```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Voxel Architect v5.1 - Sync Update</title>
   <style>
      body { margin: 0; background: #000; overflow: hidden; font-family: 'Courier New',
monospace; }
      #input_video { position: absolute; width: 100vw; height: 100vh; object-fit: cover; transform:
scaleX(-1); z-index: 1; }
      #three_canvas { position: absolute; top: 0; left: 0; z-index: 5; pointer-events: none; }
      #biometric_canvas { position: absolute; width: 100vw; height: 100vh; z-index: 10;
transform: scaleX(-1); pointer-events: none; }
      #ui {
         position: absolute; top: 20px; left: 20px; z-index: 100;
         color: #00f0ff; font-weight: bold; font-size: 14px;
         text-shadow: 0 0 10px #00f0ff; border-left: 3px solid #00f0ff; padding-left: 15px;
         background: rgba(0,0,0,0.6); padding: 15px;
      }
      .stat-val { color: #fff; }
   </style>
</head>
<body>
   <div id="ui">
      <div>BIO_SYNC: ARCHITECT_OS_v4.1</div>
      <div>STATE: <span id="mode" class="stat-val">INITIALIZING</span></div>
      <div>VOXELS: <span id="count" class="stat-val">0</span></div>
      <div style="font-size: 10px; margin-top: 5px; color: #ff3333;">2 FISTS: HOLD TO RESET |
2 PALMS: HOLD TO ROTATE</div>
      <div style="font-size: 10px; color: #ff00ff;">L-THUMB DOWN: BURST | L-THUMB UP:
RESTORE</div>
      <div style="font-size: 10px; color: #00ff00;">L-VICTORY: TOGGLE COLOR | R-VICTORY:
DISCO (PALM TO STOP)</div>
   </div>

   <video id="input_video" autoplay playsinline></video>
   <canvas id="three_canvas"></canvas>
   <canvas id="biometric_canvas"></canvas>

   <script src="https://cdn.jsdelivr.net/npm/@mediapipe/hands/hands.js"></script>
   <script src="https://cdn.jsdelivr.net/npm/@mediapipe/camera_utils/camera_utils.js"></script>
   <script src="https://cdnjs.cloudflare.com/ajax/libs/three.js/r128/three.min.js"></script>

   <script>
```

```
        const videoElement = document.getElementById('input_video');
        const bioCanvas = document.getElementById('biometric_canvas');
        const bioCtx = bioCanvas.getContext('2d');
        const modeEl = document.getElementById('mode');
        const countEl = document.getElementById('count');

        const scene = new THREE.Scene();
        const camera = new THREE.PerspectiveCamera(45, window.innerWidth /
window.innerHeight, 0.1, 1000);
        const renderer = new THREE.WebGLRenderer({ canvas:
document.getElementById('three_canvas'), antialias: true, alpha: true });
        renderer.setSize(window.innerWidth, window.innerHeight);

        const voxelGroup = new THREE.Group();
        scene.add(voxelGroup);
        const currentSketch = new THREE.Group();
        voxelGroup.add(currentSketch);

        const gridSize = 1.2;
        const placedVoxels = new Map();

        const crosshair = new THREE.Mesh(
          new THREE.BoxGeometry(gridSize, gridSize, gridSize),
          new THREE.MeshBasicMaterial({ color: 0x00f0ff, wireframe: true, transparent: true,
opacity: 0.5 })
        );
        scene.add(crosshair);

        scene.add(new THREE.AmbientLight(0xffffff, 0.5));
        const sun = new THREE.DirectionalLight(0xffffff, 1.0);
        sun.position.set(5, 5, 5);
        scene.add(sun);
        camera.position.z = 20;

        let smoothedLandmarks = { Left: [], Right: [] };
        let gravityEnabled = false;
        let rainbowActive = false;
        let gravityTimer = 0;
        let restoreTimer = 0;
        const GRAVITY_HOLD = 800;

        // --- GLOBAL COLOR PALETTE ---
        const colorPalette = [
          0x00f0ff, 0xff0000, 0x0000ff, 0x00ff00, 0xffff00, 0xff00ff,
        0xffa500, 0x800080, 0x00ff7f, 0xff1493, 0x7fff00, 0x40e0d0,
