```
➜  SGFPlayerClean git:(main) ✗ find . -name "*.swift" -not -path
'*/.*' -print0 | while IFS= read -r -d '' file; do echo ""; echo "//
FILE: $file"; cat "$file"; done

// FILE: ./ViewModels/LayoutViewModel.swift
//
//  LayoutViewModel.swift
//  SGFPlayerClean
//
//  Created: 2025-11-19
//  Purpose: Manages geometric calculations for the 2D board layout
//

import SwiftUI
import Combine

class LayoutViewModel: ObservableObject {

    // MARK: - Published State

    /// The exact screen frame of the board image
    @Published var boardFrame: CGRect = .zero

    /// The center point of the board
    @Published var boardCenter: CGPoint = .zero

    /// The total window size
    @Published var windowSize: CGSize = .zero

    // MARK: - Internal Metrics

    private var cellWidth: CGFloat = 0
    private var cellHeight: CGFloat = 0

    // MARK: - Initialization

    init() {}

    // MARK: - Layout Logic

    /// Updates the layout state based on the calculated frame from
the View
    /// Called by ContentView2D when it determines where the board
sits
    func updateBoardFrame(_ frame: CGRect, boardSize: Int) {
        self.boardFrame = frame
        self.boardCenter = CGPoint(x: frame.midX, y: frame.midY)

        // Recalculate cell metrics based on the visual frame
        // Logic: Board width / (lines + 1) to account for margins
        let divisor = CGFloat(boardSize + 1)

        if divisor > 0 {
            self.cellWidth = frame.width / divisor
            self.cellHeight = frame.height / divisor
        }

        // Notify observers that layout has changed
        self.objectWillChange.send()
    }

    /// Initial calculation logic (legacy/fallback)
    /// Used if the view hasn't explicitly set the frame yet
    func calculateLayout(containerSize: CGSize, boardSize: Int,
leftPanelWidth: CGFloat) {
        self.windowSize = containerSize

        // Default aspect ratio for Go board (slightly non-square)
        let boardAspect: CGFloat = 1.0 / 1.0773

        let availableW = leftPanelWidth
        let availableH = containerSize.height

        var w: CGFloat = 0
        var h: CGFloat = 0

        if availableW / availableH < boardAspect {
            w = availableW * 0.9 // 90% fill
            h = w / boardAspect
        } else {
            h = availableH * 0.9 // 90% fill
            w = h * boardAspect
        }

        // Center in left panel
        let x = (leftPanelWidth - w) / 2
        let y = (availableH - h) / 2

        let frame = CGRect(x: x, y: y, width: w, height: h)
        updateBoardFrame(frame, boardSize: boardSize)
    }

    func handleResize(newSize: CGSize, boardSize: Int, leftPanelWidth:
CGFloat) {
        calculateLayout(containerSize: newSize, boardSize: boardSize,
leftPanelWidth: leftPanelWidth)
    }

    // MARK: - Getters for Subviews
    func getCellWidth(boardSize: Int) -> CGFloat {
        if cellWidth > 0 { return cellWidth }
        // Fallback if frame isn't set yet
        return (boardFrame.width > 0 ? boardFrame.width : 500) /
CGFloat(boardSize + 1)
    }

    func getCellHeight(boardSize: Int) -> CGFloat {
        if cellHeight > 0 { return cellHeight }
        return (boardFrame.height > 0 ? boardFrame.height : 500) /
CGFloat(boardSize + 1)
    }

    func getLidDiameter(boardSize: Int) -> CGFloat {
        // Requested: Diameter approx 1/3 of the board height
        if boardFrame.height > 0 {
            return boardFrame.height / 3.0
        }
        return 150.0 // Fallback
    }

    func getWhiteStoneSize(boardSize: Int) -> CGFloat {
        return getCellWidth(boardSize: boardSize) * 0.95
    }

    func getBlackStoneSize(boardSize: Int) -> CGFloat {
        // Black stones are traditionally slightly larger (optical
illusion correction)
        return getCellWidth(boardSize: boardSize) * 0.97
    }
}

// FILE: ./ViewModels/GameCacheManager.swift
import Foundation

class GameCacheManager: ObservableObject {
    func loadGame(_ game: SGFGame, fingerprint: String) {
        // Placeholder for caching logic
    }

    func preCalculateGame(_ game: SGFGame, fingerprint: String) {
        // Placeholder for background calculation
    }
}

// FILE: ./ViewModels/OGSGameViewModel.swift
// MARK: - File: OGSGameViewModel.swift (v4.202)
import Foundation
import Combine
```

```
class OGSGameViewModel: ObservableObject {
    @Published var gameInfo: GameInfo?
    @Published var isMyTurn: Bool = false
    @Published var gameStatus: String = "Connecting..."
    @Published var gamePhase: String = "none"

    private var ogsClient: OGSClient
    private var player: SGFPlayer
    private var timeControl: TimeControlManager
    private var cancellables = Set<AnyCancellable>()

    init(ogsClient: OGSClient, player: SGFPlayer, timeControl:
TimeControlManager) {
        self.ogsClient = ogsClient
        self.player = player
        self.timeControl = timeControl
        setupObservers()
    }

    private func setupObservers() {
        ogsClient.$activeGameID.sink { [weak self] id in
            if id == nil {
                self?.gameStatus = "Not in a game"
                self?.gamePhase = "none"
            }
        }.store(in: &cancellables)

        ogsClient.$currentPlayerID.sink { [weak self] current in
            guard let self = self, let myID = self.ogsClient.playerID
else { return }
            self.isMyTurn = (current == myID)
            self.gameStatus = self.isMyTurn ? "Your turn" : "Waiting
for opponent"
        }.store(in: &cancellables)

        NotificationCenter.default.publisher(for:
NSNotification.Name("OGSGameDataReceived"))
            .sink { [weak self] notification in
                if let data = notification.userInfo?["gameData"] as?
[String: Any],
                   let phase = data["phase"] as? String {
                    DispatchQueue.main.async { self?.gamePhase =
phase }
                }
            }.store(in: &cancellables)
    }

    func pass() {
        guard let id = ogsClient.activeGameID else { return }
        ogsClient.sendPass(gameID: id, moveNumber:
```

```
player.serverMoveNumber)
    }

    func resign() {
        guard let id = ogsClient.activeGameID else { return }
        ogsClient.resignGame(gameID: id)
    }

    func startQuickMatch() {
        ogsClient.startAutomatch()
    }
}

// FILE: ./ViewModels/BoardViewModel.swift
// MARK: - File: BoardViewModel.swift (v8.146)
import Foundation
import SwiftUI
import Combine

class BoardViewModel: ObservableObject {
    @Published var isOnlineContext: Bool = false
    @Published var lastMovePosition: BoardPosition?
    @Published var currentMoveIndex: Int = 0
    @Published var stonesToRender: [RenderStone] = []
    @Published var ghostPosition: BoardPosition?
    @Published var ghostColor: Stone?
    @Published var blackCapturedCount: Int = 0
    @Published var whiteCapturedCount: Int = 0
    @Published var isAutoPlaying: Bool = false
    @Published var totalMoves: Int = 0

    let onRequestUpdate3D = PassthroughSubject<Void, Never>()
    var engine: SGFPlayer; var ogsClient: OGSClient
    private var jitterEngine: StoneJitter
    private var isSyncing = false
    private var cancellables = Set<AnyCancellable>()

    init(player: SGFPlayer, ogsClient: OGSClient) {
        self.engine = player; self.ogsClient = ogsClient
        self.jitterEngine = StoneJitter(boardSize: player.board.size)
        player.moveProcessed.receive(on: RunLoop.main).sink { [weak
self] in self?.syncState() }.store(in: &cancellables)
    }

    var boardSize: Int { self.engine.board.size }

    func loadGame(_ wrapper: SGFGameWrapper) { self.isOnlineContext =
false; self.engine.load(game: wrapper.game) }
    func goToStart() { engine.seek(to: 0) }
    func goToEnd() { engine.seek(to: engine.maxIndex) }
    func stepForward() { engine.stepForward() }
    func stepBackward() { engine.stepBackward() }
    func stepForwardTen() { for _ in 0..<10 { engine.stepForward() } }
    func stepBackwardTen() { for _ in 0..<10
{ engine.stepBackward() } }
    func toggleAutoPlay() { if engine.isPlaying { engine.pause() }
else { engine.play() } }
    func seekToMove(_ index: Int) { engine.seek(to: index) }
    func stopAutoPlay() { engine.pause() }
    func undoLastOnlineMove() { engine.stepBackward() }
    func resetToEmpty() { engine.clear(); self.isOnlineContext =
false; self.syncState() }

    func handleRemoteMove(x: Int, y: Int, playerId: Int?) {
        let color: Stone = (playerId ==
ogsClient.blackPlayerID) ? .black : .white
        self.engine.playMoveOptimistically(color: color, x: x, y: y)
    }

    func initializeOnlineGame(width: Int, height: Int, initialStones:
[BoardPosition: Stone], nextTurn: Stone, moveNumber: Int) {
        self.isOnlineContext = true
        self.engine.loadOnline(size: width, setup: initialStones,
nextPlayer: nextTurn, startMoveNumber: moveNumber)
        self.syncState()
        self.objectWillChange.send()
    }

    func placeStone(at pos: BoardPosition) {
        if isOnlineContext {
            let nextMoveNum = self.engine.serverMoveNumber + 1
            self.ogsClient.sendMove(gameID:
self.ogsClient.activeGameID ?? 0, x: pos.col, y: pos.row, moveNumber:
nextMoveNum)
        }
    }

    func syncState() {
        if isSyncing { return }; isSyncing = true
        let idx = self.engine.currentIndex; let last =
self.engine.lastMove.map { BoardPosition($0.y, $0.x) }
        let snap = self.engine.board
        Task.detached(priority: .userInitiated) { [weak self] in
            guard let self = self else { return }
            self.jitterEngine.prepare(forMove: idx, stones:
snap.stones)
            var list: [RenderStone] = []
            for (pos, col) in snap.stones {
                let off = self.jitterEngine.offset(forX: pos.col, y:
pos.row, moveIndex: idx, stones: snap.stones)
```

```
                list.append(RenderStone(id: pos, color: col, offset:
off))
            }
            let safeList = list
            await MainActor.run {
                self.stonesToRender = safeList; self.lastMovePosition
= last; self.currentMoveIndex = idx
                self.blackCapturedCount =
self.engine.whiteStonesCaptured; self.whiteCapturedCount =
self.engine.blackStonesCaptured
                self.isAutoPlaying = self.engine.isPlaying;
self.totalMoves = self.engine.maxIndex
                self.isSyncing = false; self.onRequestUpdate3D.send();
self.objectWillChange.send()
            }
        }
    }

    func updateGhostStone(at pos: BoardPosition?) { self.ghostPosition
= pos; self.ghostColor = self.engine.turn }
    func clearGhostStone() { self.ghostPosition = nil }
    func getJitterOffset(forPosition pos: BoardPosition) -> CGPoint
{ return self.jitterEngine.offset(forX: pos.col, y: pos.row,
moveIndex: self.currentMoveIndex, stones: self.engine.board.stones) }
}

// FILE: ./ViewModels/TimeControlManager.swift
import Foundation

class TimeControlManager: ObservableObject {
    @Published var blackTime: String = "00:00"
    @Published var whiteTime: String = "00:00"

    init() {}
}

// FILE: ./Models/AppSettings.swift
// MARK: - File: AppSettings.swift (v1.100)
import Foundation
import Combine

class AppSettings: ObservableObject {
    static let shared = AppSettings()

    // MARK: - Playback
    @Published var moveInterval: TimeInterval {
        didSet { UserDefaults.standard.set(moveInterval, forKey:
"moveInterval") }
    }
    @Published var jitterMultiplier: Double {
        didSet { UserDefaults.standard.set(jitterMultiplier, forKey:
"jitterMultiplier") }
    }
    @Published var shuffleGameOrder: Bool {
        didSet { UserDefaults.standard.set(shuffleGameOrder, forKey:
"shuffleGameOrder") }
    }
    @Published var startGameOnLaunch: Bool {
        didSet { UserDefaults.standard.set(startGameOnLaunch, forKey:
"startGameOnLaunch") }
    }

    // MARK: - Visuals
    @Published var showMoveNumbers: Bool {
        didSet { UserDefaults.standard.set(showMoveNumbers, forKey:
"showMoveNumbers") }
    }
    @Published var showLastMoveDot: Bool {
        didSet { UserDefaults.standard.set(showLastMoveDot, forKey:
"showLastMoveDot") }
    }
    @Published var showLastMoveCircle: Bool {
        didSet { UserDefaults.standard.set(showLastMoveCircle, forKey:
"showLastMoveCircle") }
    }
    @Published var showBoardGlow: Bool {
        didSet { UserDefaults.standard.set(showBoardGlow, forKey:
"showBoardGlow") }
    }
    @Published var showEnhancedGlow: Bool {
        didSet { UserDefaults.standard.set(showEnhancedGlow, forKey:
"showEnhancedGlow") }
    }
    @Published var showDropInAnimation: Bool {
        didSet { UserDefaults.standard.set(showDropInAnimation,
forKey: "showDropInAnimation") }
    }

    // MARK: - 3D Viewport Default Persistence
    @Published var camera3DRotationX: Double {
        didSet { UserDefaults.standard.set(camera3DRotationX, forKey:
"cam3D.rotX") }
    }
    @Published var camera3DRotationY: Double {
        didSet { UserDefaults.standard.set(camera3DRotationY, forKey:
"cam3D.rotY") }
    }
    @Published var camera3DDistance: Double {
        didSet { UserDefaults.standard.set(camera3DDistance, forKey:
"cam3D.dist") }
```

```
    }
    @Published var camera3DPanX: Double {
        didSet { UserDefaults.standard.set(camera3DPanX, forKey:
"cam3D.panX") }
    }
    @Published var camera3DPanY: Double {
        didSet { UserDefaults.standard.set(camera3DPanY, forKey:
"cam3D.panY") }
    }

    // MARK: - UI Appearance
    @Published var panelOpacity: Double {
        didSet { UserDefaults.standard.set(panelOpacity, forKey:
"panelOpacity") }
    }
    @Published var panelDiffusiveness: Double {
        didSet { UserDefaults.standard.set(panelDiffusiveness, forKey:
"panelDiffusiveness") }
    }

    @Published var folderURL: URL? {
        didSet {
            if let url = folderURL { try?
UserDefaults.standard.set(url.bookmarkData(), forKey:
"folderURLBookmark") }
        }
    }

    private init() {
        self.moveInterval = UserDefaults.standard.double(forKey:
"moveInterval") == 0 ? 0.5 : UserDefaults.standard.double(forKey:
"moveInterval")
        self.jitterMultiplier = UserDefaults.standard.object(forKey:
"jitterMultiplier") as? Double ?? 1.0
        self.shuffleGameOrder = UserDefaults.standard.bool(forKey:
"shuffleGameOrder")
        self.startGameOnLaunch = UserDefaults.standard.bool(forKey:
"startGameOnLaunch")
        self.showMoveNumbers = UserDefaults.standard.bool(forKey:
"showMoveNumbers")
        self.showLastMoveDot = UserDefaults.standard.object(forKey:
"showLastMoveDot") as? Bool ?? true
        self.showLastMoveCircle = UserDefaults.standard.bool(forKey:
"showLastMoveCircle")
        self.showBoardGlow = UserDefaults.standard.bool(forKey:
"showBoardGlow")
        self.showEnhancedGlow = UserDefaults.standard.bool(forKey:
"showEnhancedGlow")
        self.showDropInAnimation =
UserDefaults.standard.object(forKey: "showDropInAnimation") as?
```

```
Bool ?? true
        self.panelOpacity = UserDefaults.standard.object(forKey:
"panelOpacity") as? Double ?? 0.3
        self.panelDiffusiveness = UserDefaults.standard.object(forKey:
"panelDiffusiveness") as? Double ?? 0.8

        // Load 3D Defaults (or Natural Perspective if empty)
        self.camera3DRotationX = UserDefaults.standard.object(forKey:
"cam3D.rotX") as? Double ?? 0.75
        self.camera3DRotationY = UserDefaults.standard.object(forKey:
"cam3D.rotY") as? Double ?? 0.0
        self.camera3DDistance = UserDefaults.standard.object(forKey:
"cam3D.dist") as? Double ?? 25.0
        self.camera3DPanX = UserDefaults.standard.object(forKey:
"cam3D.panX") as? Double ?? 0.0
        self.camera3DPanY = UserDefaults.standard.object(forKey:
"cam3D.panY") as? Double ?? 0.0

        if let data = UserDefaults.standard.data(forKey:
"folderURLBookmark") {
            var isStale = false
            self.folderURL = try? URL(resolvingBookmarkData: data,
bookmarkDataIsStale: &isStale)
        }
    }
}

// FILE: ./Models/AppModel.swift
// MARK: - File: AppModel.swift (v4.247)
import Foundation
import SwiftUI
import Combine
import AVFoundation

final class AppModel: ObservableObject {
    @AppStorage("verboseLogging") var verboseLogging: Bool = false
    @AppStorage("viewMode") var viewMode: ViewMode = .view2D

    @Published var folderURL: URL? { didSet { persistFolderURL() } }
    @Published var games: [SGFGameWrapper] = []
    @Published var selection: SGFGameWrapper? = nil { didSet
{ persistLastGame() } }
    @Published var activePlaylist: [SGFGameWrapper] = []
    @Published var player = SGFPlayer()
    @Published var isLoadingGames: Bool = false
    @Published var ogsClient = OGSClient()
    @Published var layoutVM = LayoutViewModel()
    @Published var timeControl = TimeControlManager()
    @Published var ogsGame: OGSGameViewModel?
    @Published var gameCacheManager = GameCacheManager()
    @Published var isOnlineMode: Bool = false
    @Published var showPreGameOverlay: Bool = false
    @Published var browserTab: OGSBrowserTab = .challenge
    @Published var isCreatingChallenge: Bool = false
    @Published var showDebugDashboard: Bool = false

    var boardVM: BoardViewModel?
    private var stoneClickPlayer: AVAudioPlayer?
    private var cancellables: Set<AnyCancellable> = []

    init() {
        self.boardVM = BoardViewModel(player: player, ogsClient:
ogsClient)
        self.ogsGame = OGSGameViewModel(ogsClient: ogsClient, player:
player, timeControl: timeControl)
        restoreFolderURL(); if let url = folderURL { loadFolder(url) }
        setupAudio(); self.ogsClient.connect(); setupOGSObservers()
    }

    func joinOnlineGame(id: Int) {
        player.clear()
        self.ogsClient.acceptChallenge(challengeID: id) { [weak self]
newGameID, _ in
            if let gameID = newGameID { self?.finalizeJoin(gameID:
gameID) }
        }
    }

    private func finalizeJoin(gameID: Int) {
        self.ogsClient.fetchGameState(gameID: gameID) { [weak self]
(rootJson: [String: Any]?) in
            guard let self = self, let root = rootJson else { return }
            var payload: [String: Any] = [:]
            if let inner = root["gamedata"] as? [String: Any]
{ payload = inner } else { payload = root }
            DispatchQueue.main.async {
                if let auth = root["auth"] as? String
{ self.ogsClient.activeGameAuth = auth }
                NotificationCenter.default.post(name:
NSNotification.Name("OGSGameDataReceived"), object: nil, userInfo:
["gameData": payload])
                self.ogsClient.connectToGame(gameID: gameID)
            }
        }
    }

    private func setupOGSObservers() {
        let c = NotificationCenter.default
        c.publisher(for:
```

```
NSNotification.Name("OGSGameDataReceived")).receive(on:
RunLoop.main).sink { [weak self] n in
self?.handleOGSGameLoad(n) }.store(in: &cancellables)
        c.publisher(for:
NSNotification.Name("OGSMoveReceived")).receive(on: RunLoop.main).sink
{ [weak self] n in self?.handleOGSMoveUpdate(n) }.store(in:
&cancellables)
    }

    private func handleOGSGameLoad(_ notification: Notification) {
        guard let userInfo = notification.userInfo, let gameData =
userInfo["gameData"] as? [String: Any] else { return }
        player.clear()
        var initialStones: [BoardPosition: Stone] = [:]
        if let state = gameData["initial_state"] as? [String: Any],
let bStr = state["black"] as? String {
            let chars = Array(bStr)
            for i in stride(from: 0, to: chars.count, by: 2) {
                if i + 1 < chars.count {
                    let coord = String(chars[i...i+1])
                    if let (x, y) = SGFCoordinates.parse(coord)
{ initialStones[BoardPosition(y, x)] = .black }
                }
            }
        }
        let nextTurn: Stone = (gameData["initial_player"] as? String
== "white") ? .white : .black
        boardVM?.initializeOnlineGame(width: gameData["width"] as?
