        confirmButton = {
            TextButton(
                onClick = {
                    val (w, h) = if (isCustom) {
                        val cw = customW.toIntOrNull()?.coerceIn(MinDimension, MaxDimension) ?: return@TextButton
                        val ch = customH.toIntOrNull()?.coerceIn(MinDimension, MaxDimension) ?: return@TextButton
                        cw to ch
                    } else {
                        val preset = QuickPresets[selected]
                        preset.width to preset.height
                    }
                    onConfirm(name.trim().ifBlank { "■■■■■ ■■■■■" }, w, h, dpi, colorProfile)
                },
            ) {
                Text("■■■■■■■■", color = theme.accent, fontWeight = FontWeight.SemiBold)
            }
        },
        dismissButton = {
            TextButton(onClick = onDismiss) {
                Text("■■■■■■■", color = theme.textSecondary)
            }
        },
        containerColor = theme.panelBgSolid,
    )
}

@Composable
private fun OptionRow(
    label: String,
    value: String,
    theme: AppTheme,
    expanded: Boolean,
    onExpand: () -> Unit,
    menu: @Composable () -> Unit,
) {
    Row(verticalAlignment = Alignment.CenterVertically, modifier = Modifier.fillMaxWidth()) {
        Text(label, color = theme.textPrimary, fontSize = 13.sp)
        Spacer(Modifier.weight(1f))
        Box {
            TextButton(onClick = onExpand) {
                Text(value, color = theme.accent, fontWeight = FontWeight.SemiBold)
            }
            DropdownMenu(
                expanded = expanded,
                onDismissRequest = { /* onExpand is handled by parent */ },
                modifier = Modifier.background(theme.panelBg)
            ) {
                menu()
            }
        }
    }
}

@Composable
private fun QuickPresetCard(
    preset: CanvasPreset,
    selected: Boolean,
    theme: AppTheme,
    onClick: () -> Unit,
) {
    Column(
        horizontalAlignment = Alignment.CenterHorizontally,
        modifier = Modifier
            .width(100.dp)
            .clip(RoundedCornerShape(12.dp))
            .background(if (selected) theme.accent.copy(alpha = 0.15f) else theme.panelBg)
            .border(1.5.dp, if (selected) theme.accent else theme.panelStroke, RoundedCornerShape(12.dp))
            .clickable(onClick = onClick)
            .padding(vertical = 12.dp, horizontal = 8.dp),
    ) {
        PresetSwatch(preset.width, preset.height, selected, theme)
        Spacer(Modifier.height(8.dp))
        Text(preset.label, color = theme.textPrimary, fontWeight = FontWeight.Medium, fontSize = 13.sp, textAlign = Text
    Align.Center)
        Text(preset.description, color = theme.textSecondary, fontSize = 10.sp, textAlign = TextAlign.Center, maxLines =
     1)
    }
}

@Composable
private fun PresetSwatch(width: Int, height: Int, selected: Boolean, theme: AppTheme) {
    val scale = 40f / max(width.coerceAtLeast(1), height.coerceAtLeast(1))
    val previewW = (width * scale).coerceAtLeast(8f)
    val previewH = (height * scale).coerceAtLeast(8f)
    Box(modifier = Modifier.size(48.dp), contentAlignment = Alignment.Center) {
        Box(
            modifier = Modifier
                .width(previewW.dp)
                .height(previewH.dp)
```

```
                .clip(RoundedCornerShape(4.dp))
                .background(if (selected) theme.accent else Color(0xFF5A5A60))
                .border(1.dp, if (selected) Color.White else theme.panelStroke, RoundedCornerShape(4.dp)),
        )
    }
}

private const val MinDimension = 256
private const val MaxDimension = 8192
```

app/src/main/java/com/wetinknext/ui/state/EditorActions.kt

```
package com.wetinknext.ui.state

/**
 * Interface defining all possible user actions from the UI that affect the paint engine state.
 */
interface EditorActions {
    fun undo()
    fun redo()
    fun setBrushSize(px: Float)
    fun setBrushOpacity(opacity: Float)

    fun addLayer()
    fun removeLayer(id: Long)
    fun setActiveLayer(id: Long)
    fun setLayerVisible(id: Long, visible: Boolean)
    fun setLayerOpacity(id: Long, opacity: Float)
}
```

app/src/main/java/com/wetinknext/ui/state/LayerState.kt