        0xffd700, 0xff4500, 0x9370db, 0x00ced1, 0xf08080, 0xadff2f,
        0xff6347, 0x00bfff, 0xda70d6
    ];
    let globalColorIndex = 0;
    let leftPeaceWasActive = false;

    function getFloorY() {
        const vFOV = THREE.MathUtils.degToRad(camera.fov);
        const height = 2 * Math.tan(vFOV / 2) * camera.position.z;
        return -(height / 2) + (gridSize / 2);
    }

    function drawHUDCircle(ctx, x, y, progress, color) {
        ctx.beginPath();
        ctx.arc(x, y, 35, -Math.PI/2, (-Math.PI/2) + (Math.PI * 2 * progress));
        ctx.lineWidth = 5; ctx.strokeStyle = color; ctx.stroke();
        ctx.setLineDash([3, 5]);
        ctx.beginPath(); ctx.arc(x, y, 30, 0, Math.PI * 2); ctx.lineWidth = 1; ctx.stroke();
        ctx.setLineDash([]);
    }

    function drawCyberHand(ctx, landmarks, label) {
        if (!smoothedLandmarks[label] || smoothedLandmarks[label].length === 0) {
            smoothedLandmarks[label] = landmarks.map(p => ({...p}));
        } else {
            landmarks.forEach((p, i) => {
                smoothedLandmarks[label][i].x += (p.x - smoothedLandmarks[label][i].x) * 0.45;
                smoothedLandmarks[label][i].y += (p.y - smoothedLandmarks[label][i].y) * 0.45;
                smoothedLandmarks[label][i].z += (p.z - smoothedLandmarks[label][i].z) * 0.1;
            });
        }
        const pts = smoothedLandmarks[label];
        ctx.shadowBlur = 10; ctx.shadowColor = "#00f0ff";
        ctx.beginPath(); ctx.strokeStyle = "rgba(0, 240, 255, 0.6)"; ctx.lineWidth = 2;
        const CONNECTIONS =
[[0,1],[1,2],[2,3],[3,4],[0,5],[5,6],[6,7],[7,8],[9,10],[10,11],[11,12],[13,14],[14,15],[15,16],[0,17],[17,1
8],[18,19],[19,20],[5,9],[9,13],[13,17],[0,5]];
        CONNECTIONS.forEach(([a, b]) => {
            ctx.moveTo(pts[a].x * bioCanvas.width, pts[a].y * bioCanvas.height);
            ctx.lineTo(pts[b].x * bioCanvas.width, pts[b].y * bioCanvas.height);
        });
        ctx.stroke();
        pts.forEach((pt, i) => {
            const x = pt.x * bioCanvas.width, y = pt.y * bioCanvas.height;
            if ([4, 8, 12, 16, 20].includes(i)) {
                ctx.strokeStyle = "#00f0ff"; ctx.strokeRect(x - 6, y - 6, 12, 12);
            } else { ctx.fillStyle = "#fff"; ctx.fillRect(x - 2, y - 2, 4, 4); }
        });
    }

    let isGrabbing = false, grabTimer = 0;
    let grabOffset = new THREE.Vector3();
    let isBuilding = false, buildTimer = 0;
    let isErasing = false, eraseTimer = 0;
    let resetTimer = 0, rotateTimer = 0;
    let startPinchPos = null, activeAxis = null;
    let sketchKeys = new Set();

    const GRAB_HOLD = 500;
    const INTENT_HOLD = 500;
    const RESET_HOLD = 1000;
    const ROTATE_HOLD = 1000;
    const pinchThreshold = 0.05;

    function getDist(p1, p2) { return
Math.sqrt(Math.pow(p1.x-p2.x,2)+Math.pow(p1.y-p2.y,2)+(p1.z&&p2.z?Math.pow(p1.z-p2.z,2):0
)); }

    function onResults(results) {
        bioCtx.clearRect(0, 0, bioCanvas.width, bioCanvas.height);
        crosshair.visible = false;

        if (!results.multiHandLandmarks) {
            grabTimer = 0; buildTimer = 0; eraseTimer = 0;
            resetTimer = 0; rotateTimer = 0; gravityTimer = 0; restoreTimer = 0;
            return;
        }

        let lHand = null, rHand = null;
        results.multiHandedness.forEach((hand, idx) => {
            const landmarks = results.multiHandLandmarks[idx];
            drawCyberHand(bioCtx, landmarks, hand.label);
            if(hand.label === 'Left') lHand = smoothedLandmarks['Left'];
            if(hand.label === 'Right') rHand = smoothedLandmarks['Right'];
        });

        if (lHand && rHand) {
            const lFist = lHand[8].y > lHand[6].y && lHand[12].y > lHand[10].y && lHand[16].y >