Int ?? 19, height: gameData["height"] as? Int ?? 19, initialStones:
initialStones, nextTurn: nextTurn, moveNumber: gameData["move_number"]
as? Int ?? 0)
        if let moves = gameData["moves"] as? [[Any]] {
            for m in moves { if m.count >= 2, let x = m[0] as? Int,
let y = m[1] as? Int { boardVM?.handleRemoteMove(x: x, y: y, playerId:
nil) } }
        }
    }

    private func handleOGSMoveUpdate(_ notification: Notification) {
        guard let userInfo = notification.userInfo else { return }
        if let m = userInfo["move"] as? [Any], m.count >= 2, let x =
m[0] as? Int, let y = m[1] as? Int, let mn = userInfo["move_number"]
as? Int {
            let pid = userInfo["player_id"] as? Int
            let color: Stone = (pid ==
ogsClient.blackPlayerID) ? .black : .white
            playStoneClickSound(); player.applyOnlineMove(color:
color, x: x, y: y, moveNumber: mn)
        }
    }
    private func setupAudio() { if let url =
Bundle.main.url(forResource: "Stone_click_1", withExtension: "mp3")
{ stoneClickPlayer = try? AVAudioPlayer(contentsOf: url);
stoneClickPlayer?.prepareToPlay() } }
    func playStoneClickSound() { stoneClickPlayer?.play() }
    func selectGame(_ g: SGFGameWrapper) { selection = g;
boardVM?.loadGame(g) }
    func promptForFolder() { let p = NSOpenPanel(); p.canChooseFiles =
false; p.canChooseDirectories = true; if p.runModal() == .OK, let u =
p.url { folderURL = u; loadFolder(u) } }
    func loadFolder(_ u: URL) {
        isLoadingGames = true
        DispatchQueue.global(qos: .userInitiated).async {
            let fm = FileManager.default; var urls: [URL] = []
            if let en = fm.enumerator(at: u,
includingPropertiesForKeys: [.isRegularFileKey], options:
[.skipsHiddenFiles]) { for case let f as URL in en { if
f.pathExtension.lowercased() == "sgf" { urls.append(f) } } }
            urls.sort { $0.path.localizedStandardCompare($1.path)
== .orderedAscending }
            var p: [SGFGameWrapper] = []
            for f in urls { if let d = try? Data(contentsOf: f), let t
= String(data: d, encoding: .utf8), let tree = try?
SGFParser.parse(text: t) { p.append(SGFGameWrapper(url: f, game:
SGFGame.from(tree: tree))) } }
            DispatchQueue.main.async { self.games = p;
self.activePlaylist = p; self.isLoadingGames = false; if let f =
p.first { self.selectGame(f) } }
        }
    }
    private func persistFolderURL() { if let u = folderURL, let b =
try? u.bookmarkData(options: .withSecurityScope,
includingResourceValuesForKeys: nil, relativeTo: nil)
{ UserDefaults.standard.set(b, forKey: "sgfplayer.folderURL") } }
    private func restoreFolderURL() { if let b =
UserDefaults.standard.data(forKey: "sgfplayer.folderURL") { var
isStale = false; if let u = try? URL(resolvingBookmarkData: b,
options: [.withSecurityScope], relativeTo: nil, bookmarkDataIsStale:
&isStale) { if u.startAccessingSecurityScopedResource() { folderURL =
u } } } }
    private func persistLastGame() { if let s = selection
{ UserDefaults.standard.set(s.url.lastPathComponent, forKey:
"sgfplayer.lastGame") } }
}

// FILE: ./Models/StoneJitter.swift
// MARK: - File: StoneJitter.swift (v5.101)
import Foundation
import CoreGraphics
```

```
class StoneJitter {
    private struct Preset {
        var sigma: CGFloat = 0.12
        var clamp: CGFloat = 0.25
        var minDistance: CGFloat = 1.05
        var pushStrength: CGFloat = 0.6
    }

    private let boardSize: Int
    private var eccentricity: CGFloat
    private var sigma: CGFloat
    private var clamp: CGFloat
    private var minDistance: CGFloat
    private var pushStrength: CGFloat
    private let cellAspectRatio: CGFloat = 23.7 / 22.0

    private var initialJitter: [[CGPoint?]]
    private var finalOffsets: [[CGPoint?]]
    private var lastPreparedMove: Int = .min

    init(boardSize: Int = 19, eccentricity: CGFloat = 1.0) {
        self.boardSize = boardSize
        self.eccentricity = eccentricity
        self.initialJitter = Array(repeating: Array(repeating: nil,
count: boardSize), count: boardSize)
        self.finalOffsets = Array(repeating: Array(repeating: nil,
count: boardSize), count: boardSize)

        let preset = Preset()
        self.sigma = preset.sigma * eccentricity
        self.clamp = preset.clamp * eccentricity
        self.minDistance = preset.minDistance
        self.pushStrength = preset.pushStrength
    }

    func prepare(forMove moveIndex: Int, stones: [BoardPosition:
Stone]) {
        guard moveIndex != lastPreparedMove else { return }
        lastPreparedMove = moveIndex
        finalOffsets = Array(repeating: Array(repeating: nil, count:
boardSize), count: boardSize)
    }

    func offset(forX x: Int, y: Int, moveIndex: Int, stones:
[BoardPosition: Stone]) -> CGPoint {
        guard eccentricity > 0.001 else { return .zero }
        guard x >= 0, x < boardSize, y >= 0, y < boardSize else
{ return .zero }
```

```
        if let cached = finalOffsets[y][x] { return cached }

        let initial = getInitialJitter(x: x, y: y, moveIndex:
moveIndex)
        let final = resolveCollisions(x: x, y: y, initial: initial,
stones: stones)
        finalOffsets[y][x] = final
        return final
    }

    func setEccentricity(_ value: CGFloat) {
        guard value != eccentricity else { return }
        eccentricity = value
        let preset = Preset()
        sigma = preset.sigma * eccentricity
        clamp = preset.clamp * eccentricity
        clearAll()
    }

    private func getInitialJitter(x: Int, y: Int, moveIndex: Int) ->
CGPoint {
        if let cached = initialJitter[y][x] { return cached }

        let seed = UInt64(x) * 73856093 + UInt64(y) * 19349663 +
UInt64(moveIndex) * 83492791
        var rng = SeededRandomNumberGenerator(seed: seed)

        let u1 = CGFloat.random(in: 0...1, using: &rng)
        let u2 = CGFloat.random(in: 0...1, using: &rng)
        let mag = sqrt(-2.0 * log(max(u1, 1e-10)))
        let ang = 2.0 * .pi * u2

        let dx = max(-clamp, min(clamp, mag * cos(ang) * sigma))
        let dy = max(-clamp, min(clamp, mag * sin(ang) * sigma))

        let point = CGPoint(x: dx, y: dy)
        initialJitter[y][x] = point
        return point
    }

    private func resolveCollisions(x: Int, y: Int, initial: CGPoint,
stones: [BoardPosition: Stone]) -> CGPoint {
        var offset = initial
        let centerPos = CGPoint(x: CGFloat(x) + offset.x, y:
CGFloat(y) + offset.y)
        let neighbors = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]

        for (nx, ny) in neighbors {
            guard nx >= 0, ny >= 0, nx < boardSize, ny < boardSize
else { continue }
```

```
            // Speed Fix: Using the pre-passed dictionary is now O(1)
per neighbor
            guard stones[BoardPosition(ny, nx)] != nil else
{ continue }

            let nOff = initialJitter[ny][nx] ?? getInitialJitter(x:
nx, y: ny, moveIndex: lastPreparedMove)
            let nPos = CGPoint(x: CGFloat(nx) + nOff.x, y: CGFloat(ny)
+ nOff.y)

            let dx = nPos.x - centerPos.x
            let dy = (nPos.y - centerPos.y) / cellAspectRatio
            let dist = hypot(dx, dy)

            if dist < minDistance && dist > 0.001 {
                let overlap = minDistance - dist
                let pushDir = CGPoint(x: dx / dist, y: (dy / dist) *
cellAspectRatio)
                offset.x -= pushDir.x * overlap * 0.3
                offset.y -= pushDir.y * overlap * 0.3
            }
        }
        return offset
    }

    func clearAll() {
        initialJitter = Array(repeating: Array(repeating: nil, count:
boardSize), count: boardSize)
        finalOffsets = Array(repeating: Array(repeating: nil, count:
boardSize), count: boardSize)
        lastPreparedMove = .min
    }
}

private struct SeededRandomNumberGenerator: RandomNumberGenerator {
    private var state: UInt64
    init(seed: UInt64) { self.state = seed }
    mutating func next() -> UInt64 {
        state = state &* 6364136223846793005 &+ 1442695040888963407
        return state
    }
}

// FILE: ./Models/SGFPlayerEngine.swift
// MARK: - File: SGFPlayerEngine.swift (v8.129)
import Foundation
import Combine
import QuartzCore

final class SGFPlayer: ObservableObject {
    @Published var board: BoardSnapshot
    @Published var lastMove: MoveRef? = nil
    @Published var isPlaying: Bool = false
    @Published var currentIndex: Int = 0
    @Published var whiteStonesCaptured: Int = 0
    @Published var blackStonesCaptured: Int = 0
    let moveProcessed = PassthroughSubject<Void, Never>()

    private var _moves: [(Stone, (Int,Int)?)] = []
    private var _baseSetup: [(Stone, Int, Int)] = []
    private var baseSize: Int = 19
    private var initialPlayer: Stone = .black
    var serverMoveNumber: Int = 0
    private var timer: AnyCancellable?

    init() { self.board = BoardSnapshot(size: 19, grid:
Array(repeating: Array(repeating: nil, count: 19), count: 19), stones:
[:]) }

    var turn: Stone { if let last = lastMove { return last.color
== .black ? .white : .black }; return initialPlayer }
    var maxIndex: Int { _moves.count }

    func loadOnline(size: Int, setup: [BoardPosition: Stone],
nextPlayer: Stone, startMoveNumber: Int) {
        self.baseSize = size; self._moves = []; self._baseSetup =
setup.map { ($1, $0.col, $0.row) }; self.initialPlayer = nextPlayer;
self.serverMoveNumber = startMoveNumber; self.reset()
    }

    func applyOnlineMove(color: Stone, x: Int, y: Int, moveNumber:
Int) {
        guard moveNumber > serverMoveNumber || (moveNumber ==
serverMoveNumber && lastMove == nil) else { return }
        self.serverMoveNumber = moveNumber
        self.playMoveOptimistically(color: color, x: x, y: y)
    }

    func clear() { pause(); _moves = []; _baseSetup = []; currentIndex
= 0; whiteStonesCaptured = 0; blackStonesCaptured = 0;
serverMoveNumber = 0; syncSnapshot(grid: Array(repeating:
Array(repeating: nil, count: baseSize), count: baseSize));
moveProcessed.send() }

    func reset() {
        pause(); currentIndex = 0; whiteStonesCaptured = 0;
blackStonesCaptured = 0
        var grid = Array(repeating: Array(repeating: Stone?.none,
count: baseSize), count: baseSize)
        var stones: [BoardPosition: Stone] = [:]
        for (stone, x, y) in _baseSetup where x < baseSize && y <
baseSize {
            grid[y][x] = stone; stones[BoardPosition(y, x)] = stone
        }
        self.lastMove = nil; syncSnapshot(grid: grid);
moveProcessed.send()
    }

    func playMoveOptimistically(color: Stone, x: Int, y: Int) {
        _moves.append((color, (x, y))); apply(moveAt: _moves.count -
1); currentIndex = _moves.count; moveProcessed.send()
    }

    private func apply(moveAt i: Int) {
        let (color, coord) = _moves[i]; var g = board.grid
        guard let (x, y) = coord else { syncSnapshot(grid: g);
return }
        if x >= 0, y >= 0, x < baseSize, y < baseSize {
            g[y][x] = color; let opponent = color.opponent; var
captured = 0; var processed = Set<Point>()
            for (nx, ny) in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]
where nx >= 0 && nx < baseSize && ny >= 0 && ny < baseSize {
                let p = Point(x: nx, y: ny)
                if g[ny][nx] == opponent && !processed.contains(p) {
                    var visited = Set<Point>(); let group =
collectGroup(from: p, color: opponent, grid: g, visited: &visited);
processed.formUnion(visited)
                    if liberties(of: group, in: g).isEmpty { captured
+= group.count; for s in group { g[s.y][s.x] = nil } }
                }
            }
            if color == .black { whiteStonesCaptured += captured }
else { blackStonesCaptured += captured }
            var sV = Set<Point>(); let sG = collectGroup(from:
Point(x: x, y: y), color: color, grid: g, visited: &sV)
            if liberties(of: sG, in: g).isEmpty { if color == .black
{ blackStonesCaptured += sG.count } else { whiteStonesCaptured +=
sG.count }; for s in sG { g[s.y][s.x] = nil } }
            self.lastMove = MoveRef(color: color, x: x, y: y);
syncSnapshot(grid: g)
        }
    }

    // MARK: - Playback Logic (Restored)
    func stepForward() { guard currentIndex < _moves.count else
{ pause(); return }; apply(moveAt: currentIndex); currentIndex += 1;
moveProcessed.send() }
    func stepBackward() { guard currentIndex > 0 else { return };
seek(to: currentIndex - 1) }
    func seek(to idx: Int) { let target = max(0, min(idx,
_moves.count)); if target < currentIndex { reset(); for i in
0..<target { apply(moveAt: i) } } else { for i in
currentIndex..<target { apply(moveAt: i) } }; currentIndex = target;
moveProcessed.send() }
    func play() { if isPlaying { return }; isPlaying = true; timer =
Timer.publish(every: 0.75, on: .main, in: .common).autoconnect().sink
{ [weak self] _ in self?.stepForward() } }
    func pause() { isPlaying = false; timer?.cancel(); timer = nil }

    private func syncSnapshot(grid: [[Stone?]]) {
        var flat: [BoardPosition: Stone] = [:]
        for r in 0..<baseSize { for c in 0..<baseSize { if let s =
grid[r][c] { flat[BoardPosition(r, c)] = s } } }
        self.board = BoardSnapshot(size: baseSize, grid: grid, stones:
flat)
    }

    private struct Point: Hashable { let x, y: Int }
    private func collectGroup(from start: Point, color: Stone, grid:
[[Stone?]], visited: inout Set<Point>) -> [Point] {
        var stack = [start], group: [Point] = [], h = grid.count, w =
grid[0].count
        while let p = stack.popLast() {
            if visited.contains(p) { continue }; visited.insert(p);
guard p.y < h, p.x < w, grid[p.y][p.x] == color else { continue }
            group.append(p); for (nx, ny) in [(p.x-1, p.y), (p.x+1,
p.y), (p.x, p.y-1), (p.x, p.y+1)] where nx >= 0 && nx < w && ny >= 0
&& ny < h { if grid[ny][nx] == color { stack.append(Point(x: nx, y:
ny)) } }
        }
        return group
    }
    private func liberties(of group: [Point], in grid: [[Stone?]]) ->
[Point] {
        var libs = Set<Point>(), h = grid.count, w = grid[0].count
        for p in group { for (nx, ny) in [(p.x-1, p.y), (p.x+1, p.y),
(p.x, p.y-1), (p.x, p.y+1)] where nx >= 0 && nx < w && ny >= 0 && ny <
h { if grid[ny][nx] == nil { libs.insert(Point(x: nx, y: ny)) } } }
        return Array(libs)
    }
    func load(game: SGFGame) { self.baseSize = game.boardSize;
self._baseSetup = game.setup; self._moves = game.moves;
self.initialPlayer = .black; self.serverMoveNumber = 0; self.reset() }
}

// FILE: ./Models/OGSModels.swift
// MARK: - File: OGSModels.swift (v4.246)
import Foundation
import SwiftUI
```

```
// MARK: - Core Go Models
enum Stone: String, Codable, CaseIterable {
    case black, white
    var opponent: Stone { self == .black ? .white : .black }
}

struct BoardPosition: Hashable, Codable {
    let row, col: Int
    init(_ row: Int, _ col: Int) { self.row = row; self.col = col }
}

struct BoardSnapshot: Equatable {
    let size: Int; let grid: [[Stone?]]; let stones: [BoardPosition:
Stone]
}

struct MoveRef: Equatable { let color: Stone; let x, y: Int }

struct RenderStone: Identifiable, Equatable {
    let id: BoardPosition; let color: Stone; let offset: CGPoint
}

// MARK: - OGS Network & Lobby Models
struct NetworkLogEntry: Identifiable { let id = UUID(); let timestamp
= Date(); let direction, content: String; let isHeartbeat: Bool }
struct ChallengerInfo: Codable, Hashable {
    let id: Int; let username: String; let ranking: Double?; let
professional: Bool?
    var displayRank: String {
        guard let r = ranking else { return "?" }
        let rank = Int(round(r)); return rank < 30 ? "\(30 - rank)k" :
"\(rank - 29)d"
    }
}
struct ChallengeGameInfo: Codable, Hashable { let ranked: Bool?; let
width, height: Int; let rules: String? }
struct OGSChallenge: Identifiable, Decodable {
    let id: Int; let name: String?; let challenger: ChallengerInfo?;
let game: ChallengeGameInfo?; let time_per_move: Int?
    enum CodingKeys: String, CodingKey { case challenge_id, game_id,
name, user_id, username, ranking, professional, width, height, ranked,
rules, time_per_move, black, white }
    init(from decoder: Decoder) throws {
        let c = try decoder.container(keyedBy: CodingKeys.self)
        self.id = (try? c.decode(Int.self, forKey: .challenge_id)) ??
(try? c.decode(Int.self, forKey: .game_id)) ?? 0
        self.name = try? c.decode(String.self, forKey: .name);
self.time_per_move = try? c.decode(Int.self, forKey: .time_per_move)
        if c.contains(.black) || c.contains(.white) {
            self.challenger = (try? c.decode(ChallengerInfo.self,
forKey: .black)) ?? (try? c.decode(ChallengerInfo.self,
forKey: .white))
        } else {
            self.challenger = ChallengerInfo(id: (try?
c.decode(Int.self, forKey: .user_id)) ?? 0, username: (try?
c.decode(String.self, forKey: .username)) ?? "Unknown", ranking: try?
c.decode(Double.self, forKey: .ranking), professional: try?
c.decode(Bool.self, forKey: .professional))
        }
        self.game = ChallengeGameInfo(ranked: try? c.decode(Bool.self,
forKey: .ranked), width: (try? c.decode(Int.self, forKey: .width)) ??