```
package com.wetinknext.ui.state

import androidx.compose.runtime.*
import com.wetinknext.engine.core.EditorUiState

class LayerItem(
    val id: Long,
    name: String,
    visible: Boolean,
    opacity: Float,
    locked: Boolean,
) {
    var name by mutableStateOf(name)
    var visible by mutableStateOf(visible)
    var opacity by mutableFloatStateOf(opacity)
    var locked by mutableStateOf(locked)
}

class LayerState {
    val layers = mutableStateListOf<LayerItem>()
    var activeLayerId by mutableStateOf(-1L)

    fun syncFromEditor(state: EditorUiState) {
        val ids = state.layers.map { it.id }.toSet()
        layers.removeAll { it.id !in ids }

        state.layers.forEachIndexed { index, model ->
            val item = layers.firstOrNull { it.id == model.id }

            if (item == null) {
                layers.add(
                    index.coerceAtMost(layers.size),
                    LayerItem(
                        id = model.id,
                        name = model.name,
                        visible = model.isVisible,
                        opacity = model.opacity,
                        locked = model.isLocked,
                    ),
                )
            } else {
                item.name = model.name
                item.visible = model.isVisible
                item.opacity = model.opacity
                item.locked = model.isLocked

                // If the item needs to be moved to match the index in state
                val currentIndex = layers.indexOf(item)
                if (currentIndex != index && index < layers.size) {
                    layers.removeAt(currentIndex)
                    layers.add(index, item)
                }
            }
        }
```

```
        activeLayerId = state.activeLayerId
    }
}
```

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

```
package com.wetinknext.ui.theme

import androidx.compose.runtime.Immutable
import androidx.compose.ui.graphics.Color

@Immutable
data class AppTheme(
    val id: String,
    val displayName: String,
    val isLight: Boolean,

    // Core Layout
    val appBg: Color,
    val gridLine: Color,
    val panelBg: Color,
    val panelBgSolid: Color,
    val panelStroke: Color,
    val panelInset: Color,
    val panelInsetSoft: Color,

    // Canvas
    val canvasBackdrop: Color,
    val canvasGrid: Color,

    // Typography & Icons
    val textPrimary: Color,
    val textSecondary: Color,
    val iconInactive: Color,

    // Accents
    val accent: Color,
    val accentSoft: Color,
    val accentMuted: Color,
    val danger: Color
) {
    val panelBgVariant: Color get() = panelInset
}

object AppThemes {
    val Dark = AppTheme(
        id = "dark", displayName = "Dark", isLight = false,
        appBg = Color(0xFF1A0C12), gridLine = Color(0xFF2A1418),
        panelBg = Color(0xFF2A141C), panelBgSolid = Color(0xFF2D1620),
        panelStroke = Color(0xFF3D1F2A), panelInset = Color(0xFF1F1015), panelInsetSoft = Color(0xFF2A1820),
        canvasBackdrop = Color(0xFF1A0C12), canvasGrid = Color(0xFF3A1C24),
        textPrimary = Color(0xFFF5E8EC), textSecondary = Color(0xFF9A7F8A), iconInactive = Color(0xFFC8B0B8),
        accent = Color(0xFFFF4D7A), accentSoft = Color(0xFFFF7AA0), accentMuted = Color(0x55FF4D7A), danger = Color(0xFF
    FF3B5C)
    )

    val Light = AppTheme(
        id = "light", displayName = "Light", isLight = true,
        appBg = Color(0xFFFFE4E9), gridLine = Color(0xFFF5D0D7),
        panelBg = Color(0xFFFFF1F4), panelBgSolid = Color(0xFFFFFFFF),
        panelStroke = Color(0xFFF0C8D0), panelInset = Color(0xFFFFE0E6), panelInsetSoft = Color(0xFFFFD0DA),
        canvasBackdrop = Color(0xFFFFE4E9), canvasGrid = Color(0xFFF5D0D7),
        textPrimary = Color(0xFF3D1B26), textSecondary = Color(0xFF8A5A6A), iconInactive = Color(0xFFB07A88),
        accent = Color(0xFFFF4D7A), accentSoft = Color(0xFFFFA0B7), accentMuted = Color(0x33FF4D7A), danger = Color(0xFF
    FF3B5C)
    )

    val Red = AppTheme(
        id = "red", displayName = "Red", isLight = false,
        appBg = Color(0xFF1A0606), gridLine = Color(0xFF2A0A0A),
        panelBg = Color(0xFF2A0C0C), panelBgSolid = Color(0xFF2D0E0E),
        panelStroke = Color(0xFF3D1414), panelInset = Color(0xFF1F0808), panelInsetSoft = Color(0xFF2A1010),
        canvasBackdrop = Color(0xFF1A0606), canvasGrid = Color(0xFF3A1414),
        textPrimary = Color(0xFFF5E8E8), textSecondary = Color(0xFF9A7F7F), iconInactive = Color(0xFFC8B0B0),
        accent = Color(0xFFFF3D3D), accentSoft = Color(0xFFFF7A7A), accentMuted = Color(0x55FF3D3D), danger = Color(0xFF
    FF8888)
    )