lHand[14].y;
            const rFist = rHand[8].y > rHand[6].y && rHand[12].y > rHand[10].y && rHand[16].y >
rHand[14].y;
            const lPalm = lHand[8].y < lHand[6].y && lHand[12].y < lHand[10].y && lHand[20].y <
lHand[18].y;
            const rPalm = rHand[8].y < rHand[6].y && rHand[12].y < rHand[10].y && rHand[20].y <
rHand[18].y;

            if (lFist && rFist) {
              if (resetTimer < RESET_HOLD) {
                resetTimer += 16;
                drawHUDCircle(bioCtx, bioCanvas.width / 2, bioCanvas.height / 2,
resetTimer/RESET_HOLD, "#ff0055");
                modeEl.innerText = "SYSTEM: HOLD TO RESET...";
              } else if (resetTimer >= RESET_HOLD && resetTimer < 2000) {
                voxelGroup.position.set(0, 0, 0);
                voxelGroup.rotation.set(0, 0, 0);
                modeEl.innerText = "SYSTEM: HARD_RESET COMPLETE";
                resetTimer = 2000;
              }
              return;
            } else { resetTimer = 0; }

            if (lPalm && rPalm) {
              if (rotateTimer < ROTATE_HOLD) {
                rotateTimer += 16;
                drawHUDCircle(bioCtx, bioCanvas.width / 2, bioCanvas.height / 2,
rotateTimer/ROTATE_HOLD, "#00f0ff");
                modeEl.innerText = "SYSTEM: HOLD TO ENABLE ROTATION...";
              } else {
                modeEl.innerText = "SYSTEM: GLOBAL_ROTATE ACTIVE";
                voxelGroup.rotation.y += (rHand[9].x - lHand[9].x - 0.5) * 0.05;
                voxelGroup.rotation.x += (rHand[9].y - lHand[9].y) * 0.05;
              }
              return;
            } else { rotateTimer = 0; }
          }

          if (lHand) {
            const fingersCurled = lHand[8].y > lHand[6].y && lHand[12].y > lHand[10].y &&
lHand[16].y > lHand[14].y && lHand[20].y > lHand[18].y;
            const isFist = fingersCurled && getDist(lHand[4], lHand[12]) < 0.1;
            const isPalm = lHand[8].y < lHand[6].y && lHand[12].y < lHand[10].y && lHand[20].y <
lHand[18].y;
            const isThumbDown = lHand[4].y > lHand[3].y && lHand[4].y > lHand[17].y &&
fingersCurled;
            const isThumbUp = lHand[4].y < lHand[3].y && lHand[4].y < lHand[5].y &&
fingersCurled;

            // LEFT HAND PEACE SIGN (TOGGLE COLOR)
            const isLeftPeace = lHand[8].y < lHand[6].y && lHand[12].y < lHand[10].y &&
lHand[16].y > lHand[14].y && lHand[20].y > lHand[18].y;
            if (isLeftPeace && !leftPeaceWasActive) {
               globalColorIndex = (globalColorIndex + 1) % colorPalette.length;
               leftPeaceWasActive = true;
            } else if (!isLeftPeace) {
               leftPeaceWasActive = false;
            }

            if (isPalm) {
               isGrabbing = false;
               grabTimer = 0;
               modeEl.innerText = "BIO_LINK: SCANNING";
            }

            if (isThumbDown && !isFist) {
               restoreTimer = 0;
               if (!gravityEnabled) {
                  if (gravityTimer < GRAVITY_HOLD) {
                     gravityTimer += 16;
                     drawHUDCircle(bioCtx, lHand[4].x * bioCanvas.width, lHand[4].y *
bioCanvas.height, gravityTimer/GRAVITY_HOLD, "#ff00ff");
                     modeEl.innerText = "BIO_LINK: INITIATING BURST...";
                  } else {
                     gravityEnabled = true;
                     initiateGravityFall();
                     modeEl.innerText = "BIO_LINK: GRAVITY_ACTIVE";
                     gravityTimer = 0;
                  }
               }
            } else if (isThumbUp && !isFist) {
               gravityTimer = 0;
               if (gravityEnabled) {
                  if (restoreTimer < GRAVITY_HOLD) {