19, height: (try? c.decode(Int.self, forKey: .height)) ?? 19, rules:
try? c.decode(String.self, forKey: .rules))
    }
    var boardSize: String { "\(game?.width ?? 19)x\(game?.height ??
19)" }
    var speedCategory: String {
        guard let tpm = time_per_move else { return "live" }
        return tpm < 30 ? "blitz" : (tpm > 43200 ? "correspondence" :
"live")
    }
    var timeControlDisplay: String {
        guard let tpm = time_per_move else { return "No limit" }
        return tpm < 60 ? "\(tpm)s / move" : "\(tpm / 60)m / move"
    }
}

// MARK: - SGF Parser & Game Models
struct SGFNode { var props: [String:[String]] }
struct SGFTree { let nodes: [SGFNode] }
enum SGFError: Error { case parseError(String) }

enum SGFParser {
    static func parse(text: String) throws -> SGFTree {
        let s = text.replacingOccurrences(of: "\r", with: ""); var i =
s.startIndex
        func peek() -> Character? { i < s.endIndex ? s[i] : nil }
        func advance() { if i < s.endIndex { i = s.index(after: i) } }
        func skipWS() { while let cc = peek(), cc.isWhitespace
{ advance() } }
        var nodes: [SGFNode] = []; skipWS()
        guard peek() == "(" else { throw SGFError.parseError("Missing
'('") }
        func consume() throws {
            advance(); skipWS()
            while let c = peek() {
                if c == ";" { advance(); nodes.append(try
parseNode()); skipWS() }
                else if c == "(" { try skipSubtree(); skipWS() }
                else if c == ")" { advance(); break } else
```

```
{ advance() }
            }
        }
        func parseNode() throws -> SGFNode {
            var props: [String:[String]] = [:]; skipWS()
            while let c = peek(), c.isLetter {
                let sk = i; while let cc = peek(), cc.isLetter
{ advance() }
                let key = String(s[sk..<i]).uppercased(); var values:
[String] = []; skipWS()
                while peek() == "[" {
                    advance(); var val = ""
                    while let cc = peek() {
                        if cc == "\\" { advance(); if let nxt = peek()
{ val.append(nxt); advance() } }
                        else if cc == "]" { advance(); break } else
{ val.append(cc); advance() }
                    }
                    values.append(val); skipWS()
                }
                props[key] = values; skipWS()
            }
            return SGFNode(props: props)
        }
        func skipSubtree() throws { advance(); var d = 0; while let cc
= peek() { advance(); if cc == "(" { d += 1 } else if cc == ")" { if d
== 0 { break } else { d -= 1 } } } }
        try consume(); return SGFTree(nodes: nodes)
    }
}

struct SGFGame {
    struct Info { var event, playerBlack, playerWhite, blackRank,
whiteRank, result, date, komi: String? }
    var boardSize: Int = 19; var info: Info = .init(); var setup:
[(Stone, Int, Int)] = []; var moves: [(Stone, (Int,Int)?)] = []
    static func from(tree: SGFTree) -> SGFGame {
        var g = SGFGame()
        for node in tree.nodes {
            for (k, vals) in node.props {
                switch k {
                case "SZ": if let v = vals.first { g.boardSize =
Int(v.components(separatedBy: ":").first ?? "19") ?? 19 }
                case "AB": for v in vals { if let (x,y) =
SGFCoordinates.parse(v) { g.setup.append((.black, x, y)) } }
                case "AW": for v in vals { if let (x,y) =
SGFCoordinates.parse(v) { g.setup.append((.white, x, y)) } }
                case "B": g.moves.append((.black,
SGFCoordinates.parse(vals.first ?? "")))
                case "W": g.moves.append((.white,
SGFCoordinates.parse(vals.first ?? "")))
                default: continue
                }
            }
        }
        return g
    }
}

struct SGFGameWrapper: Identifiable, Hashable {
    let id = UUID(); let url: URL; let game: SGFGame
    var title: String? { if let e = game.info.event, !e.isEmpty
{ return e }; return url.lastPathComponent }
    static func == (l: SGFGameWrapper, r: SGFGameWrapper) -> Bool
{ l.id == r.id }; func hash(into h: inout Hasher) { h.combine(id) }
}

// MARK: - UI & Settings Models
struct GameInfo: Codable, Hashable { let id: Int; let name: String?;
let width, height: Int; let rules: String; let ranked: Bool; let
handicap: Int; let komi: String?; let started: String? }
enum ViewMode: String, CaseIterable, Identifiable { case view2D =
"2D", view3D = "3D"; var id: String { rawValue } }
enum OGSBrowserTab: String, CaseIterable { case challenge =
"Challenge", watch = "Watch" }
enum BoardSizeCategory: String, CaseIterable, Identifiable { case
size19 = "19", size13 = "13", size9 = "9", other = "Other"; var id:
String { rawValue } }
enum GameSpeedFilter: String, CaseIterable, Identifiable { case all =
"All Speeds", live = "Live", blitz = "Blitz", correspondence =
"Correspondence"; var id: String { rawValue } }

struct ChallengeSetup: Codable {
    var name = "Friendly Match"; var size = 19; var rules =
"japanese"; var ranked = true; var handicap = 0; var color =
"automatic"; var minRank = 0; var maxRank = 38; var timeControl =
"byoyomi"; var mainTime = 600; var periods = 5; var periodTime = 30;
var initialTime = 600; var increment = 30; var maxTime = 1200; var
perMove = 30
    func toDictionary() -> [String: Any] { ["game": ["name": name,
"rules": rules, "ranked": ranked, "width": size, "height": size],
"challenger_color": color] }
    static func load() -> ChallengeSetup { ChallengeSetup() }; func
save() {}
}

struct SGFCoordinates {
    static func parse(_ s: String) -> (Int, Int)? {
        guard s.count >= 2 else { return nil }; let chars =
Array(s.lowercased())
```

```
        let x = Int(chars[0].asciiValue ?? 0) - 97; let y = 
Int(chars[1].asciiValue ?? 0) - 97
        return (x >= 0 && x < 25 && y >= 0 && y < 25) ? (x, y) : nil
    }
    static func toSGF(x: Int, y: Int) -> String { "\
(Character(UnicodeScalar(97 + x)!))\(Character(UnicodeScalar(97 + 
y)!))" }
}

struct KeychainHelper {
    static func save(service: String, account: String, data: Data) -> 
OSStatus {
        let q: [CFString: Any] = [kSecClass: kSecClassGenericPassword, 
kSecAttrService: service as CFString, kSecAttrAccount: account as 
CFString, kSecValueData: data]
        SecItemDelete(q as CFDictionary); return SecItemAdd(q as 
CFDictionary, nil)
    }
    static func load(service: String, account: String) -> Data? {
        let q: [CFString: Any] = [kSecClass: kSecClassGenericPassword, 
kSecAttrService: service as CFString, kSecAttrAccount: account as 
CFString, kSecReturnData: kCFBooleanTrue!, kSecMatchLimit: 
kSecMatchLimitOne]
        var item: AnyObject?; let status = SecItemCopyMatching(q as 
CFDictionary, &item); return status == noErr ? (item as? Data) : nil
    }
}

// FILE: ./Models/DiagnosticTests.swift
// MARK: - File: DiagnosticTests.swift (v4.212)
//
//  Purpose: Diagnostic test views for debugging performance issues
//

import SwiftUI

// TEST 4: Full BoardViewModel WITH Combine subscriptions
struct Test4_FullBoardVM: View {
    @StateObject private var appModel = AppModel()
    @StateObject private var boardVM: BoardViewModel

    init() {
        let model = AppModel()
        _appModel = StateObject(wrappedValue: model)
        // Harmonized signature matching v8.100 architecture
        _boardVM = StateObject(wrappedValue: BoardViewModel(player: 
model.player, ogsClient: model.ogsClient))
        print("✅ Test 4: Full BoardViewModel initialized")
    }
    var body: some View {
        VStack {
            Text("Test 4: Full BoardViewModel WITH Combine")
                .foregroundColor(.white)
            // Performance Fix: Use optimized stonesToRender count
            Text("Stones (Cache): \(boardVM.stonesToRender.count)")
                .foregroundColor(.white)
            Text("Move: \(boardVM.currentMoveIndex)")
                .foregroundColor(.white)
        }
        .frame(maxWidth: .infinity, maxHeight: .infinity)
        .background(Color.black)
        .onAppear { print("✅ Test 4: Full BoardVM appeared") }
    }
}

// Minimal Test 1
struct Test1_Minimal: View {
    var body: some View {
        Text("Test 1: Minimal - No dependencies")
            .foregroundColor(.white)
            .frame(maxWidth: .infinity, maxHeight: .infinity)
            .background(Color.black)
            .onAppear { print("✅ Test 1: Minimal appeared") }
    }
}

// Test 2
struct Test2_AppModel: View {
    @StateObject private var appModel = AppModel()
    var body: some View {
        Text("Test 2: AppModel exists (not used)")
            .foregroundColor(.white)
            .frame(maxWidth: .infinity, maxHeight: .infinity)
            .background(Color.black)
            .onAppear { print("✅ Test 2: AppModel test appeared") }
    }
}

// Test 3
struct Test3_BoardVMNoCombine: View {
    @StateObject private var boardVM = BoardViewModel_NoCombine()
    var body: some View {
        VStack {
            Text("Test 3: BoardViewModel WITHOUT Combine")
                .foregroundColor(.white)
            Text("Stones: \(boardVM.stones.count)")
                .foregroundColor(.white)
        }
        .frame(maxWidth: .infinity, maxHeight: .infinity)
        .background(Color.black)
        .onAppear { print("✅ Test 3: BoardVM (no Combine) 
appeared") }
    }
}

class BoardViewModel_NoCombine: ObservableObject {
    @Published var stones: [BoardPosition: Stone] = [:]
    @Published var currentMoveIndex: Int = 0
    init() {}
}

// FILE: ./Models/CameraControlHandler.swift
// MARK: - File: CameraControlHandler.swift (v2.105)
import SwiftUI
import SceneKit

struct CameraControlHandler: View {
    @Binding var rotationX: Float
    @Binding var rotationY: Float
    @Binding var distance: CGFloat
    @Binding var panX: CGFloat
    @Binding var panY: CGFloat
    let sceneManager: SceneManager3D
    let onInteractionEnded: () -> Void // Signal to save state

    @State private var lastDragPosition: CGPoint = .zero
    @State private var isDragging: Bool = false

    var body: some View {
        Color.clear
            .contentShape(Rectangle())
            .gesture(
                DragGesture(minimumDistance: 0)
                    .onChanged { value in handleDrag(value) }
                    .onEnded { _ in
                        isDragging = false
                        onInteractionEnded()
                    }
            )
            .gesture(
                MagnificationGesture()
                    .onChanged { value in handleZoom(value) }
                    .onEnded { _ in onInteractionEnded() }
            )
    }

    private func handleDrag(_ value: DragGesture.Value) {
        let currentPosition = value.location
        if !isDragging {
            lastDragPosition = currentPosition
            isDragging = true
            return
        }
        let delta = CGPoint(
            x: currentPosition.x - lastDragPosition.x,
            y: currentPosition.y - lastDragPosition.y
        )

        #if os(macOS)
        let flags = NSEvent.modifierFlags
        let isShiftPressed = flags.contains(.shift)
        let isControlPressed = flags.contains(.control)

        if isShiftPressed && isControlPressed {
            // ZOOM: Ctrl + Shift + Drag
            let zoomSpeed: CGFloat = 0.1
            distance -= delta.y * zoomSpeed
            distance = max(10.0, min(100.0, distance))
        } else if isShiftPressed {
            // TRANSLATION: Shift + Drag
            // Pulling Board (+dx) -> Pushes board right
            // Pulling Board (+dy) -> Pushes board toward user
            let panSpeed: CGFloat = 0.05
            panX += delta.x * panSpeed
            panY += delta.y * panSpeed
        } else {
            // ROTATION: Pulling the board world
            let rotationSpeed: Float = 0.005
            // Mouse Down (+dy) -> Increases X rotation (tilts board
forward)
            rotationX += Float(delta.y) * rotationSpeed
            // Mouse Right (+dx) -> Increases Y rotation (turns board
clockwise)
            rotationY += Float(delta.x) * rotationSpeed

            // Limit: 0.05 (side on) to 1.57 (90 deg top down)
            rotationX = max(0.05, min(1.57, rotationX))
        }
        #endif

        sceneManager.updateCameraPosition(
            distance: distance,
            rotationX: rotationX,
            rotationY: rotationY,
            panX: panX,
            panY: panY
        )
        lastDragPosition = currentPosition
    }
    private func handleZoom(_ value: MagnificationGesture.Value) {
        let newDistance = distance / value
        distance = max(10.0, min(100.0, newDistance))
        sceneManager.updateCameraPosition(
            distance: distance,
            rotationX: rotationX,
            rotationY: rotationY,
            panX: panX,
            panY: panY
        )
    }
}

// FILE: ./Models/SceneManager3D.swift
// MARK: - File: SceneManager3D.swift (v4.974)
import Foundation
import SceneKit
import AppKit
import QuartzCore

class SceneManager3D: ObservableObject {
    let scene = SCNScene(); let cameraNode = SCNNode(); let pivotNode
= SCNNode()
    private let worldAnchor = SCNNode()
    private let stonesContainer = SCNNode()

    private var stoneNodeMap: [BoardPosition: SCNNode] = [:]
    private var blackStoneGeometry: SCNGeometry?; private var
whiteStoneGeometry: SCNGeometry?
    private var markerMaterial: SCNMaterial?; private var
glowMaterial: SCNMaterial?
    private var boardNode: SCNNode?; private var ghostNode: SCNNode?

    private var upperLidStones: [SCNNode] = []; private var
lowerLidStones: [SCNNode] = []
    private var upperLidNode: SCNNode?; private var lowerLidNode:
SCNNode?

    private var boardSize: Int = 19; private let boardThickness:
CGFloat = 2.0
    private let stoneRadius: CGFloat = 0.48; private let stoneScaleY:
CGFloat = 0.486; private var previousLastMove: BoardPosition?
    private var effectiveCellWidth: CGFloat { CGFloat(18) /
CGFloat(max(1, boardSize - 1)) }
    private var effectiveCellHeight: CGFloat { (CGFloat(18) /
CGFloat(max(1, boardSize - 1))) * 1.0773 }

    init() {
        scene.rootNode.addChildNode(worldAnchor)
        scene.rootNode.addChildNode(pivotNode)
        pivotNode.addChildNode(cameraNode)
        worldAnchor.addChildNode(stonesContainer)
        setupMaterials(); setupCamera(); setupLighting();
setupBackground(); createBoard(); createLids(); setupGhostNode()
    }

    private func setupMaterials() {
        blackStoneGeometry = SCNSphere(radius: stoneRadius); let bM =
SCNMaterial(); bM.diffuse.contents = NSColor(white: 0.1, alpha: 1.0);
bM.specular.contents = NSColor(white: 0.3, alpha: 1.0);
bM.lightingModel = .blinn; blackStoneGeometry?.materials = [bM]
        whiteStoneGeometry = SCNSphere(radius: stoneRadius); let wM =
SCNMaterial(); wM.diffuse.contents = NSColor(white: 0.95, alpha: 1.0);
wM.specular.contents = NSColor(white: 1.0, alpha: 1.0);
wM.lightingModel = .blinn; whiteStoneGeometry?.materials = [wM]
        markerMaterial = SCNMaterial();
markerMaterial?.diffuse.contents = NSColor.red;
markerMaterial?.emission.contents = NSColor.red
        glowMaterial = SCNMaterial(); glowMaterial?.lightingModel
= .constant; glowMaterial?.blendMode = .alpha;
glowMaterial?.diffuse.contents = generateRedGlowTexture();
glowMaterial?.writesToDepthBuffer = false
    }

    private func generateRedGlowTexture() -> NSImage {
        let size = 128; return NSImage(size: NSSize(width: size,
height: size), flipped: false) { rect in
            guard let ctx = NSGraphicsContext.current?.cgContext else
{ return false }
            let center = CGPoint(x: size/2, y: size/2)
            let colors =
[NSColor.red.withAlphaComponent(0.75).cgColor,
NSColor.red.withAlphaComponent(0.0).cgColor] as CFArray
            let grad = CGGradient(colorsSpace:
CGColorSpaceCreateDeviceRGB(), colors: colors, locations: [0.0, 1.0])
            ctx.drawRadialGradient(grad!, startCenter: center,
startRadius: 0, endCenter: center, endRadius: CGFloat(size/2),
options: .drawsBeforeStartLocation); return true
        }
    }

    func updateStones(from cache: [RenderStone], lastMove:
BoardPosition?, moveIndex: Int, settings: AppSettings) {
        SCNTransaction.begin(); SCNTransaction.animationDuration = 0
        let w = CGFloat(boardSize - 1) * effectiveCellWidth; let h =
CGFloat(boardSize - 1) * effectiveCellHeight; let offX = -w / 2.0; let
offZ = -h / 2.0; let surfaceY = boardThickness / 2.0
        let currentPos = Set(cache.map { $0.id }); for (pos, node) in
stoneNodeMap { if !currentPos.contains(pos)
```

```
{ node.removeFromParentNode(); stoneNodeMap.removeValue(forKey:
pos) } }
        for rs in cache {
            let x = CGFloat(rs.id.col) * effectiveCellWidth + offX +
(rs.offset.x * effectiveCellWidth); let z = CGFloat(rs.id.row) *
effectiveCellHeight + offZ + (rs.offset.y * effectiveCellHeight)
            if let n = stoneNodeMap[rs.id] { n.position =
SCNVector3(x, surfaceY, z) } else {
                let anchor = SCNNode(); anchor.position =
SCNVector3(x, surfaceY, z)
                let stone = SCNNode(geometry: rs.color == .black ?
blackStoneGeometry : whiteStoneGeometry)
                stone.scale = SCNVector3(1, stoneScaleY, 1);
stone.position = SCNVector3(0, stoneRadius * stoneScaleY, 0)
                anchor.addChildNode(stone);
stonesContainer.addChildNode(anchor); stoneNodeMap[rs.id] = anchor
                if settings.showDropInAnimation { anchor.opacity = 0;
anchor.position.y += 1.0; anchor.runAction(.group([.fadeIn(duration:
0.15), .move(to: SCNVector3(x, surfaceY, z), duration: 0.15)])) }
            }
        }
        if previousLastMove != lastMove || true {
            if let p = previousLastMove, let n = stoneNodeMap[p]
{ n.childNodes.filter({ $0.name?.contains("MARKER") ??