    val Vampire = AppTheme(
        id = "vampire", displayName = "Vampire", isLight = false,
        appBg = Color(0xFF1B1C1B), gridLine = Color(0xFF252525),
        panelBg = Color(0xFF262626), panelBgSolid = Color(0xFF2D2D2D),
        panelStroke = Color(0xFF4B2529), panelInset = Color(0xFF171717), panelInsetSoft = Color(0xFF2A1B1E),
        canvasBackdrop = Color(0xFF1B1C1B), canvasGrid = Color(0xFF2B2B2B),
        textPrimary = Color(0xFFF2ECEC), textSecondary = Color(0xFFCDA2A6), iconInactive = Color(0xFFE15761),
        accent = Color(0xFFFF3547), accentSoft = Color(0xFFFFA3A8), accentMuted = Color(0xAAFF3547), danger = Color(0xFF
    FF4A5F)
    )

    val Olive = AppTheme(
```

```
    id = "olive", displayName = "Olive", isLight = false,
    appBg = Color(0xFF171B0D), gridLine = Color(0xFF252A16),
    panelBg = Color(0xFF202711), panelBgSolid = Color(0xFF263014),
    panelStroke = Color(0xFF566032), panelInset = Color(0xFF121806), panelInsetSoft = Color(0xFF2B3418),
    canvasBackdrop = Color(0xFF171B0D), canvasGrid = Color(0xFF2F351C),
    textPrimary = Color(0xFFF1E7D0), textSecondary = Color(0xFFC7A77E), iconInactive = Color(0xFFE3A15E),
    accent = Color(0xFFFF982F), accentSoft = Color(0xFFFFC46A), accentMuted = Color(0x88FF982F), danger = Color(0xFF
FF624D)
)

val Blue = AppTheme(
    id = "blue", displayName = "Blue", isLight = false,
    appBg = Color(0xFF06101A), gridLine = Color(0xFF0A1828),
    panelBg = Color(0xFF0C1A2A), panelBgSolid = Color(0xFF0E1D2E),
    panelStroke = Color(0xFF14283D), panelInset = Color(0xFF08141F), panelInsetSoft = Color(0xFF101F2A),
    canvasBackdrop = Color(0xFF06101A), canvasGrid = Color(0xFF14283A),
    textPrimary = Color(0xFFE8EEF5), textSecondary = Color(0xFF7F8C9A), iconInactive = Color(0xFFB0BBC8),
    accent = Color(0xFF4D7AFF), accentSoft = Color(0xFF7AA0FF), accentMuted = Color(0x554D7AFF), danger = Color(0xFF
FF3B5C)
)

val Studio = AppTheme(
    id = "studio", displayName = "Studio", isLight = true,
    appBg = Color(0xFFE1E1E1), gridLine = Color(0xFFD1D6DA),
    panelBg = Color(0xFFF0F2F4), panelBgSolid = Color(0xFFFBFCFD),
    panelStroke = Color(0xFFD4DAE0), panelInset = Color(0xFFE7EBEF), panelInsetSoft = Color(0xFFDCE8F3),
    canvasBackdrop = Color(0xFFDADADA), canvasGrid = Color(0xFFCFCFCF),
    textPrimary = Color(0xFF20252A), textSecondary = Color(0xFF65737F), iconInactive = Color(0xFF5C8DB3),
    accent = Color(0xFF4A9AD8), accentSoft = Color(0xFF7FBCEB), accentMuted = Color(0x334A9AD8), danger = Color(0xFF
E65C73)
)

val Neon = AppTheme(
    id = "neon", displayName = "Neon", isLight = false,
    appBg = Color(0xFF2B123E), gridLine = Color(0xFF32184A),
    panelBg = Color(0xFF4A2366), panelBgSolid = Color(0xFF522A70),
    panelStroke = Color(0xFF6C3A8D), panelInset = Color(0xFF321344), panelInsetSoft = Color(0xFF3C1853),
    canvasBackdrop = Color(0xFF2B123E), canvasGrid = Color(0xFF3C1D55),
    textPrimary = Color(0xFFF3EAFE), textSecondary = Color(0xFFC6A8D8), iconInactive = Color(0xFF2FE0EA),
    accent = Color(0xFF25DDE8), accentSoft = Color(0xFF73F3FA), accentMuted = Color(0x5525DDE8), danger = Color(0xFF
FF6E9C)
)