                     restoreTimer += 16;
```

```
                drawHUDCircle(bioCtx, lHand[4].x * bioCanvas.width, lHand[4].y *
bioCanvas.height, restoreTimer/GRAVITY_HOLD, "#00ff88");
                modeEl.innerText = "BIO_LINK: RESTORING COORDS...";
              } else {
                gravityEnabled = false;
                modeEl.innerText = "BIO_LINK: STRUCTURE_RESTORED";
                restoreTimer = 0;
              }
            }
          } else {
            gravityTimer = 0; restoreTimer = 0;
          }

          if (isFist && !isThumbDown && !isThumbUp) {
            if (grabTimer < GRAB_HOLD) {
              grabTimer += 16;
              drawHUDCircle(bioCtx, lHand[0].x * bioCanvas.width, lHand[0].y *
bioCanvas.height, grabTimer/GRAB_HOLD, "#ffbb00");
            } else {
              const handWorldPos = new THREE.Vector3((0.5 - lHand[9].x) * 25, (0.5 -
lHand[9].y) * 18, 0);
              if (!isGrabbing) { grabOffset.copy(voxelGroup.position).sub(handWorldPos);
isGrabbing = true; }
              voxelGroup.position.copy(handWorldPos).add(grabOffset);
              modeEl.innerText = "BIO_LINK: GRABBED";
            }
          } else if (isGrabbing) {
            const handWorldPos = new THREE.Vector3((0.5 - lHand[9].x) * 25, (0.5 -
lHand[9].y) * 18, 0);
            voxelGroup.position.copy(handWorldPos).add(grabOffset);
          }
        }

        if (rHand) {
          const thumbTip = rHand[4], indexTip = rHand[8], midTip = rHand[12];
          const pinchingNow = getDist(thumbTip, indexTip) < pinchThreshold;
          const pointingNow = indexTip.y < rHand[6].y && midTip.y > rHand[10].y;
          const palmOpen = rHand[8].y < rHand[6].y && rHand[12].y < rHand[10].y &&
rHand[20].y < rHand[18].y;

          const isPeace = indexTip.y < rHand[6].y && midTip.y < rHand[10].y && rHand[16].y >
rHand[14].y && rHand[20].y > rHand[18].y;
          if (isPeace) rainbowActive = true;
          else if (palmOpen) rainbowActive = false;
```

```
            const px = indexTip.x * bioCanvas.width, py = indexTip.y * bioCanvas.height;
            const worldPos = new THREE.Vector3((0.5 - indexTip.x) * 25, (0.5 - indexTip.y) * 18,
0);
            const localPos = voxelGroup.worldToLocal(worldPos.clone());

            let gx = Math.round(localPos.x / gridSize) * gridSize;
            let gy = Math.round(localPos.y / gridSize) * gridSize;
            let gz = 0;

            const lPinching = lHand && getDist(lHand[4], lHand[8]) < pinchThreshold;

            if (lPinching && pointingNow && !palmOpen) {
               buildTimer = 0;
               if (eraseTimer < INTENT_HOLD) {
                  eraseTimer += 16;
                  drawHUDCircle(bioCtx, px, py, eraseTimer/INTENT_HOLD, "#ff3333");
                  modeEl.innerText = "INTENT: ERASER_LOCKING...";
               } else {
                  isErasing = true;
                  const key = `${gx.toFixed(1)},${gy.toFixed(1)},${gz.toFixed(1)}`;
                  if (placedVoxels.has(key)) {
                     voxelGroup.remove(placedVoxels.get(key));
                     placedVoxels.delete(key);
                     countEl.innerText = placedVoxels.size;
                  }
                  modeEl.innerText = "INTENT: ERASER_ACTIVE";
               }
            } else if (pinchingNow && !isGrabbing && !palmOpen) {
               eraseTimer = 0;
               if (buildTimer < INTENT_HOLD) {
                  buildTimer += 16;
                  drawHUDCircle(bioCtx, px, py, buildTimer/INTENT_HOLD, "#00ffcc");
                  modeEl.innerText = "INTENT: BUILD_SYNCING...";
               } else {
                  if (!isBuilding) { startPinchPos = { x: gx, y: gy, z: gz }; sketchKeys.clear();