false }).forEach { $0.removeFromParentNode() } }
            if let c = lastMove, let n = stoneNodeMap[c] {
                let color = cache.first(where: { $0.id ==
c })?.color ?? .black
                applyMarkers(to: n, color: color, index: moveIndex,
settings: settings)
            }
            previousLastMove = lastMove
        }
        SCNTransaction.commit()
    }

    private func applyMarkers(to group: SCNNode, color: Stone, index:
Int, settings: AppSettings) {
        let markerApex = (stoneRadius * stoneScaleY * 2.0) + 0.01
        let textColor = color == .black ? NSColor.white :
NSColor.black
        if settings.showLastMoveDot { let d = SCNNode(geometry:
SCNSphere(radius: stoneRadius * 0.12)); d.geometry?.firstMaterial =
markerMaterial; d.name = "MARKER_DOT"; d.position = SCNVector3(0,
markerApex + 0.02, 0); group.addChildNode(d) }
        if settings.showLastMoveCircle { let r = SCNNode(geometry:
SCNTorus(ringRadius: stoneRadius * 0.6, pipeRadius: 0.025));
r.geometry?.firstMaterial = markerMaterial; r.name = "MARKER_CIRCLE";
r.position = SCNVector3(0, markerApex, 0); group.addChildNode(r) }
        if settings.showMoveNumbers {
            let t = SCNText(string: "\(index)", extrusionDepth: 0.1);
t.font = NSFont.boldSystemFont(ofSize: 0.45); t.flatness = 0.01;
t.materials = [SCNMaterial()]; t.materials[0].diffuse.contents =
textColor
            let tn = SCNNode(geometry: t); tn.name = "MARKER_NUMBER";
tn.eulerAngles.x = -.pi/2; let (min, max) = t.boundingBox
            tn.pivot = SCNMatrix4MakeTranslation(CGFloat((max.x -
min.x)/2.0 + min.x), CGFloat((max.y - min.y)/2.0 + min.y), 0)
            tn.position = SCNVector3(0, markerApex, 0);
group.addChildNode(tn)
        }
        if settings.showBoardGlow || settings.showEnhancedGlow {
            let s = settings.showEnhancedGlow ? 4.5 : 3.2; let p =
SCNNode(geometry: SCNPlane(width: stoneRadius * s, height: stoneRadius
* s)); p.geometry?.firstMaterial = glowMaterial; p.name =
"MARKER_GLOW"; p.eulerAngles.x = -.pi/2; p.position = SCNVector3(0,
0.02, 0); p.renderingOrder = 3000; group.addChildNode(p)
        }
    }

    func updateCameraPosition(distance: CGFloat, rotationX: Float,
rotationY: Float, panX: CGFloat, panY: CGFloat) {
        // Apply panning and rotation to the worldAnchor (Board) to
simulate "Pulling the board"
        worldAnchor.position = SCNVector3(x: panX, y: 0, z: panY)
        worldAnchor.eulerAngles.y = CGFloat(rotationY)
        worldAnchor.eulerAngles.x = CGFloat(rotationX)

        let r = distance / 25.0
        cameraNode.position = SCNVector3(x: 0, y: 15.0 * r, z: 20.0 *
r)
        cameraNode.look(at: SCNVector3(x: 0, y: 0, z: 0))
    }

    func updateCapturedStones(black: Int, white: Int) { guard let u =
upperLidNode, let l = lowerLidNode else { return };
SCNTransaction.begin(); SCNTransaction.animationDuration = 0;
updateLidDiff(lid: u, current: &upperLidStones, target: black, geom:
whiteStoneGeometry); updateLidDiff(lid: l, current: &lowerLidStones,
target: white, geom: blackStoneGeometry); SCNTransaction.commit() }
    private func updateLidDiff(lid: SCNNode, current: inout [SCNNode],
target: Int, geom: SCNGeometry?) { if current.count == target
{ return }; if current.count > target { while current.count > target
{ current.last?.removeFromParentNode(); current.removeLast() } } else
{ for i in current.count..<target { let n = SCNNode(geometry: geom);
let phi = 137.5 * (.pi / 180.0); let r = (3.5 * 0.7) *
sqrt(Double(i) / 100.0); let th = Double(i) * phi; n.position =
SCNVector3(x: CGFloat(cos(th)*r), y: 0.15, z: CGFloat(sin(th)*r));
lid.addChildNode(n); current.append(n) } } }
    private func setupCamera() { let c = SCNCamera(); c.zNear = 0.1;
```

```
c.zFar = 1000.0; cameraNode.camera = c;
pivotNode.addChildNode(cameraNode); updateCameraPosition(distance:
25.0, rotationX: 0.75, rotationY: 0.0, panX: 0, panY: 0) }
    private func setupLighting() { let amb = SCNNode(); amb.light =
SCNLight(); amb.light?.type = .ambient; amb.light?.color =
NSColor(white: 0.4, alpha: 1.0); worldAnchor.addChildNode(amb); let
dir = SCNNode(); dir.light = SCNLight(); dir.light?.type
= .directional; dir.light?.color = NSColor(white: 0.8, alpha: 1.0);
dir.light?.castsShadow = true; dir.position = SCNVector3(-10, 20,
-10); dir.look(at: SCNVector3(x: 0, y: 0, z: 0));
worldAnchor.addChildNode(dir) }
    private func setupBackground() { scene.background.contents =
NSColor(red: 0.01, green: 0.01, blue: 0.05, alpha: 1.0) }
    private func createBoard() { boardNode?.removeFromParentNode();
let bW = CGFloat(boardSize + 1) * effectiveCellWidth; let bL =
CGFloat(boardSize + 1) * effectiveCellHeight; let bG = SCNBox(width:
bW, height: boardThickness, length: bL, chamferRadius: 0.0); let m =
SCNMaterial(); m.diffuse.contents = NSImage(named: "board_kaya") ??
NSColor.brown; bG.materials = [m]; let n = SCNNode(geometry: bG);
worldAnchor.addChildNode(n); self.boardNode = n; createGridLines() }
    private func createGridLines() { let tY = boardThickness / 2.0 +
0.02; let w = CGFloat(boardSize - 1) * effectiveCellWidth; let h =
CGFloat(boardSize - 1) * effectiveCellHeight; for i in 0..<boardSize
{ let lZ = SCNBox(width: w, height: 0.002, length: 0.02,
chamferRadius: 0); lZ.firstMaterial?.diffuse.contents = NSColor.black;
let nZ = SCNNode(geometry: lZ); nZ.position = SCNVector3(x: 0, y: tY,
z: CGFloat(i) * effectiveCellHeight - (h/2.0));
worldAnchor.addChildNode(nZ); let lX = SCNBox(width: 0.02, height:
0.002, length: h, chamferRadius: 0);
lX.firstMaterial?.diffuse.contents = NSColor.black; let nX =
SCNNode(geometry: lX); nX.position = SCNVector3(x: CGFloat(i) *
effectiveCellWidth - (w/2.0), y: tY, z: 0);
worldAnchor.addChildNode(nX) } }
    func createLids() { upperLidNode?.removeFromParentNode();
lowerLidNode?.removeFromParentNode(); upperLidNode =
createLidNode(textureName: "go_lid_1", pos: SCNVector3(x: 14.0, y:
-0.2, z: -5.0)); lowerLidNode = createLidNode(textureName: "go_lid_2",
pos: SCNVector3(x: 14.0, y: -0.2, z: 5.0)); if let u = upperLidNode
{ worldAnchor.addChildNode(u) }; if let l = lowerLidNode
{ worldAnchor.addChildNode(l) } }
    private func createLidNode(textureName: String, pos: SCNVector3)
-> SCNNode { let cyl = SCNCylinder(radius: 3.5, height: 0.3); let m =
SCNMaterial(); m.diffuse.contents = NSImage(named: textureName) ??
NSColor.brown; cyl.materials = [m]; let n = SCNNode(geometry: cyl);
n.position = pos; return n }
    func hitTest(point: CGPoint, in view: SCNView) -> (x: Int, y:
Int)? { guard let res = view.hitTest(point, options: [.searchMode:
SCNHitTestSearchMode.all.rawValue]).first(where: { $0.node ==
self.boardNode }) else { return nil }; let w = CGFloat(boardSize - 1)
* effectiveCellWidth; let h = CGFloat(boardSize - 1) *
```

```
effectiveCellHeight; let c =
Int(round((CGFloat(res.localCoordinates.x) + w/2.0) /
effectiveCellWidth)); let r =
Int(round((CGFloat(res.localCoordinates.z) + h/2.0) /
effectiveCellHeight)); return (c >= 0 && c < boardSize && r >= 0 && r
< boardSize) ? (c, r) : nil }
    private func setupGhostNode() { let s = SCNSphere(radius:
stoneRadius); let m = SCNMaterial(); m.diffuse.contents =
NSColor(white: 1.0, alpha: 0.5); s.materials = [m]; ghostNode =
SCNNode(geometry: s); ghostNode?.opacity = 0.0;
stonesContainer.addChildNode(ghostNode!) }
}

// FILE: ./Models/StoneMeshFactory.swift
import SceneKit

class StoneMeshFactory {

    /// Generates a custom SCNGeometry for a Go stone with top-down
(Planar) UV mapping.
    static func createStoneGeometry(radius: CGFloat, thickness:
CGFloat) -> SCNGeometry {
        let steps = 32 // Horizontal resolution (higher = smoother
roundness)
        let layers = 16 // Vertical resolution (higher = smoother
curve)

        var vertices: [SCNVector3] = []
        var normals: [SCNVector3] = []
        var texCoords: [CGPoint] = []
        var indices: [Int32] = []

        // 1. Generate Vertices & UVs
        for i in 0...layers {
            // Calculate vertical slice (latitude)
            // theta goes from -PI/2 (bottom) to PI/2 (top)
            let theta = Float(i) * Float.pi / Float(layers) -
(Float.pi / 2)
            let y = sin(theta) * Float(thickness) * 0.5
            let ringRadius = cos(theta) * Float(radius)

            for j in 0...steps {
                // Calculate horizontal position (longitude)
                let phi = Float(j) * 2 * Float.pi / Float(steps)

                let x = cos(phi) * ringRadius
                let z = sin(phi) * ringRadius

                // A. Position
                vertices.append(SCNVector3(x, y, z))
                // B. Normal (Same as position for a sphere/ellipsoid,
normalized)
                let normal = SCNVector3(x, y / Float(thickness) *
Float(radius) * 2, z).normalized()
                normals.append(normal)

                // C. UV Mapping (CRITICAL STEP)
                // Instead of wrapping around, we project from the top
down.
                // Map X/Z range [-radius, radius] -> [0, 1]
                let u = CGFloat(x / Float(radius) * 0.5 + 0.5)
                let v = CGFloat(z / Float(radius) * 0.5 + 0.5) // Flip
V if texture is upside down

                // Clamp UVs to prevent edge bleeding artifacts
                let clampedU = min(max(u, 0.01), 0.99)
                let clampedV = min(max(v, 0.01), 0.99)

                // Only apply texture to the top half to avoid bottom
mirroring weirdness
                if y > -0.1 {
                    texCoords.append(CGPoint(x: clampedU, y:
clampedV))
                } else {
                    // Bottom half gets a generic white/black pixel
(center of texture)
                    texCoords.append(CGPoint(x: 0.5, y: 0.5))
                }
            }
        }

        // 2. Generate Indices (Triangles)
        for i in 0..<layers {
            for j in 0..<steps {
                let current = Int32(i * (steps + 1) + j)
                let next = Int32(current + 1)
                let above = Int32((i + 1) * (steps + 1) + j)
                let aboveNext = Int32(above + 1)

                // Triangle 1
                indices.append(current)
                indices.append(next)
                indices.append(above)

                // Triangle 2
                indices.append(next)
                indices.append(aboveNext)
                indices.append(above)
            }
        }

        // 3. Create Geometry Sources
        let vertexSource = SCNGeometrySource(vertices: vertices)
        let normalSource = SCNGeometrySource(normals: normals)
        let uvSource = SCNGeometrySource(textureCoordinates:
texCoords)

        let element = SCNGeometryElement(indices: indices,
primitiveType: .triangles)

        let geometry = SCNGeometry(sources: [vertexSource,
normalSource, uvSource], elements: [element])
        return geometry
    }
}

extension SCNVector3 {
    func normalized() -> SCNVector3 {
        let len = sqrt(x*x + y*y + z*z)
        if len == 0 { return self }
        return SCNVector3(x/len, y/len, z/len)
    }
}

// FILE: ./Models/SGFPlayerCleanApp.swift
//
//  SGFPlayerCleanApp.swift
//  SGFPlayerClean
//
//  Purpose: Main application entry point
//  Now with 2D/3D mode switching
//

import SwiftUI

@main
struct SGFPlayerCleanApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
        .commands {
            CommandGroup(replacing: .newItem) {
                Button("Open SGF File...") {
                    openSGFFile()
                }
                .keyboardShortcut("o", modifiers: .command)
            }
        }
    }

    /// Open SGF file picker
    private func openSGFFile() {
        print("📂 Opening file picker...")
        let panel = NSOpenPanel()
        panel.allowsMultipleSelection = false
        panel.canChooseDirectories = false
        panel.canChooseFiles = true
        panel.allowedContentTypes = [.init(filenameExtension: "sgf")!]
        panel.title = "Choose an SGF file"

        panel.begin { response in
            print("📂 File picker response: \(response == .OK ? "OK" :
"Cancel")")
            if response == .OK, let url = panel.url {
                print("📂 Selected file: \(url.path)")
                print("📂 Posting notification...")
                NotificationCenter.default.post(
                    name: NSNotification.Name("LoadSGFFile"),
                    object: url
                )
                print("📂 Notification posted")
            } else {
                print("📂 No file selected or picker cancelled")
            }
        }
    }
}


// FILE: ./Models/GameSettings.swift
import Foundation
import SwiftUI

// MARK: - Game Settings for Live Play

/// Rules for the game
enum GameRules: String, CaseIterable, Identifiable, Codable {
    case japanese = "Japanese"
    case chinese = "Chinese"
    case aga = "AGA"
    case korean = "Korean"
    case newZealand = "New Zealand"
    case ing = "Ing"

    var id: String { rawValue }
    var apiValue: String { rawValue.lowercased() }
}
```

```swift
/// Time control systems
enum TimeControlSystem: String, CaseIterable, Identifiable, Codable {
    case fischer = "Fischer"
    case byoyomi = "Byo-Yomi"
    case canadian = "Canadian"
    case simple = "Simple"
    case absolute = "Absolute"
    case none = "None"

    var id: String { rawValue }
    var apiValue: String {
        switch self {
        case .byoyomi: return "byoyomi"
        case .fischer: return "fischer"
        case .canadian: return "canadian"
        case .simple: return "simple"
        case .absolute: return "absolute"
        case .none: return "none"
        }
    }
}

/// Handicap options
enum HandicapOption: String, CaseIterable, Identifiable, Codable {
    case automatic = "Automatic"
    case none = "None"
    case two = "2"
    case three = "3"
    case four = "4"
    case five = "5"
    case six = "6"
    case seven = "7"
    case eight = "8"
    case nine = "9"

    var id: String { rawValue }
    var apiValue: Int {
        switch self {
        case .automatic: return -1
        case .none: return 0
        case .two: return 2
        case .three: return 3
        case .four: return 4
        case .five: return 5
        case .six: return 6
        case .seven: return 7
        case .eight: return 8
        case .nine: return 9
        }
    }
}

/// Komi options
enum KomiOption: String, CaseIterable, Identifiable, Codable {
    case automatic = "Automatic"
    case custom = "Custom"

    var id: String { rawValue }
}

/// Color preference for matchmaking
enum ColorPreference: String, CaseIterable, Identifiable, Codable {
    case automatic = "Automatic"
    case black = "Black"
    case white = "White"

    var id: String { rawValue }

    /// Value for OGS API ("automatic", "black", "white")
    var apiValue: String { rawValue.lowercased() }
}

/// Main time options (in minutes)
let mainTimeOptions = [1, 2, 3, 5, 10, 15, 20, 30, 45, 60, 90]

/// Time per period options (in seconds) - for Byo-Yomi
let periodTimeOptions = [5, 10, 15, 20, 30, 45, 60, 90, 120]

/// Period count options - for Byo-Yomi
let periodCountOptions = [1, 2, 3, 5, 7, 10]

/// Fischer time increment options (in seconds)
let fischerIncrementOptions = [5, 10, 15, 20, 30, 45, 60]

/// Fischer max time options (in minutes)
let fischerMaxTimeOptions = [5, 10, 15, 20, 30, 45, 60, 90, 120]

/// Rank restriction for above/below
enum RankRestriction: Codable, Equatable {
    case any
    case limit(Int)

    var displayValue: String {
        switch self {
        case .any: return "Any"
        case .limit(let value): return "\(value)"
        }
    }
}
```

```
/// Game settings for live play on OGS
struct GameSettings: Codable {
    // Basic settings
    var gameName: String
    var inviteOnly: Bool
    var rules: GameRules

    // Board settings
    var boardSize: Int
    var ranked: Bool

    // Handicap & Komi
    var handicap: HandicapOption
    var komi: KomiOption
    var customKomi: Double

    // Time control - Byo-Yomi/Canadian/Simple
    var timeControlSystem: TimeControlSystem
    var mainTimeMinutes: Int
    var periodTimeSeconds: Int  // Byo-Yomi: time per period
    var periods: Int             // Byo-Yomi: number of periods

    // Time control - Fischer
    var fischerIncrementSeconds: Int  // Time added per move
    var fischerMaxTimeMinutes: Int    // Maximum time that can
accumulate

    // Player settings
    var colorPreference: ColorPreference
    var disableAnalysis: Bool

    // Rank restrictions
    var restrictRank: Bool
    var ranksAbove: RankRestriction
    var ranksBelow: RankRestriction

    /// Computed: derive game speed from time settings
    var gameSpeed: String {
        let totalSeconds = mainTimeMinutes * 60
        if totalSeconds < 10 * 60 {
            return "blitz"
        } else if totalSeconds < 20 * 60 {
            return "rapid"
        } else if totalSeconds < 24 * 60 * 60 {
            return "live"
        } else {
            return "correspondence"
        }
    }
    /// Default settings for quick play
    static let `default` = GameSettings(
        gameName: "SGFPlayer3D Game",
        inviteOnly: false,
        rules: .japanese,
        boardSize: 19,
        ranked: true,
        handicap: .automatic,
        komi: .automatic,
        customKomi: 6.5,
        timeControlSystem: .byoyomi,
        mainTimeMinutes: 5,
        periodTimeSeconds: 30,
        periods: 5,
        fischerIncrementSeconds: 30,
        fischerMaxTimeMinutes: 10,
        colorPreference: .automatic,
        disableAnalysis: false,
        restrictRank: false,
        ranksAbove: .any,
        ranksBelow: .any
    )

    // MARK: - UserDefaults Keys
    private static let gameNameKey = "gameSettings.gameName"
    private static let inviteOnlyKey = "gameSettings.inviteOnly"
    private static let rulesKey = "gameSettings.rules"
    private static let boardSizeKey = "gameSettings.boardSize"
    private static let rankedKey = "gameSettings.ranked"
    private static let handicapKey = "gameSettings.handicap"
    private static let komiKey = "gameSettings.komi"
    private static let customKomiKey = "gameSettings.customKomi"
    private static let timeControlSystemKey =
"gameSettings.timeControlSystem"
    private static let mainTimeMinutesKey =
"gameSettings.mainTimeMinutes"
    private static let periodTimeSecondsKey =
"gameSettings.periodTimeSeconds"
    private static let periodsKey = "gameSettings.periods"
    private static let fischerIncrementSecondsKey =
"gameSettings.fischerIncrementSeconds"
    private static let fischerMaxTimeMinutesKey =
"gameSettings.fischerMaxTimeMinutes"
    private static let colorPreferenceKey =
"gameSettings.colorPreference"
    private static let disableAnalysisKey =
"gameSettings.disableAnalysis"
    private static let restrictRankKey = "gameSettings.restrictRank"

    /// Load settings from UserDefaults
    static func load() -> GameSettings {
        let gameName = UserDefaults.standard.string(forKey:
gameNameKey) ?? "SGFPlayer3D Game"
        let inviteOnly = UserDefaults.standard.bool(forKey:
inviteOnlyKey)

        let rulesRaw = UserDefaults.standard.string(forKey:
rulesKey) ?? GameRules.japanese.rawValue
        let rules = GameRules(rawValue: rulesRaw) ?? .japanese

        let boardSize = UserDefaults.standard.object(forKey:
boardSizeKey) as? Int ?? 19
        let ranked = UserDefaults.standard.object(forKey: rankedKey)
as? Bool ?? true

        let handicapRaw = UserDefaults.standard.string(forKey:
handicapKey) ?? HandicapOption.automatic.rawValue
        let handicap = HandicapOption(rawValue:
handicapRaw) ?? .automatic

        let komiRaw = UserDefaults.standard.string(forKey: komiKey) ??