val Cyan = AppTheme(
    id = "cyan", displayName = "Cyan", isLight = false,
    appBg = Color(0xFF06181A), gridLine = Color(0xFF0A2628),
    panelBg = Color(0xFF0C282A), panelBgSolid = Color(0xFF0E2C2E),
    panelStroke = Color(0xFF143C3D), panelInset = Color(0xFF081F20), panelInsetSoft = Color(0xFF102A2A),
    canvasBackdrop = Color(0xFF06181A), canvasGrid = Color(0xFF143A3A),
    textPrimary = Color(0xFFE8F5F5), textSecondary = Color(0xFF7F9A9A), iconInactive = Color(0xFFB0C8C8),
    accent = Color(0xFF26D9D9), accentSoft = Color(0xFF7AE0E0), accentMuted = Color(0x5526D9D9), danger = Color(0xFF
FF3B5C)
)

val Green = AppTheme(
    id = "green", displayName = "Green", isLight = false,
    appBg = Color(0xFF071A0C), gridLine = Color(0xFF0A2814),
    panelBg = Color(0xFF0C2A14), panelBgSolid = Color(0xFF0E2D16),
    panelStroke = Color(0xFF143D1F), panelInset = Color(0xFF081F0C), panelInsetSoft = Color(0xFF102A14),
    canvasBackdrop = Color(0xFF071A0C), canvasGrid = Color(0xFF143A1F),
    textPrimary = Color(0xFFE8F5EC), textSecondary = Color(0xFF7F9A88), iconInactive = Color(0xFFB0C8B8),
    accent = Color(0xFF3DDB6E), accentSoft = Color(0xFF7AE0A0), accentMuted = Color(0x553DDB6E), danger = Color(0xFF
FF3B5C)
)

val Yellow = AppTheme(
    id = "yellow", displayName = "Yellow", isLight = true,
    appBg = Color(0xFFFFF8E0), gridLine = Color(0xFFF5E5A0),
    panelBg = Color(0xFFFFF4D0), panelBgSolid = Color(0xFFFFFCF0),
    panelStroke = Color(0xFFE6CC60), panelInset = Color(0xFFFFEFB8), panelInsetSoft = Color(0xFFFFE890),
    canvasBackdrop = Color(0xFFFFF8E0), canvasGrid = Color(0xFFF0DA80),
    textPrimary = Color(0xFF3D2E08), textSecondary = Color(0xFF8A7A40), iconInactive = Color(0xFFB09860),
    accent = Color(0xFFF5B800), accentSoft = Color(0xFFFFD050), accentMuted = Color(0x55F5B800), danger = Color(0xFF
FF3B5C)
)

val Orange = AppTheme(
    id = "orange", displayName = "Orange", isLight = true,
    appBg = Color(0xFFFFE8D5), gridLine = Color(0xFFF5C898),
    panelBg = Color(0xFFFFE0C5), panelBgSolid = Color(0xFFFFF1E5),
    panelStroke = Color(0xFFE69C58), panelInset = Color(0xFFFFD8B5), panelInsetSoft = Color(0xFFFFC890),
    canvasBackdrop = Color(0xFFFFE8D5), canvasGrid = Color(0xFFF0BC80),
    textPrimary = Color(0xFF3D1E08), textSecondary = Color(0xFF8A5A30), iconInactive = Color(0xFFB07A50),
    accent = Color(0xFFFF8800), accentSoft = Color(0xFFFFB050), accentMuted = Color(0x55FF8800), danger = Color(0xFF
FF3B5C)
)

val Gray = AppTheme(
    id = "gray", displayName = "Gray", isLight = false,
    appBg = Color(0xFF1A1A1A), gridLine = Color(0xFF2A2A2A),
    panelBg = Color(0xFF2A2A2A), panelBgSolid = Color(0xFF303030),
    panelStroke = Color(0xFF404040), panelInset = Color(0xFF1F1F1F), panelInsetSoft = Color(0xFF2B2B2B),
    canvasBackdrop = Color(0xFF1A1A1A), canvasGrid = Color(0xFF3A3A3A),
```

```
        textPrimary = Color(0xFFF0F0F0), textSecondary = Color(0xFF909090), iconInactive = Color(0xFFB8B8B8),
        accent = Color(0xFFB0B0B0), accentSoft = Color(0xFFD0D0D0), accentMuted = Color(0x55B0B0B0), danger = Color(0xFF
    FF3B5C)
    )