isBuilding = true; activeAxis = null; }
                  else {
                     const dx = Math.abs(gx - startPinchPos.x), dy = Math.abs(gy -
startPinchPos.y);
                     if (!activeAxis && (dx > 0.4 || dy > 0.4)) {
                        if (dx >= dy) activeAxis = 'x';
                        else activeAxis = 'y';
                     }
```

```
                        let tx = startPinchPos.x, ty = startPinchPos.y;
                        if (activeAxis === 'x') tx = gx; else if (activeAxis === 'y') ty = gy;
                        addSketchVoxel(tx, ty, gz);
                    }
                    modeEl.innerText = "INTENT: BUILDING";
                }
            } else {
                if (palmOpen) { if (isBuilding) commitVoxels(); isBuilding = false; isErasing = false;
buildTimer = 0; eraseTimer = 0; modeEl.innerText = "BIO_LINK: NAVIGATING"; }
            }

            if (isBuilding || buildTimer > 0 || isErasing || eraseTimer > 0) {
                crosshair.visible = true;
                crosshair.position.copy(voxelGroup.localToWorld(new THREE.Vector3(gx, gy, gz)));
                crosshair.material.color.set((isErasing || eraseTimer > 0) ? 0xff3333 :
colorPalette[globalColorIndex]);
            }
        }
    }

    function initiateGravityFall() {
        placedVoxels.forEach((v) => {
            v.velocity.set((Math.random() - 0.5) * 0.8, 0.4 + Math.random() * 0.5, (Math.random() -
0.5) * 0.8);
            v.isBouncing = false;
        });
    }

    function addSketchVoxel(x, y, z) {
        const key = `${x.toFixed(1)},${y.toFixed(1)},${z.toFixed(1)}`;
        if (sketchKeys.has(key) || placedVoxels.has(key)) return;
        const mesh = new THREE.Mesh(new THREE.BoxGeometry(gridSize*0.98,
gridSize*0.98, gridSize*0.98), new THREE.MeshBasicMaterial({ color:
colorPalette[globalColorIndex], wireframe: true }));
        mesh.position.set(x, y, z);
        currentSketch.add(mesh);
        sketchKeys.add(key);
    }

    function commitVoxels() {
        while(currentSketch.children.length > 0) {
            const f = currentSketch.children[0];
            const key =
`${f.position.x.toFixed(1)},${f.position.y.toFixed(1)},${f.position.z.toFixed(1)}`;
```

```
            const cube = createFinalCube(f.position.x, f.position.y, f.position.z);
            voxelGroup.add(cube);
            placedVoxels.set(key, cube);
            currentSketch.remove(f);
         }
         countEl.innerText = placedVoxels.size;
      }

      function createFinalCube(x, y, z) {
         const g = new THREE.BoxGeometry(gridSize*0.95, gridSize*0.95, gridSize*0.95);
         const m = new THREE.MeshPhongMaterial({ color: 0x001122, emissive:
colorPalette[globalColorIndex], emissiveIntensity: 0.4, transparent: true, opacity: 0.8 });
         const mesh = new THREE.Mesh(g, m);
         mesh.position.set(x, y, z);
         mesh.origin = new THREE.Vector3(x, y, z);
         mesh.velocity = new THREE.Vector3(0,0,0);
         mesh.add(new THREE.LineSegments(new THREE.EdgesGeometry(g), new
THREE.LineBasicMaterial({ color: colorPalette[globalColorIndex] })));
         return mesh;
      }

      const hands = new Hands({locateFile: (f) =>
`https://cdn.jsdelivr.net/npm/@mediapipe/hands/${f}`});
      hands.setOptions({ maxNumHands: 2, modelComplexity: 1, minDetectionConfidence: 0.8,
minTrackingConfidence: 0.8 });
      hands.onResults(onResults);
      new Camera(videoElement, { onFrame: async () => { bioCanvas.width =
videoElement.videoWidth; bioCanvas.height = videoElement.videoHeight; await
hands.send({image: videoElement}); }, width: 1280, height: 720 }).start();

      function animate() {
```