KomiOption.automatic.rawValue
        let komi = KomiOption(rawValue: komiRaw) ?? .automatic
        let customKomi = UserDefaults.standard.double(forKey:
customKomiKey)

        let timeControlRaw = UserDefaults.standard.string(forKey:
timeControlSystemKey) ?? TimeControlSystem.byoyomi.rawValue
        let timeControlSystem = TimeControlSystem(rawValue:
timeControlRaw) ?? .byoyomi

        let mainTimeMinutes = UserDefaults.standard.object(forKey:
mainTimeMinutesKey) as? Int ?? 5
        let periodTimeSeconds = UserDefaults.standard.object(forKey:
periodTimeSecondsKey) as? Int ?? 30
        let periods = UserDefaults.standard.object(forKey: periodsKey)
as? Int ?? 5
        let fischerIncrementSeconds =
UserDefaults.standard.object(forKey: fischerIncrementSecondsKey) as?
Int ?? 30
        let fischerMaxTimeMinutes =
UserDefaults.standard.object(forKey: fischerMaxTimeMinutesKey) as?
Int ?? 10

        let colorPrefRaw = UserDefaults.standard.string(forKey:
colorPreferenceKey) ?? ColorPreference.automatic.rawValue
        let colorPreference = ColorPreference(rawValue:
colorPrefRaw) ?? .automatic

        let disableAnalysis = UserDefaults.standard.bool(forKey:
```

```
disableAnalysisKey)
        let restrictRank = UserDefaults.standard.bool(forKey:
restrictRankKey)

        return GameSettings(
            gameName: gameName,
            inviteOnly: inviteOnly,
            rules: rules,
            boardSize: boardSize,
            ranked: ranked,
            handicap: handicap,
            komi: komi,
            customKomi: customKomi == 0 ? 6.5 : customKomi,
            timeControlSystem: timeControlSystem,
            mainTimeMinutes: mainTimeMinutes,
            periodTimeSeconds: periodTimeSeconds,
            periods: periods,
            fischerIncrementSeconds: fischerIncrementSeconds,
            fischerMaxTimeMinutes: fischerMaxTimeMinutes,
            colorPreference: colorPreference,
            disableAnalysis: disableAnalysis,
            restrictRank: restrictRank,
            ranksAbove: .any,
            ranksBelow: .any
        )
    }

    /// Save settings to UserDefaults
    func save() {
        UserDefaults.standard.set(gameName, forKey: Self.gameNameKey)
        UserDefaults.standard.set(inviteOnly, forKey:
Self.inviteOnlyKey)
        UserDefaults.standard.set(rules.rawValue, forKey:
Self.rulesKey)
        UserDefaults.standard.set(boardSize, forKey:
Self.boardSizeKey)
        UserDefaults.standard.set(ranked, forKey: Self.rankedKey)
        UserDefaults.standard.set(handicap.rawValue, forKey:
Self.handicapKey)
        UserDefaults.standard.set(komi.rawValue, forKey: Self.komiKey)
        UserDefaults.standard.set(customKomi, forKey:
Self.customKomiKey)
        UserDefaults.standard.set(timeControlSystem.rawValue, forKey:
Self.timeControlSystemKey)
        UserDefaults.standard.set(mainTimeMinutes, forKey:
Self.mainTimeMinutesKey)
        UserDefaults.standard.set(periodTimeSeconds, forKey:
Self.periodTimeSecondsKey)
        UserDefaults.standard.set(periods, forKey: Self.periodsKey)
        UserDefaults.standard.set(fischerIncrementSeconds, forKey:
```

```
Self.fischerIncrementSecondsKey)
        UserDefaults.standard.set(fischerMaxTimeMinutes, forKey:
Self.fischerMaxTimeMinutesKey)
        UserDefaults.standard.set(colorPreference.rawValue, forKey:
Self.colorPreferenceKey)
        UserDefaults.standard.set(disableAnalysis, forKey:
Self.disableAnalysisKey)
        UserDefaults.standard.set(restrictRank, forKey:
Self.restrictRankKey)
    }
}

// FILE: ./Views/ContentView2D.swift
// MARK: - File: ContentView2D.swift (v8.100)
import SwiftUI

struct ContentView2D: View {
    @EnvironmentObject var app: AppModel
    private let boardAspectRatio: CGFloat = 1.0773
    private let sidebarW: CGFloat = 320

    private func getLayout(area: CGSize) -> (bW: CGFloat, bH: CGFloat,
lidD: CGFloat, gap: CGFloat) {
        let totalWFactor: CGFloat = 1.412
        let bH = min((area.width * 0.88 / totalWFactor) *
boardAspectRatio, area.height * 0.78)
        let bW = bH / boardAspectRatio
        return (bW, bH, bH / 3.0, bW / 19.0)
    }

    var body: some View {
        GeometryReader { window in
            let area = CGSize(width: window.size.width - sidebarW,
height: window.size.height)
            let L = getLayout(area: area)
            let margin = L.bW * 0.065

            ZStack {
                TatamiBackground(boardHeight: L.bH)
                HStack(spacing: 0) {
                    VStack(spacing: 0) {
                        Spacer()
                        HStack(alignment: .top, spacing: L.gap) {
                            if let bvm = app.boardVM {
                                BoardView2D(boardVM: bvm, layoutVM:
app.layoutVM, size: CGSize(width: L.bW, height: L.bH))
                                    .frame(width: L.bW, height: L.bH)
                                VStack(spacing: 0) {
                                    // WHITE PLAYER SIDE (TOP):
Captured Black stones
```

```
                                    SimpleLidView(stoneColor: .black,
stoneCount: bvm.whiteCapturedCount, stoneSize: L.lidD * 0.15,
lidNumber: 1, lidSize: L.lidD)
                                        .padding(.top, margin -
(L.lidD / 2))
                                    Spacer()
                                    // BLACK PLAYER SIDE (BOTTOM):
Captured White stones
                                    SimpleLidView(stoneColor: .white,
stoneCount: bvm.blackCapturedCount, stoneSize: L.lidD * 0.15,
lidNumber: 2, lidSize: L.lidD)
                                        .padding(.bottom, margin -
(L.lidD / 2))
                                }
                                .frame(height: L.bH).frame(width:
L.lidD)
                            }
                        }
                        .frame(width: L.bW + L.gap + L.lidD)
                        Spacer()
                        if let bvm = app.boardVM {
                            PlaybackControlsView(boardVM:
bvm).padding(EdgeInsets(top: 0, leading: 0, bottom: 25, trailing: 0))
                        }
                    }
                    .frame(width: area.width)
                    RightPanelView().frame(width:
sidebarW).background(Color.black.opacity(0.15))
                }
            }.keyboardShortcuts(boardVM: app.boardVM!)
        }
    }
}

struct PlaybackControlsView: View {
    @ObservedObject var boardVM: BoardViewModel
    @State private var localIdx: Double = 0
    var body: some View {
        HStack(spacing: 12) {
            HStack(spacing: 6) {
                Button(action: { boardVM.goToStart() })
{ Image(systemName: "backward.end.fill") }.buttonStyle(.plain)
                Button(action: { boardVM.stepBackward() })
{ Image(systemName: "backward.fill") }.buttonStyle(.plain)
                Button(action: { boardVM.toggleAutoPlay() })
{ Image(systemName: boardVM.isAutoPlaying ? "pause.fill" :
"play.fill") }.buttonStyle(.plain)
                Button(action: { boardVM.stepForward() })
{ Image(systemName: "forward.fill") }.buttonStyle(.plain)
                Button(action: { boardVM.goToEnd() })
```

```
{ Image(systemName: "forward.end.fill") }.buttonStyle(.plain)
            }
            .font(.system(size: 13, weight: .bold))
            Divider().frame(height:
14).background(Color.white.opacity(0.3))
            Slider(value: $localIdx, in: 0...Double(max(1,
boardVM.totalMoves)), onEditingChanged: { dragging in
                if !dragging { boardVM.seekToMove(Int(localIdx)) }
            })
            .onChange(of: localIdx) { _, val in
boardVM.seekToMove(Int(val)) }
            .onChange(of: boardVM.currentMoveIndex) { _, val in
localIdx = Double(val) }
            .tint(.white).frame(width: 140)
            Text("\(boardVM.currentMoveIndex)/\
(boardVM.totalMoves)").font(.system(size: 9,
design: .monospaced)).foregroundColor(.white.opacity(0.8)).frame(width
: 50)
        }.padding(.horizontal, 12).padding(.vertical,
6).background(Color.black.opacity(0.65)).cornerRadius(18).foregroundCo
lor(.white).fixedSize()
    }
}

// FILE: ./Views/KeyboardShortcuts.swift
// MARK: - File: KeyboardShortcuts.swift (v5.100)
import SwiftUI

struct KeyboardShortcutsModifier: ViewModifier {
    let boardVM: BoardViewModel
    func body(content: Content) -> some View {
        content.focusable().onKeyPress(phases: .down) { press in
            let isShift = press.modifiers.contains(.shift)
            switch press.key {
            case .leftArrow: isShift ? boardVM.stepBackwardTen() :
boardVM.stepBackward(); return .handled
            case .rightArrow: isShift ? boardVM.stepForwardTen() :
boardVM.stepForward(); return .handled
            case .upArrow: boardVM.goToStart(); return .handled
            case .downArrow: boardVM.goToEnd(); return .handled
            case .space: boardVM.toggleAutoPlay(); return .handled
            default: return .ignored
            }
        }
    }
}
extension View { func keyboardShortcuts(boardVM: BoardViewModel) ->
some View { modifier(KeyboardShortcutsModifier(boardVM: boardVM)) } }

// FILE: ./Views/ActiveGamePanel.swift
// MARK: - File: ActiveGamePanel.swift (v3.502)
import SwiftUI

struct ActiveGamePanel: View {
    @EnvironmentObject var app: AppModel

    var body: some View {
        VStack(spacing: 12) {
            headerSection
            Divider().background(Color.white.opacity(0.1))

            if let username = app.ogsClient.undoRequestedUsername {
                VStack(spacing: 8) {
                    Text("Undo request from \
(username)").font(.caption).bold()
                    HStack {
                        Button("Reject") {
                            if let id = app.ogsClient.activeGameID
{ app.ogsClient.sendUndoReject(gameID: id) }
                        }.tint(.red)
                        Button("Accept") {
                            if let id = app.ogsClient.activeGameID
{ app.ogsClient.sendUndoAccept(gameID: id) }
                        }.tint(.green)
                    }.buttonStyle(.borderedProminent).controlSize(.sma
ll)
                }
                .padding().background(Color.orange.opacity(0.2)).corne
rRadius(8)
            }

            playerInfoSection
            Spacer()
            controlsSection
        }
        .padding()
    }

    private var headerSection: some View {
        HStack {
            Text("Game #\(app.ogsClient.activeGameID?.description ??
"?")").font(.headline)
            Spacer()
            Button("Resign") {
                if let id = app.ogsClient.activeGameID
{ app.ogsClient.resignGame(gameID: id) }
            }.foregroundColor(.red)
        }
    }
    private var playerInfoSection: some View {
        HStack {
            VStack(alignment: .leading) {
                Text(app.ogsClient.blackPlayerName ?? "Black").bold()
                Text("\(Int(app.ogsClient.blackTimeRemaining ??
0))s").font(.title3).monospacedDigit()
            }
            Spacer()
            VStack(alignment: .trailing) {
                Text(app.ogsClient.whitePlayerName ?? "White").bold()
                Text("\(Int(app.ogsClient.whiteTimeRemaining ??
0))s").font(.title3).monospacedDigit()
            }
        }
    }

    private var controlsSection: some View {
        HStack {
            Button("Undo") {
                if let id = app.ogsClient.activeGameID {
                    app.ogsClient.sendUndoRequest(gameID: id,
moveNumber: app.player.serverMoveNumber)
                }
            }
            Spacer()
            Button("Pass") {
                if let id = app.ogsClient.activeGameID {
                    // FIX: Added moveNumber argument to match OGS
protocol
                    app.ogsClient.sendPass(gameID: id, moveNumber:
app.player.serverMoveNumber)
                }
            }
        }.buttonStyle(.bordered)
    }
}

// FILE: ./Views/SettingsPanel.swift
//
//  SettingsPanel.swift
//  SGFPlayerClean
//
//  Created: 2025-12-01
//  Purpose: Slide-out settings drawer with visual preferences and
library management
//

import SwiftUI

struct SettingsPanel: View {
```

```swift
@ObservedObject var app: AppModel
@ObservedObject var settings = AppSettings.shared
@Binding var isPresented: Bool

@State private var isMarkersExpanded = true

// Logarithmic binding for Time Delay
private var delayBinding: Binding<Double> {
    Binding<Double>(
        get: {
            let y = max(0.1, settings.moveInterval)
            return log10(y / 0.1) / log10(100.0)
        },
        set: { sliderValue in
            let y = 0.1 * pow(100.0, sliderValue)
            settings.moveInterval = y
        }
    )
}

var body: some View {
    ZStack(alignment: .leading) {
        // 1. Dimmed Background
        Color.black.opacity(0.2)
            .ignoresSafeArea()
            .onTapGesture { close() }

        // 2. The Sidebar Panel
        VStack(alignment: .leading, spacing: 0) {
            headerView

            ScrollView {
                VStack(alignment: .leading, spacing: 24) {
                    playbackSection
                    Divider().background(Color.white.opacity(0.3))
                    markersSection
                    Divider().background(Color.white.opacity(0.3))
                    librarySection
                }
                .padding()
            }

            Spacer()
            footerView
        }
        .frame(width: 350)
        .frostedGlassStyle()
        .padding(.leading, 10)
        .padding(.vertical, 10)
        .transition(.move(edge: .leading))
        }
        .zIndex(100)
    }

    // MARK: - Components

    private var headerView: some View {
        HStack {
            Text("Settings")
                .font(.title2.bold())
                .foregroundColor(.white)
            Spacer()
            Button(action: close) {
                Image(systemName: "xmark.circle.fill")
                    .font(.title2)
                    .foregroundColor(.white.opacity(0.6))
            }
            .buttonStyle(.plain)
        }
        .padding()
        .background(Color.black.opacity(0.1))
    }

    private var playbackSection: some View {
        VStack(alignment: .leading, spacing: 12) {
            Label("Playback", systemImage: "play.circle.fill")
                .font(.headline).foregroundColor(.white)

            // Delay Slider
            VStack(alignment: .leading, spacing: 4) {
                HStack {
                    Text("Move Delay")
                    Spacer()
                    Text(String(format: "%.1fs",
settings.moveInterval))
                        .monospacedDigit().foregroundColor(.white)
                }
                .font(.caption).foregroundColor(.white.opacity(0.8))
                Slider(value: delayBinding, in: 0.0...1.0).tint(.cyan)
            }

            // Jitter Slider
            VStack(alignment: .leading, spacing: 4) {
                HStack {
                    Text("Stone Jitter")
                    Spacer()
                    Text(String(format: "%.2f",
settings.jitterMultiplier))
                        .monospacedDigit().foregroundColor(.white)
                }
                .font(.caption).foregroundColor(.white.opacity(0.8))
                Slider(value: $settings.jitterMultiplier, in:
0.0...2.0, step: 0.05).tint(.cyan)
            }

            Group {
                Toggle("Shuffle Games", isOn:
$settings.shuffleGameOrder)
                Toggle("Start on Launch", isOn:
$settings.startGameOnLaunch)
            }
            .toggleStyle(SwitchToggleStyle(tint: .cyan))
            .font(.caption).foregroundColor(.white)
        }
    }

    private var markersSection: some View {
        DisclosureGroup("Last move markers", isExpanded:
$isMarkersExpanded) {
            VStack(alignment: .leading, spacing: 12) {

                // --- SLIDERS HIDDEN (Hardcoded preferences) ---
                /*
                VStack(alignment: .leading, spacing: 4) {
                    HStack {
                        Text("Panel Opacity")
                        Spacer()
                        Text(String(format: "%.0f%%",
settings.panelOpacity * 100))
                            .monospacedDigit().foregroundColor(.white)
                    }
                    .font(.caption).foregroundColor(.white.opacity(0.8
))
                    Slider(value: $settings.panelOpacity, in:
0.0...1.0).tint(.orange)
                }

                VStack(alignment: .leading, spacing: 4) {
                    HStack {
                        Text("Glass Blur")
                        Spacer()
                        Text(String(format: "%.1f",
settings.panelDiffusiveness))
                            .monospacedDigit().foregroundColor(.white)
                    }
                    .font(.caption).foregroundColor(.white.opacity(0.8
))
                    Slider(value: $settings.panelDiffusiveness, in:
0.0...1.0).tint(.purple)
                }
                .padding(.bottom, 8)
                */
                // ---------------------------------------------

                Group {
                    Toggle("Move Numbers", isOn:
$settings.showMoveNumbers)
                    Toggle("Dot", isOn: $settings.showLastMoveDot)
                    Toggle("Circle", isOn:
$settings.showLastMoveCircle)
                    Toggle("Board Glow", isOn:
$settings.showBoardGlow)
                    Toggle("Enhanced Effects", isOn:
$settings.showEnhancedGlow)
                    Toggle("Drop Stone Animation", isOn:
$settings.showDropInAnimation)
                        .disabled(app.viewMode == .view2D)
                        .foregroundColor(app.viewMode
== .view2D ? .white.opacity(0.4) : .white)
                }
            }
            .toggleStyle(SwitchToggleStyle(tint: .cyan))
            .font(.caption)
            .foregroundColor(.white)
            .padding(.leading, 10)
            .padding(.top, 5)
        }
        .font(.headline).foregroundColor(.white).accentColor(.white)
    }

    private var librarySection: some View {
        VStack(alignment: .leading, spacing: 10) {
            Label("Library", systemImage: "books.vertical.fill")
                .font(.headline).foregroundColor(.white)

            Button(action: { app.promptForFolder() }) {
                HStack {
                    Image(systemName: "folder.badge.plus")
                    Text("Choose Folder")
                }
                .frame(maxWidth: .infinity).padding(8)
                .background(Color.white.opacity(0.15)).cornerRadius(6)
            }
            .buttonStyle(.plain)

            if !app.games.isEmpty {
                ScrollView {
                    LazyVStack(alignment: .leading, spacing: 0) {
                        ForEach(app.games) { wrapper in
                            Button(action:
```

```
{ app.selectGame(wrapper) }) {
                                GameListRow(wrapper: wrapper,
isSelected: app.selection?.id == wrapper.id)
                            }
                            .buttonStyle(.plain)

Divider().background(Color.white.opacity(0.1))
                        }
                    }
                }
                .frame(height: 250)
                .background(Color.black.opacity(0.2)).cornerRadius(6)
            } else {
                Text("No games
loaded").font(.caption).foregroundColor(.gray)
            }
        }
    }

    private var footerView: some View {
        HStack {
            Text("SGFPlayer Clean v1.0.47")
                .font(.caption2).foregroundColor(.white.opacity(0.5))
            Spacer()
        }
        .padding()
        .background(Color.black.opacity(0.2))
    }

    private func close() {
        withAnimation(.easeInOut(duration: 0.45)) { isPresented =
false }
    }
}

struct GameListRow: View {
    let wrapper: SGFGameWrapper
    let isSelected: Bool

    var body: some View {
        VStack(alignment: .leading, spacing: 2) {
            HStack {
                if isSelected {
                    Image(systemName: "play.fill")
                        .font(.caption2)
                        .foregroundColor(.cyan)
                }

                Text(wrapper.game.info.playerBlack ?? "?")