    val White = AppTheme(
        id = "white", displayName = "White", isLight = true,
        appBg = Color(0xFFFFFFFF), gridLine = Color(0xFFE8E8E8),
        panelBg = Color(0xFFF5F5F5), panelBgSolid = Color(0xFFFFFFFF),
        panelStroke = Color(0xFFD8D8D8), panelInset = Color(0xFFEEEEEE), panelInsetSoft = Color(0xFFE0E0E0),
        canvasBackdrop = Color(0xFFFFFFFF), canvasGrid = Color(0xFFE0E0E0),
        textPrimary = Color(0xFF202020), textSecondary = Color(0xFF707070), iconInactive = Color(0xFF989898),
        accent = Color(0xFF606060), accentSoft = Color(0xFF888888), accentMuted = Color(0x55606060), danger = Color(0xFF
    FF3B5C)
    )

    val all = listOf(
        Dark, Light, Red, Vampire, Olive, Blue, Studio, Neon, Cyan, Green, Yellow, Orange, Gray, White
    )

    val default = Gray
}

enum class UiFont { Classic, Italic }
```

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

```
package com.wetinknext.ui.theme

import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Typography
import androidx.compose.material3.darkColorScheme
import androidx.compose.material3.lightColorScheme
import androidx.compose.runtime.Composable
import androidx.compose.runtime.CompositionLocalProvider
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.text.TextStyle
import androidx.compose.ui.text.font.FontFamily
import androidx.compose.ui.text.font.FontStyle

@Composable
fun WetInkTheme(
    theme: AppTheme,
    fontMode: UiFont,
    content: @Composable () -> Unit
) {
    val fontStyle = if (fontMode == UiFont.Italic) FontStyle.Italic else FontStyle.Normal
    val fontFamily = if (fontMode == UiFont.Italic) FontFamily.Serif else FontFamily.SansSerif

    val typography = Typography(
        bodyLarge = TextStyle(fontStyle = fontStyle, fontFamily = fontFamily),
        bodyMedium = TextStyle(fontStyle = fontStyle, fontFamily = fontFamily),
        bodySmall = TextStyle(fontStyle = fontStyle, fontFamily = fontFamily),
        titleLarge = TextStyle(fontStyle = fontStyle, fontFamily = fontFamily),
        titleMedium = TextStyle(fontStyle = fontStyle, fontFamily = fontFamily),
        titleSmall = TextStyle(fontStyle = fontStyle, fontFamily = fontFamily),
        labelLarge = TextStyle(fontStyle = fontStyle, fontFamily = fontFamily),
        labelMedium = TextStyle(fontStyle = fontStyle, fontFamily = fontFamily),
        labelSmall = TextStyle(fontStyle = fontStyle, fontFamily = fontFamily),
    )

    val scheme = if (!theme.isLight) {
        darkColorScheme(
            primary = theme.accent,
            onPrimary = theme.textPrimary,
            background = theme.appBg, // ■■■■ ■■■■■
            onBackground = theme.textPrimary,
            surface = theme.panelBg, // ■■■■ ■■■■■■■
            onSurface = theme.textPrimary,
            surfaceVariant = theme.panelBgVariant,
            onSurfaceVariant = theme.textSecondary,
        )
    } else {
        lightColorScheme(
            primary = theme.accent,
            onPrimary = Color.White,
            background = theme.appBg,
            onBackground = theme.textPrimary,
            surface = theme.panelBg,
            onSurface = theme.textPrimary,
            surfaceVariant = theme.panelBgVariant,
            onSurfaceVariant = theme.textSecondary,
        )
    }

    CompositionLocalProvider(LocalAppTheme provides theme) {
        MaterialTheme(
            colorScheme = scheme,
            typography = typography,
            content = content,
```

```
        )
    }
}
```

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

```
package com.wetinknext.ui.theme

import android.content.Context
import androidx.compose.runtime.*
import androidx.compose.runtime.saveable.Saver
import androidx.compose.runtime.saveable.rememberSaveable
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.platform.LocalContext
import kotlin.math.pow

val LocalAppTheme = compositionLocalOf { AppThemes.Gray }

data class CustomThemeColors(
    val appBackground: Int = 0xFF101826.toInt(),
    val grid: Int = 0xFF38424C.toInt(),
    val panel: Int = 0xFF172033.toInt(),
    val accent: Int = 0xFF58A6FF.toInt(),
    val text: Int = 0xFFF4F7FB.toInt(),
)

class ThemeController(
    initialThemeId: String = "gray",
    initialFontMode: String = "Classic",
    initialCustomColors: CustomThemeColors = CustomThemeColors()
) {
    var current by mutableStateOf(resolveTheme(initialThemeId, initialCustomColors))
        private set
    var font by mutableStateOf(if (initialFontMode == "Italic") UiFont.Italic else UiFont.Classic)
        private set
    var customColors by mutableStateOf(initialCustomColors)
        private set
    val customTheme: AppTheme
        get() = createCustomTheme(customColors)

    fun select(theme: AppTheme) {
        current = if (theme.id == "custom") createCustomTheme(customColors) else theme
    }

    fun selectCustom() {
        current = createCustomTheme(customColors)
    }

    fun selectFont(f: UiFont) { font = f }

    fun updateCustomColors(newColors: CustomThemeColors) {
        customColors = newColors
        if (current.id == "custom") current = createCustomTheme(newColors)
    }

    private fun resolveTheme(id: String, colors: CustomThemeColors): AppTheme {
        if (id == "custom") return createCustomTheme(colors)
        return AppThemes.all.find { it.id == id } ?: AppThemes.Gray
    }

    private fun createCustomTheme(colors: CustomThemeColors): AppTheme {
        val appBg = Color(colors.appBackground)
        val panel = Color(colors.panel)
        val accent = Color(colors.accent)
        val text = Color(colors.text)
        val grid = Color(colors.grid)
        val isDark = luminanceOf(appBg) < 0.5f

        return AppTheme(
            id = "custom", displayName = "■■■■", isLight = !isDark,
            appBg = appBg, gridLine = grid,
            panelBg = panel, panelBgSolid = panel.mix(if (isDark) Color.White else Color.Black, 0.08f),
            panelStroke = panel.mix(accent, 0.3f),
            panelInset = panel.mix(if (isDark) Color.Black else Color.White, 0.2f),
            panelInsetSoft = panel.mix(accent, 0.1f),
            canvasBackdrop = appBg, canvasGrid = grid,
            textPrimary = text, textSecondary = text.copy(alpha = 0.6f),
            iconInactive = text.copy(alpha = 0.4f),
            accent = accent, accentSoft = accent.copy(alpha = 0.7f),
            accentMuted = accent.copy(alpha = 0.3f),
            danger = Color(0xFFFF4A5F)
        )
    }

    private fun Color.mix(target: Color, amount: Float): Color {
        return Color(
            red = red * (1f - amount) + target.red * amount,
            green = green * (1f - amount) + target.green * amount,
            blue = blue * (1f - amount) + target.blue * amount,
            alpha = alpha
```

```
        )
    }

    private fun luminanceOf(color: Color): Float {
        fun channel(value: Float): Float =
            if (value <= 0.03928f) value / 12.92f else ((value + 0.055f) / 1.055f).pow(2.4f)
        return 0.2126f * channel(color.red) + 0.7152f * channel(color.green) + 0.0722f * channel(color.blue)
    }

    companion object {
        val Saver: Saver<ThemeController, *> = Saver(
            save = { listOf(it.current.id, it.font.name, it.customColors.appBackground, it.customColors.grid, it.customC
    olors.panel, it.customColors.accent, it.customColors.text) },
            restore = { list ->
                val l = list as List<*>
                ThemeController(
                    initialThemeId = l[0] as String,
                    initialFontMode = l[1] as String,
                    initialCustomColors = CustomThemeColors(
                        l[2] as Int, l[3] as Int, l[4] as Int, l[5] as Int, l[6] as Int
                    )
                )
            }
        )
    }
}

@Composable
fun rememberThemeController(): ThemeController {
    val context = LocalContext.current
    val prefs = remember { context.getSharedPreferences("wetinknext_theme_prefs", Context.MODE_PRIVATE) }

    val controller = rememberSaveable(saver = ThemeController.Saver) {
        val themeId = prefs.getString("theme_id", "gray") ?: "gray"
        val fontMode = prefs.getString("font_mode", "Classic") ?: "Classic"
        val customColors = CustomThemeColors(
            appBackground = prefs.getInt("custom_bg", 0xFF101826.toInt()),
            grid = prefs.getInt("custom_grid", 0xFF38424C.toInt()),
            panel = prefs.getInt("custom_panel", 0xFF172033.toInt()),
            accent = prefs.getInt("custom_accent", 0xFF58A6FF.toInt()),
            text = prefs.getInt("custom_text", 0xFFF4F7FB.toInt())
        )
        ThemeController(themeId, fontMode, customColors)
    }

    LaunchedEffect(controller.current.id, controller.font, controller.customColors) {
        prefs.edit()
            .putString("theme_id", controller.current.id)
            .putString("font_mode", controller.font.name)
            .putInt("custom_bg", controller.customColors.appBackground)
            .putInt("custom_grid", controller.customColors.grid)
            .putInt("custom_panel", controller.customColors.panel)
            .putInt("custom_accent", controller.customColors.accent)
            .putInt("custom_text", controller.customColors.text)
            .apply()
    }

    return controller
}
```

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/BrushDynamicsTest.kt