                    .fontWeight(.bold) +
                Text(" vs ") +
                Text(wrapper.game.info.playerWhite ?? "?")
                    .fontWeight(.bold)

                Spacer()
            }
            .font(.caption)
            .foregroundColor(isSelected ? .cyan : .white)

            HStack {
                Text(wrapper.game.info.date ?? "Unknown
Date").font(.caption2).foregroundColor(.gray)
                Spacer()
                Text(wrapper.game.info.result ??
"").font(.caption).bold().foregroundColor(.yellow)
            }
        }
        .padding(.vertical, 6)
        .padding(.horizontal, 8)
        .background(isSelected ? Color.white.opacity(0.15) :
Color.clear)
    }
}

// FILE: ./Views/Logger.swift
//
//  Logger.swift
//  SGFPlayerClean
//
//  v3.155: Console Mode.
//  - Redirects all logs to Xcode Console via NSLog.
//  - Removes UI storage to prevent memory overhead/scrolling issues.
//

import Foundation

class Logger: ObservableObject {
    static let shared = Logger()

    // Kept for compatibility, but empty so UI doesn't render it
    @Published var logs: [LogEntry] = []

    struct LogEntry: Identifiable {
        let id = UUID()
        let time = Date()
        let text: String
        let type: LogType
    }

    enum LogType {
        case info, success, error, network
    }

    func log(_ text: String, type: LogType = .info) {
        // Use NSLog for guaranteed visibility in Xcode Console
        let prefix: String
        switch type {
        case .info: prefix = "ℹ️ [INFO]"
        case .success: prefix = "✅ [SUCCESS]"
        case .error: prefix = "🚨 [ERROR]"
        case .network: prefix = "🌍 [NET]"
        }

        NSLog("\(prefix) \(text)")
    }

    func clear() {
        // No-op
    }
}

// FILE: ./Views/OGSBrowserView.swift
// MARK: - File: OGSBrowserView.swift (v4.202)
import SwiftUI

struct OGSBrowserView: View {
    @EnvironmentObject var app: AppModel
    @Binding var isPresentingCreate: Bool
    var onJoin: (Int) -> Void

    @AppStorage("ogs_filter_speed") private var selectedSpeed:
GameSpeedFilter = .all
    @AppStorage("ogs_filter_ranked") private var showRankedOnly: Bool
= false
    @AppStorage("ogs_filter_sizes") private var sizeFiltersRaw: String
= "19,13,9,Other"

    private var sizeFilters: Set<BoardSizeCategory> {
        Set(sizeFiltersRaw.split(separator: ",").compactMap
{ BoardSizeCategory(rawValue: String($0)) })
    }

    var body: some View {
        VStack(spacing: 0) {
            headerSection
            Divider().background(Color.white.opacity(0.1))
            if filteredChallenges.isEmpty {
                emptyState
            } else {
                challengeList
            }
        }
        .onAppear {
            if !app.ogsClient.isSubscribedToSeekgraph
{ app.ogsClient.subscribeToSeekgraph() }
        }
    }

    private var headerSection: some View {
        VStack(spacing: 12) {
            HStack {
                Text("Lobby").font(.headline).foregroundColor(.white)
                Button(action: { isPresentingCreate = true }) {
                    Image(systemName: "plus").font(.system(size: 14,
weight: .bold)).frame(width: 24, height: 24)
                }.buttonStyle(.bordered).tint(.blue).disabled(!
app.ogsClient.isConnected)
                Spacer()
                if app.ogsClient.isConnected { Label("Connected",
systemImage: "circle.fill").font(.caption).foregroundColor(.green) }
                else { Button("Retry")
{ app.ogsClient.connect() }.font(.caption).buttonStyle(.borderedPromin
ent).tint(.orange) }
            }
            HStack(spacing: 12) {

Text("Size:").font(.caption).foregroundColor(.white.opacity(0.7))
                ForEach(BoardSizeCategory.allCases) { size in
                    Toggle(size.rawValue, isOn: sizeBinding(for:
size)).toggleStyle(.checkbox).font(.caption)
                }
                Spacer()
            }
            HStack {
                Picker("", selection: $selectedSpeed) {
                    ForEach(GameSpeedFilter.allCases) { speed in
Text(speed.rawValue).tag(speed) }
                }.labelsHidden().frame(width: 120).controlSize(.small)
                Toggle("Rated Only", isOn:
$showRankedOnly).toggleStyle(.checkbox).font(.caption)
                Spacer()
            }
        }.padding().background(Color.black.opacity(0.1))
    }

    private var emptyState: some View {
        VStack { Spacer(); Text(app.ogsClient.isConnected ? "No
challenges found." :
"Connecting...").foregroundColor(.white.opacity(0.5)); Spacer() }
    }
    private var challengeList: some View {
        List {
            ForEach(filteredChallenges) { challenge in
                ChallengeRow(challenge: challenge, isMine:
isChallengeMine(challenge)) {
                    if isChallengeMine(challenge)
{ app.ogsClient.cancelChallenge(challengeID: challenge.id) }
                    else { onJoin(challenge.id) }
                }
            }
        }
        .listStyle(.inset).scrollContentBackground(.hidden)
    }

    // FIX: Broken down into sub-expressions to prevent compiler
timeout
    private var filteredChallenges: [OGSChallenge] {
        let all = app.ogsClient.availableGames
        let filtered = all.filter { game in
            if showRankedOnly && !(game.game?.ranked ?? false)
{ return false }

            let w = game.game?.width ?? 19
            let h = game.game?.height ?? 19
            let cat: BoardSizeCategory = (w == 19 && h ==
19) ? .size19 : (w == 13 && h == 13) ? .size13 : (w == 9 && h ==
9) ? .size9 : .other
            if !sizeFilters.contains(cat) { return false }

            if selectedSpeed != .all {
                let speed = game.speedCategory
                if selectedSpeed == .live && speed != "live" { return
false }
                if selectedSpeed == .blitz && speed != "blitz"
{ return false }
                if selectedSpeed == .correspondence && speed !=
"correspondence" { return false }
            }
            return true
        }
        return filtered.sorted { $0.id > $1.id }
    }

    private func isChallengeMine(_ challenge: OGSChallenge) -> Bool {
        guard let pid = app.ogsClient.playerID else { return false }
        return challenge.challenger?.id == pid
    }

    private func sizeBinding(for size: BoardSizeCategory) ->
```

```
Binding<Bool> {
        Binding(
            get: { sizeFilters.contains(size) },
            set: { isActive in
                var current = sizeFilters
                if isActive { current.insert(size) } else
{ current.remove(size) }
                sizeFiltersRaw = current.map
{ $0.rawValue }.sorted().joined(separator: ",")
            }
        )
    }
}

struct ChallengeRow: View {
    let challenge: OGSChallenge; let isMine: Bool; let onAction: () ->
Void
    var body: some View {
        HStack(alignment: .center, spacing: 14) {
            Button(action: onAction) {
                Text(isMine ? "CANCEL" : "ACCEPT").font(.system(size:
10, weight: .bold)).padding(.horizontal,
2).foregroundColor(isMine ? .black : .white)
            }.buttonStyle(.borderedProminent).tint(isMine ? .yellow :
.green).controlSize(.small)

            VStack(alignment: .leading, spacing: 3) {
                HStack(spacing: 6) {
                    Text(challenge.challenger?.displayRank ??
"?").font(.system(size: 13, weight: .bold)).foregroundColor(rankColor)
                    Text(challenge.challenger?.username ??
"Unknown").font(.system(size: 13, weight: .medium)).lineLimit(1)
                }
                Text(challenge.timeControlDisplay).font(.system(size:
11)).foregroundColor(.white.opacity(0.6))
            }.frame(minWidth: 100, alignment: .leading)

            Spacer()

            VStack(alignment: .leading, spacing: 3) {
                Text(challenge.boardSize).font(.system(size: 13,
weight: .bold))
                Text(challenge.game?.rules?.capitalized ??
"Japanese").font(.system(size:
11)).foregroundColor(.white.opacity(0.6))
            }.frame(width: 60, alignment: .leading)

            if challenge.game?.ranked ?? false {
                Text("RATED").font(.system(size: 10,
weight: .heavy)).foregroundColor(.white.opacity(0.8)).padding(.horizon
```

```
tal, 6).padding(.vertical, 2).overlay(RoundedRectangle(cornerRadius:
4).stroke(Color.white.opacity(0.3), lineWidth: 1))
            }
        }.padding(.vertical, 6).listRowBackground(Color.clear)
    }

    var rankColor: Color {
        guard let rank = challenge.challenger?.ranking else
{ return .gray }
        if rank >= 30 { return .cyan }
        if rank >= 20 { return .green }
        return .orange
    }
}

// FILE: ./Views/SharedOverlays.swift
//
//  SharedOverlays.swift
//  SGFPlayerClean
//
//  Created: 2025-11-28
//  Purpose: Floating UI controls (Settings, View Mode, Full Screen)
shared by 2D/3D
//

import SwiftUI

struct SharedOverlays: View {
    @Binding var showSettings: Bool
    @Binding var buttonsVisible: Bool
    @ObservedObject var app: AppModel

    var body: some View {
        ZStack {
            // 1. Settings Overlay (Slide-Out)
            if showSettings {
                SettingsPanel(app: app, isPresented: $showSettings)
                    .zIndex(20)
            }

            // 2. Floating Buttons (Top LEFT Alignment)
            VStack {
                HStack(spacing: 12) {
                    if buttonsVisible {
                        // A. Settings Button
                        Button(action: {
                            // Slower animation for better slide feel
(0.35s)
                            withAnimation(.easeInOut(duration: 0.35))
{
```

```
                                showSettings.toggle()
                            }
                        }) {
                            Image(systemName: "gearshape.fill")
                                .font(.system(size: 16))
                                .foregroundColor(.white)
                                .frame(width: 32, height: 32)
                                .background(Circle().fill(Color.black.
opacity(0.6)))
                                .overlay(Circle().stroke(Color.white.o
pacity(0.3), lineWidth: 1))
                        }
                        .buttonStyle(.plain)
                        .help("Settings & Load Game")

                        // B. View Mode Toggle
                        Button(action: {
                            withAnimation {
                                app.viewMode = app.viewMode
== .view2D ? .view3D : .view2D
                            }
                        }) {
                            Text(app.viewMode == .view2D ? "3D" :
"2D")
                                .font(.system(size: 12,
weight: .bold))
                                .foregroundColor(.white)
                                .frame(width: 32, height: 32)
                                .background(Circle().fill(Color.black.
opacity(0.6)))
                                .overlay(Circle().stroke(Color.white.o
pacity(0.3), lineWidth: 1))
                        }
                        .buttonStyle(.plain)
                        .help("Switch View Mode")

                        // C. Full Screen Toggle
                        Button(action: toggleFullScreen) {
                            Image(systemName:
"arrow.up.left.and.arrow.down.right")
                                .font(.system(size: 14))
                                .foregroundColor(.white)
                                .frame(width: 32, height: 32)
                                .background(Circle().fill(Color.black.
opacity(0.6)))
                                .overlay(Circle().stroke(Color.white.o
pacity(0.3), lineWidth: 1))
                        }
                        .buttonStyle(.plain)
                        .help("Toggle Full Screen")
```

```
                    }

                    Spacer()
                }
                .padding(20)
                .transition(.opacity)

                Spacer()
            }
        }
    }

    private func toggleFullScreen() {
        if let window = NSApp.keyWindow {
            window.toggleFullScreen(nil)
        }
    }
}

// FILE: ./Views/SupportingViews.swift
// MARK: - File: SupportingViews.swift (v8.100)
import SwiftUI

struct TatamiBackground: View {
    let boardHeight: CGFloat
    var body: some View {
        let scale = max(0.1, boardHeight / 800.0)
        #if os(macOS)
        let img = NSImage(named: "tatami.jpg") ?? NSImage(named:
"tatami")
        #else
        let img = UIImage(named: "tatami.jpg") ?? UIImage(named:
"tatami")
        #endif
        return ZStack {
            Color(red: 0.88, green: 0.84, blue: 0.68)
            if let actual = img {
                #if os(macOS)
                Rectangle().fill(ImagePaint(image: Image(nsImage:
actual), scale: scale))
                #else
                Rectangle().fill(ImagePaint(image: Image(uiImage:
actual), scale: scale))
                #endif
            }
        }.ignoresSafeArea()
    }
}

struct SafeImage: View {
    let name: String; let resizingMode: Image.ResizingMode
    var body: some View {
        #if os(macOS)
        if let img = NSImage(named: name) ?? NSImage(named: (name as 
NSString).deletingPathExtension) {
            Image(nsImage: img).resizable(resizingMode: resizingMode)
        } else { Color.white.opacity(0.05) }
        #else
        if let img = UIImage(named: name) {
            Image(uiImage: img).resizable(resizingMode: resizingMode)
        } else { Color.clear }
        #endif
    }
}

struct BoardGridShape: Shape {
    let boardSize: Int
    func path(in rect: CGRect) -> Path {
        var path = Path()
        let stepX = rect.width / CGFloat(boardSize - 1), stepY = 
rect.height / CGFloat(boardSize - 1)
        for i in 0..<boardSize {
            let x = CGFloat(i) * stepX; path.move(to: CGPoint(x: x, y: 
0)); path.addLine(to: CGPoint(x: x, y: rect.height))
            let y = CGFloat(i) * stepY; path.move(to: CGPoint(x: 0, y: 
y)); path.addLine(to: CGPoint(x: rect.width, y: y))
        }
        return path
    }
}

struct FrostedGlass: ViewModifier {
    @ObservedObject var settings = AppSettings.shared
    func body(content: Content) -> some View {
        content
            .background(.ultraThinMaterial.opacity(settings.panelDiffu
siveness))
            .background(Color(red: 0.05, green: 0.1, blue: 
0.1).opacity(settings.panelOpacity))
            .cornerRadius(12).overlay(RoundedRectangle(cornerRadius: 
12).stroke(Color.white.opacity(0.15), lineWidth: 1))
    }
}
extension View { func frostedGlassStyle() -> some View 
{ modifier(FrostedGlass()) } }

struct StatRow: View {
    let label: String; let icon: String; let value: String; var 
isMonospaced: Bool = false
    var body: some View {
        HStack {
            Label(label, systemImage: icon).font(.caption)
            Spacer()
            Text(value).font(isMonospaced ? .system(.caption,
design: .monospaced) : .caption)
        }.foregroundColor(.white.opacity(0.9))
    }
}

// FILE: ./Views/SimpleBowlView.swift
// MARK: - File: SimpleBowlView.swift (v6.400)
import SwiftUI

struct SimpleLidView: View {
    let stoneColor: Stone; let stoneCount: Int; let stoneSize:
CGFloat; let lidNumber: Int; let lidSize: CGFloat
    var body: some View {
        ZStack {
            SafeImage(name: lidNumber == 1 ? "go_lid_1.png" :
"go_lid_2.png", resizingMode: .stretch)
                .frame(width: lidSize, height:
lidSize).shadow(color: .black.opacity(0.4), radius: 5, x: 2, y: 3)
            if stoneCount > 0 { LidStonesPile(color: stoneColor,
count: min(stoneCount, 35), stoneSize: stoneSize, lidSize: lidSize) }
        }.frame(width: lidSize, height: lidSize)
    }
}
struct LidStonesPile: View {
    let color: Stone; let count: Int; let stoneSize: CGFloat; let
lidSize: CGFloat
    var body: some View {
        let adjS = color == .black ? stoneSize * 1.015 : stoneSize *
0.995
        ZStack {
            ForEach(0..<count, id: \.self) { i in
                let off = getOff(i: i, r: lidSize * 0.3)
                StoneView2D(color: color, position: BoardPosition(0,
0), seedOverride: i * 7)
                    .frame(width: adjS, height: adjS).position(x:
(lidSize / 2) + off.x, y: (lidSize / 2) + off.y)
            }
        }
    }
    private func getOff(i: Int, r: CGFloat) -> CGPoint {
        let a = Double(i) * 2.39996, d = r * sqrt(Double(i) /
Double(count))
        return CGPoint(x: CGFloat(cos(a) * d), y: CGFloat(sin(a) * d))
    }
}
```

```
// FILE: ./Views/RightPanelView.swift
// MARK: - File: RightPanelView.swift (v8.102)
import SwiftUI

struct RightPanelView: View {
    @EnvironmentObject var app: AppModel
    @State private var selectedTab: RightPanelTab = .local
    enum RightPanelTab: String { case local = "Local", online =
"Online" }

    var body: some View {
        VStack(spacing: 0) {
            Picker("Mode", selection: $selectedTab) {
                Text("Local").tag(RightPanelTab.local)
                Text("Online").tag(RightPanelTab.online)
            }.pickerStyle(.segmented).padding(10)

            Divider().background(Color.white.opacity(0.1))

            ZStack {
                if selectedTab == .local {
                    VStack(spacing: 0) {
                        if let game = app.selection, let bvm =
app.boardVM {
                            LocalGameMetadataView(game: game, boardVM:
bvm)

Divider().background(Color.white.opacity(0.1))
                        }
                        LocalPlaylistView()
                    }
                } else {
                    onlineTabContent
                }
            }
        }.frostedGlassStyle()
    }

    @ViewBuilder
    private var onlineTabContent: some View {
        VStack {
            if app.ogsClient.activeGameID != nil &&
app.ogsClient.isConnected {
                ActiveGamePanel().transition(.opacity)
            } else {
                OGSBrowserView(isPresentingCreate:
$app.isCreatingChallenge) { id in
                    app.joinOnlineGame(id: id)
                }.transition(.opacity)
            }
        }
    }
}

struct LocalGameMetadataView: View {
    let game: SGFGameWrapper; @ObservedObject var boardVM:
BoardViewModel
    var body: some View {
        VStack(spacing: 12) {
            HStack(alignment: .top) {
                VStack(alignment: .leading, spacing: 4) {
                    Label(game.game.info.playerBlack ?? "Black",
systemImage: "circle.fill").font(.headline)
                    Text("\(boardVM.blackCapturedCount)
prisoners").font(.caption).foregroundColor(.white.opacity(0.7))
                }
                Spacer()
                VStack(alignment: .trailing, spacing: 4) {
                    Label(game.game.info.playerWhite ?? "White",
systemImage: "circle").font(.headline)
                    Text("\(boardVM.whiteCapturedCount)
prisoners").font(.caption).foregroundColor(.white.opacity(0.7))
                }
            }
        }.padding().background(Color.black.opacity(0.1))
    }
}

struct LocalPlaylistView: View {
    @EnvironmentObject var app: AppModel
    var body: some View {
        VStack(spacing: 0) {
            if app.games.isEmpty {
                Spacer(); Text("No SGF files
loaded").foregroundColor(.secondary); Spacer()
            } else {
                ScrollView {
                    LazyVStack(alignment: .leading, spacing: 0) {
                        ForEach(app.games) { game in
                            Button(action: { app.selectGame(game) }) {
                                LocalGameRow(game: game, isSelected:
app.selection?.id == game.id)
                            }.buttonStyle(.plain)

Divider().background(Color.white.opacity(0.05))
                        }
                    }
                }
            }
        }
    }
}

struct LocalGameRow: View {
    let game: SGFGameWrapper; let isSelected: Bool
    var body: some View {
        VStack(alignment: .leading, spacing: 2) {
            HStack {
                if isSelected { Image(systemName:
"play.fill").font(.caption2).foregroundColor(.cyan) }
                Text(game.game.info.playerBlack ??