```
package com.wetinknext.engine.brush

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

class BrushDynamicsTest {
    @Test
    fun resolve_separatesLocalCoverageFromGlobalOpacity() {
        val settings = BrushSettings(
            baseRadiusPx = 10f,
            opacity = 0.2f,
            pressureGamma = 2f,
            minSizeRatio = 0.1f,
            tiltToSize = 0.5f,
            tiltToOpacity = 1f,
        )
        val result = ResolvedDab()

        BrushDynamics.resolve(
            settings = settings,
            pressure = 0.5f,
            tiltX = 1f,
            tiltY = 0f,
            out = result,
        )

        // p = 0.5 ^ 2 = 0.25; pressure size = .325; tilt size = 1.5.
        assertEquals(4.875f, result.radius, 0.0001f)
        // Tilt attenuates, rather than amplifies, local coverage.
        assertEquals(0.1375f, result.coverage, 0.0001f)
        assertEquals(result.coverage, result.opacity, 0.0001f)
    }
}
```

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

```
package com.wetinknext.engine.brush

import kotlinx.serialization.encodeToString
import org.junit.Assert.assertEquals
import org.junit.Test

class BrushSerializationTest {

    @Test
    fun settingsRoundTripKeepsRenderFields() {
        val json = BrushRepository.defaultJson
        val original = BrushSettings(
            name = "Pencil 6B",
            grainAssetPath = "asset:brush/pencil_6b.png",
            grainScale = 5.5f,
            textureDepth = .65f,
            tiltToSize = .55f,
            blendPolicy = BlendPolicy.NORMAL_BUILDUP,
        )

        val encoded = json.encodeToString(original)
        val restored = json.decodeFromString<BrushSettings>(encoded)

        assertEquals(original, restored)
    }

    @Test
    fun presetRoundTrip() {
        val repo = BrushRepository()
        val original = BrushPreset(
            id = "test-preset",
            settings = BrushSettings(name = "Test Brush")
        )

        val encoded = repo.encode(original)
        val restored = repo.decode(encoded)

        assertEquals(original, restored)
    }
}
```

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

```
package com.wetinknext.engine.brush

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

class BrushSettingsTest {
    @Test
    fun resolved_clampsBrushValues() {
        val result = BrushSettings(
            baseRadiusPx = -10f,
            opacity = 3f,
            flow = -1f,
            spacing = 100f,
            pressureGamma = 0f,
            minSizeRatio = 4f,
        ).resolved()

        assertEquals(0.25f, result.baseRadiusPx, 0.001f)
        assertEquals(1f, result.opacity, 0.001f)
        assertEquals(0f, result.flow, 0.001f)
        assertEquals(4f, result.spacing, 0.001f)
        assertEquals(0.05f, result.pressureGamma, 0.001f)
        assertEquals(1f, result.minSizeRatio, 0.001f)
    }
}
```

app/src/test/java/com/wetinknext/engine/brush/CapsuleEmitterTest.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.assertTrue
import org.junit.Test

class CapsuleEmitterTest {

    @Test
    fun finishFlushesTheLastInterpolatedSection() {
        val settings = BrushSettings(
            renderMode = BrushRenderMode.RIBBON,
            baseRadiusPx = 10f,
            smoothing = 0.5f,
            streamline = 0f,
        )
        val emitter = CapsuleEmitter(settings)
        // We need a dummy renderer to receive segments
        // Since we can't easily mock GL context here, we just use a small maxSegments
        val renderer = CapsuleStrokeRenderer(maxSegments = 128)

        emitter.begin(batch(InputAction.DOWN, 0f, 0f, 0L), renderer)
        emitter.append(batch(InputAction.MOVE, 100f, 0f, 16_000_000L), renderer)
        emitter.append(batch(InputAction.MOVE, 200f, 30f, 32_000_000L), renderer)

        val segmentsBeforeFinish = renderer.segmentCount
        emitter.finish(renderer, cancel = false)
        val segmentsAfterFinish = renderer.segmentCount

        assertTrue("Should have some segments after append", segmentsBeforeFinish > 0)
        assertTrue("Finish should have flushed more segments", segmentsAfterFinish > segmentsBeforeFinish)
        assertTrue(emitter.hasStroke)
    }

    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/CapsuleStrokeRendererTest.kt

```
package com.wetinknext.engine.brush

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

class CapsuleStrokeRendererTest {

    @Test
    fun initiallyEmpty() {
        val renderer = CapsuleStrokeRenderer(maxSegments = 100)
```

```
        assertTrue(renderer.isEmpty)
        assertEquals(0, renderer.segmentCount)
    }