"?").fontWeight(.bold) + Text(" vs ") +
Text(game.game.info.playerWhite ?? "?").fontWeight(.bold)
                Spacer()
            }.font(.caption).foregroundColor(isSelected ? .cyan : .whi
te)
            HStack {
                Text(game.game.info.date ?? "Unknown
Date").font(.caption2).foregroundColor(.gray)
                Spacer()
                if let result = game.game.info.result
{ Text(result).font(.caption).bold().foregroundColor(.yellow) }
            }
        }.padding(.vertical, 6).padding(.horizontal,
8).contentShape(Rectangle()).background(isSelected ?
Color.white.opacity(0.15) : Color.clear)
    }
}

// FILE: ./Views/ContentView3D.swift
// MARK: - File: ContentView3D.swift (v4.974)
import SwiftUI
import SceneKit
import Combine

struct ContentView3D: View {
    @EnvironmentObject var app: AppModel
    @StateObject private var sceneManager = SceneManager3D()
    @FocusState private var isBoardFocused: Bool

    // Initialize @State from persisted AppSettings
    @State private var rotX: Float
    @State private var rotY: Float
    @State private var distance: CGFloat
    @State private var panX: CGFloat
    @State private var panY: CGFloat

    init() {
        let settings = AppSettings.shared
        _rotX = State(initialValue: Float(settings.camera3DRotationX))
        _rotY = State(initialValue: Float(settings.camera3DRotationY))
        _distance = State(initialValue:
CGFloat(settings.camera3DDistance))
        _panX = State(initialValue: CGFloat(settings.camera3DPanX))
        _panY = State(initialValue: CGFloat(settings.camera3DPanY))
    }

    var body: some View {
        ZStack {
            if let bVM = app.boardVM {
                // 3D Scene Layer
                InteractiveSceneView(scene: sceneManager.scene,
cameraNode: sceneManager.cameraNode, sceneManager: sceneManager,
boardVM: bVM, rotationX: $rotX, rotationY: $rotY)
                    .edgesIgnoringSafeArea(.all)

                // Gesture Overlay with Capture Logic
                CameraControlHandler(
                    rotationX: $rotX,
                    rotationY: $rotY,
                    distance: $distance,
                    panX: $panX,
                    panY: $panY,
                    sceneManager: sceneManager,
                    onInteractionEnded: { saveViewportDefault() }
                )

                // UI Overlay Layer
                VStack(spacing: 0) {
                    HStack(spacing: 0)
{ Color.clear.allowsHitTesting(false); RightPanelView().frame(width:
320) }
                    Spacer(); PlaybackControlsView(boardVM:
bVM).padding(.bottom, 25)
                }
            }
        }
        .focused($isBoardFocused)
        .keyboardShortcuts(boardVM: app.boardVM!)
        .onAppear {
            isBoardFocused = true
            updateScene()
        }
        .onReceive(app.boardVM?.onRequestUpdate3D ??
PassthroughSubject<Void, Never>()) { _ in
            updateScene()
        }
    }

    private func updateScene() {
        guard let bVM = app.boardVM else { return }
        sceneManager.updateStones(from: bVM.stonesToRender, lastMove:
bVM.lastMovePosition, moveIndex: bVM.currentMoveIndex, settings:
AppSettings.shared)
        sceneManager.updateCapturedStones(black:
bVM.blackCapturedCount, white: bVM.whiteCapturedCount)
        sceneManager.updateCameraPosition(distance: distance,
rotationX: rotX, rotationY: rotY, panX: panX, panY: panY)
    }

    private func saveViewportDefault() {
        // Sync the current interactive state back to the persisted
Defaults
        let settings = AppSettings.shared
        settings.camera3DRotationX = Double(rotX)
        settings.camera3DRotationY = Double(rotY)
        settings.camera3DDistance = Double(distance)
        settings.camera3DPanX = Double(panX)
        settings.camera3DPanY = Double(panY)
    }
}

struct InteractiveSceneView: NSViewRepresentable {
    let scene: SCNScene; let cameraNode: SCNNode; let sceneManager:
SceneManager3D; let boardVM: BoardViewModel
    @Binding var rotationX: Float; @Binding var rotationY: Float
    func makeNSView(context: Context) -> ClickableSCNView {
        let v = ClickableSCNView(); v.scene = scene; v.pointOfView =
cameraNode; v.backgroundColor = .black; v.antialiasingMode
= .multisampling4X
        v.onClick = { p in if let (c, r) = sceneManager.hitTest(point:
p, in: v) { boardVM.placeStone(at: BoardPosition(r, c)) } }
        return v
    }
    func updateNSView(_ v: ClickableSCNView, context: Context)
{ v.scene = scene; v.pointOfView = cameraNode }
}

class ClickableSCNView: SCNView {
    var onClick: ((CGPoint) -> Void)?
    private var downEvent: NSEvent?
    override func updateTrackingAreas() {
        super.updateTrackingAreas(); for t in self.trackingAreas
{ self.removeTrackingArea(t) }
        self.addTrackingArea(NSTrackingArea(rect: self.bounds,
options:
[.mouseMoved, .activeInKeyWindow, .inVisibleRect, .activeAlways],
owner: self, userInfo: nil))
    }
    override func mouseDown(with e: NSEvent) { self.downEvent = e;
```

```
super.mouseDown(with: e) }
    override func mouseUp(with e: NSEvent) { if let d = downEvent { if
hypot(d.locationInWindow.x - e.locationInWindow.x,
d.locationInWindow.y - e.locationInWindow.y) < 10 { onClick?
(self.convert(e.locationInWindow, from: nil)) } }; self.downEvent =
nil; super.mouseUp(with: e) }
}

// FILE: ./Views/DebugOverlay.swift
// MARK: - File: DebugOverlay.swift (v6.401)
import SwiftUI

struct DebugOverlay: View {
    @ObservedObject var app: AppModel
    @ObservedObject var client: OGSClient
    @ObservedObject var board: BoardViewModel

    var body: some View {
        VStack(alignment: .leading, spacing: 8) {
            Text("--- DIAGNOSTICS
---").font(.caption2).bold().foregroundColor(.gray)
            VStack(alignment: .leading, spacing: 2) {
                Text("Game ID: \(client.activeGameID?.description ??
"NIL")").foregroundColor(client.activeGameID != nil ? .green : .red)
                Text("Socket: \(client.isConnected ? "CONNECTED" :
"OFFLINE")").foregroundColor(client.isConnected ? .green : .orange)
                Text("Auth: \(client.isSocketAuthenticated ?
"VERIFIED" :
"PENDING")").foregroundColor(client.isSocketAuthenticated ? .green : .
red)
                Text("JWT: \(client.userJWT != nil ? "OK" :
"MISSING")").foregroundColor(client.userJWT != nil ? .green : .red)
            }
            Divider().background(Color.white.opacity(0.3))
            VStack(alignment: .leading, spacing: 2) {
                Text("Context: \(board.isOnlineContext ? "ONLINE" :
"LOCAL")").foregroundColor(board.isOnlineContext ? .cyan : .yellow)
                // Optimized: Using the Render Cache array instead of
the raw grid dictionary
                Text("Stones: \(board.stonesToRender.count)")
                Text("MoveIdx: \(board.currentMoveIndex)")
                if let last = board.lastMovePosition {
                    Text("Last: \(last.col), \
(last.row)").foregroundColor(.gray)
                }
            }
            if let err = client.lastError {
                Divider().background(Color.white.opacity(0.3))
                Text("ERR: \(err)").foregroundColor(.red).lineLimit(3)
            }
        }
        .font(.system(size: 10, design: .monospaced))
        .padding(8)
        .background(Color.black.opacity(0.85))
        .foregroundColor(.white)
        .cornerRadius(8)
        .frame(maxWidth: 220, alignment: .leading)
        .padding()
    }
}

// FILE: ./Views/BoardView2D.swift
// MARK: - File: BoardView2D.swift (v8.103)
import SwiftUI

struct BoardView2D: View {
    @ObservedObject var boardVM: BoardViewModel
    @ObservedObject var layoutVM: LayoutViewModel
    let size: CGSize

    var body: some View {
        let margin = size.width * 0.065
        let gridW = size.width - (margin * 2)
        let gridH = size.height - (margin * 2)
        let bSize = boardVM.boardSize
        let cellP = gridW / CGFloat(max(1, bSize - 1))
        let rowP = gridH / CGFloat(max(1, bSize - 1))

        ZStack {
            ZStack {
                Color(red: 0.82, green: 0.65, blue: 0.4)
                SafeImage(name: "board_kaya.jpg",
resizingMode: .stretch)
            }
            .frame(width: size.width, height: size.height)
            .cornerRadius(2)
            .shadow(color: .black.opacity(0.4), radius: 8)

            ZStack(alignment: .topLeading) {
                BoardGridShape(boardSize: bSize)
                    .stroke(Color.black.opacity(0.8), lineWidth: 1.0)

                ForEach(starPoints(size: bSize), id: \.self) { pt in
                    Circle().fill(Color.black)
                        .frame(width: size.width * 0.012, height:
size.width * 0.012)
                        .position(x: CGFloat(pt.col) * cellP, y:
CGFloat(pt.row) * rowP)
                }
                ForEach(boardVM.stonesToRender) { rs in
                    let sS = (rs.color == .black ? 1.015 : 0.995) *
cellP
                    StoneView2D(color: rs.color, position: rs.id)
                        .frame(width: sS, height: sS)
                        .position(x: CGFloat(rs.id.col) * cellP +
(rs.offset.x * cellP),
                                  y: CGFloat(rs.id.row) * rowP +
(rs.offset.y * rowP))
                }

                if let ghostPos = boardVM.ghostPosition, let color =
boardVM.ghostColor {
                    StoneView2D(color: color, position: ghostPos)
                        .frame(width: cellP, height: cellP)
                        .opacity(0.4)
                        .position(x: CGFloat(ghostPos.col) * cellP, y:
CGFloat(ghostPos.row) * rowP)
                }

                if let lastPos = boardVM.lastMovePosition {
                    let j = boardVM.getJitterOffset(forPosition:
lastPos)
                    Circle().stroke(Color.white, lineWidth: max(1.5,
size.width * 0.005))
                        .frame(width: cellP * 0.45, height: cellP *
0.45)
                        .position(x: CGFloat(lastPos.col) * cellP +
(j.x * cellP),
                                  y: CGFloat(lastPos.row) * rowP +
(j.y * rowP))
                }
            }
            .frame(width: gridW, height: gridH)

            // Interaction Layer (Top-most)
            Color.clear.contentShape(Rectangle())
                .frame(width: size.width, height: size.height)
                .onContinuousHover { phase in
                    switch phase {
                    case .active(let loc):
                        let c = Int(round((loc.x - margin) / cellP))
                        let r = Int(round((loc.y - margin) / rowP))
                        if c >= 0 && c < bSize && r >= 0 && r < bSize
{
                            boardVM.updateGhostStone(at:
BoardPosition(r, c))
                        } else { boardVM.clearGhostStone() }
                    case .ended: boardVM.clearGhostStone()
                    }
                }
                .onTapGesture { loc in
                    let c = Int(round((loc.x - margin) / cellP))
                    let r = Int(round((loc.y - margin) / rowP))
                    if c >= 0 && c < bSize && r >= 0 && r < bSize {
                        boardVM.placeStone(at: BoardPosition(r, c))
                    }
                }
        }
    }

    private func starPoints(size: Int) -> [BoardPosition] {
        if size == 19 { return [BoardPosition(3,3),
BoardPosition(3,9), BoardPosition(3,15), BoardPosition(9,3),
BoardPosition(9,9), BoardPosition(9,15), BoardPosition(15,3),
BoardPosition(15,9), BoardPosition(15,15)] }
        return []
    }
}

// FILE: ./Views/DebugDashboard.swift
// MARK: - File: DebugDashboard.swift (v3.260)
//
//  A comprehensive debug view for OGS traffic and state inspection.
//  Updated to harmonize with BoardViewModel v3.250.
//

import SwiftUI

struct DebugDashboard: View {
    @ObservedObject var appModel: AppModel
    @State private var showHeartbeats: Bool = false

    var body: some View {
        HStack(spacing: 0) {
            // LEFT PANEL: Traffic Logs
            VStack(spacing: 0) {
                // Header
                HStack {
                    Text("Traffic Inspector")
                        .font(.headline)
                        .foregroundColor(.white)

                    Spacer()

                    Toggle("Beat", isOn: $showHeartbeats)
                        .toggleStyle(SwitchToggleStyle(tint: .blue))
                        .controlSize(.mini)
                        .labelsHidden()
                    Text("❤️").font(.caption2).foregroundColor(.gray)
                    Button(action:
{ appModel.ogsClient.trafficLogs.removeAll() }) {
                        Image(systemName: "trash")
                            .foregroundColor(.white)
                    }
                    .buttonStyle(.plain)
                    .padding(.leading, 8)
                }
                .padding(10)
                .background(Color.black.opacity(0.8))

                // Log List
                List {
                    ForEach(appModel.ogsClient.trafficLogs) { entry in
                        if showHeartbeats || !entry.isHeartbeat {
                            LogRow(entry: entry)
                        }
                    }
                }
                .listStyle(.plain)
                .background(Color.black.opacity(0.9))
            }
            .frame(minWidth: 350)

            Divider().background(Color.gray)

            // RIGHT PANEL: State Inspector
            VStack(alignment: .leading, spacing: 20) {
                Text("State Inspector")
                    .font(.headline)
                    .foregroundColor(.white)
                    .padding(.bottom, 10)

                // Connection & Auth Info
                VStack(alignment: .leading, spacing: 8) {
                    StateRow(label: "Connected", value:
appModel.ogsClient.isConnected ? "YES" : "NO")
                    StateRow(label: "Socket Auth", value:
appModel.ogsClient.isSocketAuthenticated ? "YES" : "NO")
                    StateRow(label: "Game ID", value: "\
(appModel.ogsClient.activeGameID ?? -1)")
                    StateRow(label: "Auth Token", value:
appModel.ogsClient.activeGameAuth == nil ? "Missing" : "OK")
                    StateRow(label: "Player ID", value: "\
(appModel.ogsClient.playerID ?? -1)")
                }

                Divider().background(Color.gray)
                // Sync Status (Engine State)
                VStack(alignment: .leading, spacing: 8) {
                    Text("Sync Status")
                        .font(.subheadline)
                        .foregroundColor(.yellow)

                    if let boardVM = appModel.boardVM {
                        StateRow(label: "Move Index", value: "\
(boardVM.currentMoveIndex)")
                        StateRow(label: "Total Moves", value: "\
(boardVM.totalMoves)")
                        StateRow(label: "Next Turn", value:
appModel.player.turn == .black ? "Black" : "White")
                        StateRow(label: "My Color", value:
appModel.ogsClient.playerColor == .black ? "Black" :
(appModel.ogsClient.playerColor == .white ? "White" : "Spectator"))
                    } else {
                        Text("Board VM Not
Active").foregroundColor(.red)
                    }
                }

                Spacer()

                Button("Close") {
                    appModel.showDebugDashboard = false
                }
                .frame(maxWidth: .infinity)
                .padding()
                .background(Color.white.opacity(0.1))
                .cornerRadius(8)
            }
            .padding()
            .frame(width: 250)
            .background(Color(white: 0.15))
        }
        .frame(minWidth: 600, minHeight: 400)
    }
}

struct LogRow: View {
    let entry: NetworkLogEntry

    var color: Color {
        if entry.direction == "⬆️" { return Color.blue }
        if entry.direction == "⚡" { return Color.yellow }
        if entry.direction == "⚠️" { return Color.red }
        return Color.green
    }
    var body: some View {
        VStack(alignment: .leading, spacing: 4) {
            HStack {
                Text(entry.direction)
                Text(entry.timestamp, style: .time)
                    .font(.system(size: 10, design: .monospaced))
                    .foregroundColor(.gray)
                Spacer()
            }

            Text(entry.content)
                .font(.system(size: 11, design: .monospaced))
                .foregroundColor(color)
                .fixedSize(horizontal: false, vertical: true)
                .textSelection(.enabled)
        }
        .padding(.vertical, 4)
        .listRowBackground(Color.clear)
    }
}

struct StateRow: View {
    let label: String
    let value: String

    var body: some View {
        HStack {
            Text(label)
                .font(.caption)
                .foregroundColor(.gray)
            Spacer()
            Text(value)
                .font(.system(.body, design: .monospaced))
                .foregroundColor(.white)
        }
    }
}

// FILE: ./Views/StoneView2D.swift
// MARK: - File: StoneView2D.swift (v8.100)
import SwiftUI

struct StoneView2D: View {
    let color: Stone; let position: BoardPosition; var seedOverride:
Int? = nil
    init(color: Stone, position: BoardPosition = BoardPosition(0,0),
seedOverride: Int? = nil) {
        self.color = color; self.position = position;
self.seedOverride = seedOverride
    }
    var body: some View {
        ZStack {
            Circle().fill(Color.black.opacity(0.35)).offset(x: 1.2, y:
1.2)
            if color == .black { SafeImage(name: "stone_black.png",
resizingMode: .stretch) }
            else {
                let idx = ((seedOverride ?? (position.row * 31 +
position.col)) % 5) + 1
                SafeImage(name: String(format: "clam_%02d.png", idx),
resizingMode: .stretch)
            }
        }
    }
}

// FILE: ./Views/ContentView.swift
// MARK: - File: ContentView.swift (v7.200)
import SwiftUI

struct ContentView: View {
    @StateObject var appModel = AppModel()
    @State private var showSettings: Bool = false
    @State private var buttonsVisible: Bool = true

    var body: some View {
        ZStack {
            mainInterface.disabled(appModel.isCreatingChallenge)
            SharedOverlays(showSettings: $showSettings,
buttonsVisible: $buttonsVisible, app: appModel)
            VStack {
                HStack {
                    Spacer()
                    Button(action:
{ appModel.showDebugDashboard.toggle() }) {
                        Image(systemName:
"ladybug.fill").foregroundColor(appModel.ogsClient.isConnected ? .gree
n : .red)
                            .padding(8).background(Color.black.opacity
(0.6)).clipShape(Circle())
                    }.buttonStyle(.plain).padding()
                }
                Spacer()
            }
            if appModel.isCreatingChallenge {
                OGSCreateChallengeView(isPresented:
$appModel.isCreatingChallenge).background(Color.black.opacity(0.6)).tr
ansition(.opacity).zIndex(200)
            }
        }
        .environmentObject(appModel).preferredColorScheme(.dark)
        .sheet(isPresented: $appModel.showDebugDashboard)
{ DebugDashboard(appModel: appModel).frame(minWidth: 700, minHeight:
500) }
    }

    @ViewBuilder
    private var mainInterface: some View {
        if appModel.viewMode == .view3D { ContentView3D() }
        else { ContentView2D() }
    }
}

// FILE: ./Views/OGSCreateChallengeView.swift
// MARK: - File: OGSCreateChallengeView.swift (v4.200)
import SwiftUI

struct OGSCreateChallengeView: View {
    @EnvironmentObject var app: AppModel
    @Binding var isPresented: Bool
    @State private var setup = ChallengeSetup.load()
    @State private var isSending = false
    @State private var errorMessage: String?

    var body: some View {
        VStack(spacing: 0) {
            headerSection
            Divider().background(Color.white.opacity(0.2))
            ScrollView {
                VStack(alignment: .leading, spacing: 0) {
                    gameInfoSection
                    Divider().background(Color.white.opacity(0.15))
                    boardSection
                    Divider().background(Color.white.opacity(0.15))
                    timeControlSection
                    Divider().background(Color.white.opacity(0.15))
                    rankRangeSection
                    if let error = errorMessage
{ Text(error).foregroundColor(.red).font(.caption).padding() }
                }
            }.scrollContentBackground(.hidden)
        }
        .frame(minWidth: 400, minHeight: 650)
        .background(Color(white: 0.15))
        .cornerRadius(12)
    }

    private var headerSection: some View {
        HStack {
            Button(action: { isPresented = false })
```

```
{ Text("Cancel").foregroundColor(.white.opacity(0.8)) }.buttonStyle(.p
lain)
            Spacer(); Text("New
Challenge").font(.headline).foregroundColor(.white); Spacer()
            Button(action: submitChallenge) {
                if isSending { ProgressView().controlSize(.small) }
                else
{ Text("Create").fontWeight(.bold).foregroundColor(.white) }
            }.buttonStyle(.borderedProminent).tint(.green).disabled(is
Sending)
        }.padding().background(Color.black.opacity(0.1))
    }

    private var gameInfoSection: some View {
        VStack(alignment: .leading) {
            SectionHeader(text: "Game Info")
            VStack(spacing: 12) {
                HStack { Text("Name"); Spacer(); TextField("Name",
text:
$setup.name).textFieldStyle(.plain).multilineTextAlignment(.trailing).
frame(width: 150) }
                Toggle("Ranked", isOn: $setup.ranked)
                HStack {
                    Text("Color"); Spacer()
                    Picker("", selection: $setup.color) {
                        Text("Auto").tag("automatic")
                        Text("Black").tag("black")
                        Text("White").tag("white")
                    }.labelsHidden().fixedSize()
                }
                HStack {
                    Text("Handicap"); Spacer()
                    Picker("", selection: $setup.handicap) {
                        Text("None").tag(0)
                        ForEach(2...9, id: \.self) { i in Text("\
(i)").tag(i) }
                    }.labelsHidden().fixedSize()
                }
            }.padding()
        }
    }

    private var boardSection: some View {
        VStack(alignment: .leading) {
            SectionHeader(text: "Board")
            VStack(spacing: 12) {
                Picker("", selection: $setup.size) {
                    Text("19x19").tag(19)
                    Text("13x13").tag(13)
                    Text("9x9").tag(9)
                }.pickerStyle(.segmented)
                HStack {
                    Text("Rules"); Spacer()
                    Picker("", selection: $setup.rules) {
                        Text("Japanese").tag("japanese")
                        Text("Chinese").tag("chinese")
                        Text("AGA").tag("aga")
                    }.labelsHidden().fixedSize()
                }
            }.padding()
        }
    }

    private var timeControlSection: some View {
        VStack(alignment: .leading) {
            SectionHeader(text: "Time Control")
            VStack(spacing: 12) {
                Picker("", selection: $setup.timeControl) {
                    Text("Byoyomi").tag("byoyomi")
                    Text("Fischer").tag("fischer")
                    Text("Simple").tag("simple")
                }.pickerStyle(.segmented)

                if setup.timeControl == "byoyomi" {
                    TimeFieldRow(label: "Main Time", value: 
$setup.mainTime)
                    TimeFieldRow(label: "Period Time", value: 
$setup.periodTime)
                    HStack { Text("Periods"); Spacer(); Stepper("\
(setup.periods)", value: $setup.periods, in: 1...10) }
                } else if setup.timeControl == "fischer" {
                    TimeFieldRow(label: "Initial", value: 
$setup.initialTime)
                    TimeFieldRow(label: "Increment", value: 
$setup.increment)
                    TimeFieldRow(label: "Max", value: $setup.maxTime)
                } else {
                    TimeFieldRow(label: "Per Move", value: 
$setup.perMove)
                }
            }.padding()
        }
    }

    private var rankRangeSection: some View {
        VStack(alignment: .leading) {
            SectionHeader(text: "Rank Range")
            VStack(spacing: 12) {
                HStack {
                    Text("Min: \(formatRank(setup.minRank))")
                    Spacer()
                    Stepper("", value: $setup.minRank, in:
0...setup.maxRank)
                }
                HStack {
                    Text("Max: \(formatRank(setup.maxRank))")
                    Spacer()
                    Stepper("", value: $setup.maxRank, in:
setup.minRank...38)
                }
            }.padding()
        }
    }

    private func submitChallenge() {
        setup.save(); isSending = true; errorMessage = nil
        app.ogsClient.createChallenge(setup: setup) { success, _ in
            DispatchQueue.main.async { self.isSending = false; if
success { isPresented = false } else { errorMessage = "Creation
failed" } }
        }
    }
    private func formatRank(_ val: Int) -> String { val < 30 ? "\(30 -
val)k" : "\(val - 29)d" }

    struct SectionHeader: View {
        let text: String
        var body: some View
{ Text(text).font(.caption).fontWeight(.bold).foregroundColor(.white.o
pacity(0.5)).frame(maxWidth: .infinity,
alignment: .leading).padding(.horizontal).padding(.top, 16) }
    }
    struct TimeFieldRow: View {
        let label: String; @Binding var value: Int
        var body: some View { HStack { Text(label); Spacer();
TextField("", value: $value,
format: .number).textFieldStyle(.plain).multilineTextAlignment(.traili
ng).frame(width:
50).padding(4).background(Color.white.opacity(0.1)).cornerRadius(4);
Text("s").foregroundColor(.gray) } }
    }
}

// FILE: ./Services/FileLoadingService.swift
//
//  FileLoadingService.swift
//  SGFPlayerClean
//
//  Created: 2025-11-26
//  Purpose: Centralized file loading and game library management
```

```
//
//  ARCHITECTURE:
//  - Replaces NotificationCenter-based file loading (tech debt
removal)
//  - Manages game library state
//  - Reusable by any view that needs file operations
//

import Foundation
import SwiftUI

/// Service for loading SGF files and managing game library
class FileLoadingService: ObservableObject {

    // MARK: - Published State

    /// List of games loaded from folder
    @Published var games: [SGFGameWrapper] = []

    /// Currently selected game
    @Published var selection: SGFGameWrapper?

    /// Loading indicator
    @Published var isLoadingGames: Bool = false

    // MARK: - Dependencies

    private weak var boardVM: BoardViewModel?

    // MARK: - Initialization

    init(boardVM: BoardViewModel? = nil) {
        self.boardVM = boardVM
    }

    // MARK: - Public API

    /// Load a single SGF file
    func loadFile(from url: URL) {
        print("📂 Loading SGF file: \(url.path)")

        do {
            let data = try Data(contentsOf: url)
            let text = String(data: data, encoding: .utf8) ??
String(decoding: data, as: UTF8.self)
            let tree = try SGFParser.parse(text: text)
            let game = SGFGame.from(tree: tree)
            let wrapper = SGFGameWrapper(url: url, game: game)

            print("✅ Successfully loaded: \(wrapper.title ??
"Untitled")")

            // Load into BoardViewModel
            boardVM?.loadGame(wrapper)

        } catch {
            print("❌ Failed to load SGF file: \(error)")
        }
    }

    /// Load all games from a folder
    func loadGamesFromFolder(_ folderURL: URL) {
        // Set loading state
        DispatchQueue.main.async {
            self.isLoadingGames = true
        }

        // Save folder URL to settings
        AppSettings.shared.folderURL = folderURL

        // Load in background to avoid blocking UI
        DispatchQueue.global(qos: .userInitiated).async {
            let fm = FileManager.default
            var sgfURLs: [URL] = []

            if let enumerator = fm.enumerator(at: folderURL,
includingPropertiesForKeys: [.isRegularFileKey], options:
[.skipsHiddenFiles]) {
                for case let fileURL as URL in enumerator {
                    if fileURL.pathExtension.lowercased() == "sgf" {
                        sgfURLs.append(fileURL)
                    }
                }
            }

            // Sort by path
            sgfURLs.sort { $0.path.localizedStandardCompare($1.path)
== .orderedAscending }

            // Shuffle if requested
            if AppSettings.shared.shuffleGameOrder {
                sgfURLs.shuffle()
                print("🔀 Shuffled game order")
            }

            print("📂 Found \(sgfURLs.count) games, loading all...")

            var parsed: [SGFGameWrapper] = []
            for fileURL in sgfURLs {
                do {
                    let data = try Data(contentsOf: fileURL)
                    let text = String(data: data, encoding: .utf8) ??
String(decoding: data, as: UTF8.self)
                    let tree = try SGFParser.parse(text: text)
                    let game = SGFGame.from(tree: tree)
                    parsed.append(SGFGameWrapper(url: fileURL, game:
game))
                } catch {
                    print("❌ Failed to parse \
(fileURL.lastPathComponent): \(error)")
                }
            }

            // Update on main thread
            DispatchQueue.main.async {
                self.games = parsed

                // Auto-select and load first game if enabled
                if AppSettings.shared.startGameOnLaunch, let firstGame
= parsed.first {
                    self.selection = firstGame
                    self.loadFile(from: firstGame.url)
                    print("🎮 Auto-launching first game: \
(firstGame.title ?? "Untitled")")
                } else {
                    self.selection = parsed.first
                }

                self.isLoadingGames = false
                print("📚 Loaded \(parsed.count) games")
            }
        }
    }
}

// FILE: ./Services/OGSClient.swift
// MARK: - File: OGSClient.swift (v4.248)
import Foundation
import Combine

class OGSClient: NSObject, ObservableObject,
URLSessionWebSocketDelegate {
    @Published var isConnected = false
    @Published var isSocketAuthenticated = false
    @Published var isAuthenticated = false
    @Published var isSubscribedToSeekgraph = false
    @Published var username: String?
    @Published var playerID: Int?
    @Published var userJWT: String? { didSet { if userJWT != nil &&
isConnected { sendSocketAuth() } } }
```

```
    @Published var lastError: String? = nil

    @Published var activeGameID: Int?
    @Published var activeGameAuth: String?
    @Published var playerColor: Stone?
    @Published var currentPlayerID: Int?
    @Published var availableGames: [OGSChallenge] = []
    @Published var trafficLogs: [NetworkLogEntry] = []

    @Published var blackPlayerID: Int?; @Published var whitePlayerID:
Int?
    @Published var blackPlayerName: String?; @Published var
whitePlayerName: String?
    @Published var blackPlayerRank: Double?; @Published var
whitePlayerRank: Double?
    @Published var blackTimeRemaining: TimeInterval?; @Published var
whiteTimeRemaining: TimeInterval?
    @Published var undoRequestedUsername: String? = nil
    @Published var undoRequestedMoveNumber: Int? = nil

    internal var webSocketTask: URLSessionWebSocketTask?
    internal var urlSession: URLSession?
    private var pingTimer: Timer?

    override init() {
        super.init()
        let config = URLSessionConfiguration.default
        config.httpCookieStorage = HTTPCookieStorage.shared
        if let data = KeychainHelper.load(service:
"com.davemarvit.SGFPlayerClean.OGS", account: "session_id"),
           let sid = String(data: data, encoding: .utf8) {
            let cookie = HTTPCookie(properties: [.domain: ".online-
go.com", .path: "/", .name: "sessionid", .value: sid, .secure:
"TRUE", .expires: NSDate(timeIntervalSinceNow: 31556926)])
            config.httpCookieStorage?.setCookie(cookie!)
        }
        self.urlSession = URLSession(configuration: config, delegate:
self, delegateQueue: OperationQueue())
        fetchUserConfig();
DispatchQueue.main.asyncAfter(deadline: .now() + 1.0)
{ self.connect() }
    }

    func connect() {
        guard let url = URL(string: "wss://wsp.online-go.com/") else
{ return }
        var request = URLRequest(url: url); request.setValue("https://
online-go.com", forHTTPHeaderField: "Origin")
        webSocketTask = urlSession?.webSocketTask(with: request);
webSocketTask?.resume(); receiveMessage()
```

```
    }

    private func sendOGS(_ event: String, _ payload: Any, _ moveNum: 
Int? = nil) {
        var array: [Any] = [event, payload]
        if let mn = moveNum { array.append(mn) }
        if let data = try? JSONSerialization.data(withJSONObject: 
array), let s = String(data: data, encoding: .utf8) {
            sendSocketMessage(s)
        }
    }

    func handleSocketRawMessage(_ text: String) {
        if text == "2" { sendSocketMessage("3"); return }
        guard let data = text.data(using: .utf8), let array = try? 
JSONSerialization.jsonObject(with: data) as? [Any], array.count >= 2, 
let event = array[0] as? String else { return }
        let payload = array[1]
        if event == "seekgraph/global" {
            if let list = payload as? [[String: Any]] { for d in list 
{ processSeekgraphItem(d) } }
            else if let d = payload as? [String: Any] 
{ processSeekgraphItem(d) }
        } else if event.hasSuffix("/gamedata") {
            NotificationCenter.default.post(name: 
NSNotification.Name("OGSGameDataReceived"), object: nil, userInfo: 
["gameData": payload])
        } else if event.hasSuffix("/move") {
            NotificationCenter.default.post(name: 
NSNotification.Name("OGSMoveReceived"), object: nil, userInfo: payload 
as? [String: Any])
        } else if event == "authenticate" {
            DispatchQueue.main.async { self.isSocketAuthenticated = 
true; self.startHighLevelPing() }
        }
    }

    private func processSeekgraphItem(_ dict: [String: Any]) {
        DispatchQueue.main.async {
            guard let id = (dict["challenge_id"] as? Int) ?? 
(dict["game_id"] as? Int) else { return }
            if dict["delete"] != nil { self.availableGames.removeAll 
{ $0.id == id } }
            else if let data = try? 
JSONSerialization.data(withJSONObject: dict), let challenge = try? 
JSONDecoder().decode(OGSChallenge.self, from: data) {
                if let idx = self.availableGames.firstIndex(where: 
{ $0.id == id }) { self.availableGames[idx] = challenge }
                else { self.availableGames.append(challenge) }
            }
        }
    }

    func sendSocketAuth() { if let jwt = self.userJWT, isConnected
{ sendOGS("authenticate", ["jwt": jwt]) } }
    func sendMove(gameID: Int, x: Int, y: Int, moveNumber: Int)
{ sendOGS("game/move", ["game_id": gameID, "move":
SGFCoordinates.toSGF(x: x, y: y)], moveNumber) }
    func sendPass(gameID: Int, moveNumber: Int) { sendOGS("game/pass",
["game_id": gameID], moveNumber) }
    func resignGame(gameID: Int) { sendOGS("game/resign", ["game_id":
gameID]) }
    func startAutomatch() { /* Structural link for OGSGameViewModel
*/ }
    func cancelChallenge(challengeID: Int) { sendOGS("seek_graph/
remove", ["challenge_id": challengeID]) }
    func subscribeToSeekgraph() { if isConnected
{ sendOGS("seek_graph/connect", ["channel": "global"]);
DispatchQueue.main.async { self.isSubscribedToSeekgraph = true } } }
    func connectToGame(gameID: Int) {
        DispatchQueue.main.async { self.activeGameID = gameID }
        sendOGS("game/connect", ["game_id": gameID, "chat": true]);
sendOGS("ui-pushes/subscribe", ["channel": "game-\(gameID)"])
    }
    func acceptChallenge(challengeID: Int, completion: @escaping
(Int?, String?) -> Void) {
        guard let url = URL(string: "https://online-go.com/api/v1/
challenges/\(challengeID)/accept") else { return }
        ensureCSRFToken { token in
            var req = URLRequest(url: url); req.httpMethod = "POST";
req.setValue("https://online-go.com", forHTTPHeaderField: "Origin");
req.setValue("https://online-go.com/", forHTTPHeaderField: "Referer")
            if let t = token { req.setValue(t, forHTTPHeaderField: "X-
CSRFToken") }
            self.urlSession?.dataTask(with: req) { d, _, _ in if let
data = d, let json = try? JSONSerialization.jsonObject(with: data) as?
[String: Any], let gId = json["game_id"] as? Int { completion(gId,
nil) } else { completion(nil, "Error") } }.resume()
        }
    }
    func createChallenge(setup: ChallengeSetup, completion: @escaping
(Bool, String?) -> Void) {
        ensureCSRFToken { token in
            guard let token = token, let url = URL(string: "https://
online-go.com/api/v1/challenges") else { return }
            var req = URLRequest(url: url); req.httpMethod = "POST";
req.setValue("application/json", forHTTPHeaderField: "Content-Type");
req.setValue("https://online-go.com", forHTTPHeaderField: "Origin");
req.setValue(token, forHTTPHeaderField: "X-CSRFToken")
            req.httpBody = try? JSONSerialization.data(withJSONObject:
```

```
setup.toDictionary()); self.urlSession?.dataTask(with: req) { _, r, _
in completion((r as? HTTPURLResponse)?.statusCode == 201,
nil) }.resume()
        }
    }
    func fetchGameState(gameID: Int, completion: @escaping ([String:
Any]?) -> Void) {
        guard let url = URL(string: "https://online-go.com/api/v1/
games/\(gameID)") else { return }
        var req = URLRequest(url: url); req.setValue("https://online-
go.com", forHTTPHeaderField: "Origin")
        urlSession?.dataTask(with: req) { data, _, _ in if let d =
data, let json = try? JSONSerialization.jsonObject(with: d) as?
[String: Any] { completion(json) } }.resume()
    }
    func sendUndoReject(gameID: Int) { sendOGS("game/undo/reject",
["game_id": gameID]) }
    func sendUndoAccept(gameID: Int) { sendOGS("game/undo/accept",
["game_id": gameID]) }
    func sendUndoRequest(gameID: Int, moveNumber: Int)
{ sendOGS("game/undo/request", ["game_id": gameID, "move_number":
moveNumber]) }
    internal func sendSocketMessage(_ t: String)
{ webSocketTask?.send(.string(t)) { _ in } }
    internal func receiveMessage() { webSocketTask?.receive { [weak
self] r in if case .success(let m) = r, case .string(let t) = m
{ self?.handleSocketRawMessage(t); self?.receiveMessage() } else
{ DispatchQueue.main.asyncAfter(deadline: .now() + 5.0)
{ self?.connect() } } } }
    private func startHighLevelPing() { pingTimer?.invalidate();
pingTimer = Timer.scheduledTimer(withTimeInterval: 5.0, repeats: true)
{ [weak self] _ in let ts = Int(Date().timeIntervalSince1970 * 1000);
self?.sendOGS("net/ping", ["client": ts]) } }
    func urlSession(_ s: URLSession, webSocketTask:
URLSessionWebSocketTask, didOpenWithProtocol p: String?)
{ DispatchQueue.main.async { self.isConnected = true;
self.sendSocketAuth(); self.subscribeToSeekgraph() } }
    func fetchUserConfig() {
        guard let url = URL(string: "https://online-go.com/api/v1/ui/
config") else { return }
        var req = URLRequest(url: url); req.setValue("https://online-
go.com", forHTTPHeaderField: "Origin")
        urlSession?.dataTask(with: req) { data, _, _ in if let d =
data, let json = try? JSONSerialization.jsonObject(with: d) as?
[String: Any], let user = json["user"] as? [String: Any]
{ DispatchQueue.main.async { self.username = user["username"] as?
String; self.playerID = user["id"] as? Int; self.userJWT = user["jwt"]
as? String; self.isAuthenticated = true } } }.resume()
    }
    func ensureCSRFToken(completion: @escaping (String?) -> Void) {
        guard let url = URL(string: "https://online-go.com/api/v1/ui/
config") else { return }
        var req = URLRequest(url: url); req.setValue("https://online-
go.com", forHTTPHeaderField: "Origin")
        urlSession?.dataTask(with: req) { _, _, _ in
completion(self.urlSession?.configuration.httpCookieStorage?.cookies(f
or: URL(string: "https://online-go.com")!)?.first(where: { $0.name ==
"csrftoken" })?.value) }.resume()
    }
}
```