    @Test
    fun addsSegmentsWithinCapacity() {
        val renderer = CapsuleStrokeRenderer(maxSegments = 10)
        renderer.beginStroke()

        for (i in 0 until 10) {
            val added = renderer.addSegment(0f, 0f, 5f, 10f, 10f, 5f)
            assertTrue("Segment $i should be added", added)
        }

        assertEquals(10, renderer.segmentCount)
        assertFalse(renderer.isEmpty)

        val addedOverflow = renderer.addSegment(0f, 0f, 5f, 10f, 10f, 5f)
        assertFalse("Overflow segment should not be added", addedOverflow)
        assertEquals(1, renderer.overflowCount)
    }

    @Test
    fun clearResetsStateButNotOverflow() {
        val renderer = CapsuleStrokeRenderer(maxSegments = 10)
        renderer.beginStroke()
        renderer.addSegment(0f, 0f, 5f, 10f, 10f, 5f)

        // Overflow it
        for (i in 0 until 10) renderer.addSegment(0f, 0f, 1f, 1f, 1f, 1f)
        assertEquals(1, renderer.overflowCount)

        renderer.clearStrokeData()
        assertTrue(renderer.isEmpty)
        assertEquals(0, renderer.segmentCount)
        assertEquals(1, renderer.overflowCount)
    }
}
```

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.srgb8ToLi
    near(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 com.wetinknext.engine.core.DirtyRect
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,1f));assertTrue(b.add(1f,1f,1f,0f,1
    f,1f));assertFalse(b.add(2f,2f,1f,0f,1f,1f));assertEquals(1L,b.overflowCount)}
    @Test fun clearResets(){val b=DabBuffer(1);b.add(0f,0f,1f,0f,1f,1f);b.clear();assertEquals(0,b.count)}
    @Test fun dirtyRectIncludesTextureMargin(){
        val buffer = DabBuffer(1)
        buffer.add(10f, 20f, 5f, 0f, 1f, 1f)
        val rect = DirtyRect()
        buffer.includeDirtyRect(rect, extraMargin = 4f)
        assertEquals(1f, rect.left, 0.001f)
        assertEquals(19f, rect.right, 0.001f)
        assertEquals(11f, rect.top, 0.001f)
        assertEquals(29f, rect.bottom, 0.001f)
    }
}
```

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);assertEqu
    als(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_0
    00,10f,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/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);ass
    ertEquals(200f,s.y,.0001f)}
    @Test fun velocityIsPositiveAfterMovement(){val s=Stabilizer();s.process(0,0f,0f);s.process(1_000_000_000,100f,0f);a
    ssertTrue(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 movingStrokeHasNoExtraDownDab(){val e=StampEmitter(BrushSettings(baseRadiusPx=10f,spacing=1f,spacingUsesDi
    ameter=false,pressureToSize=false,pressureToOpacity=false));val d=DabBuffer(16);val down=InputBatch(1).apply{begin(Input
    Action.DOWN);addSample(0f,0f,1f,0f,0f,0f,0,0,PointerTool.STYLUS,false)};val move=InputBatch(1).apply{begin(InputAction.M
    OVE);addSample(20f,0f,1f,0f,0f,0f,1,0,PointerTool.STYLUS,false)};e.begin(down,d);e.append(move,d);e.finish(d,false);asse
    rtEquals(2,d.count)}

    @Test fun tapProducesExactlyOneDab(){val e=StampEmitter(BrushSettings());val d=DabBuffer(16);val down=InputBatch(1).
    apply{begin(InputAction.DOWN);addSample(5f,5f,1f,0f,0f,0f,0,0,PointerTool.STYLUS,false)};e.begin(down,d);e.finish(d,fals
    e);assertEquals(1,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);as
    sertEquals(.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.assertNotNull
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.peekUndo())
        assertTrue(manager.commitUndo(entry))
        assertFalse(manager.canUndo)
        assertTrue(manager.canRedo)
        assertSame(entry, manager.peekRedo())
        assertTrue(manager.commitRedo(entry))
        assertTrue(manager.canUndo)
        assertFalse(manager.canRedo)
    }

    @Test
    fun aNewEntryClearsRedoHistory() {
        val manager = UndoManager()
        val first = entry(1L)
        manager.push(first)
        val undo = manager.peekUndo()
        assertNotNull(undo)
        assertTrue(manager.commitUndo(checkNotNull(undo)))
        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)
    }

    @Test
    fun rejectedCommitLeavesUndoHistoryUntouched() {
        val manager = UndoManager()
        val stored = entry(1L)
        manager.push(stored)

        assertFalse(manager.commitUndo(entry(2L)))
        assertSame(stored, manager.peekUndo())
        assertTrue(manager.canUndo)
        assertFalse(manager.canRedo)
    }

    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),
    )
}
```