```
import streamlit as st
from PIL import Image, ImageOps, ImageDraw
import io, random, heapq, time
import hashlib  # for implementing fingerprint for uploaded file
from io import BytesIO  # for file handling

tab1, tab2 = st.tabs([
    "Puzzle Game",
    "Code",
])

with tab1:
    st.audio("./songpuzzle.mp3", format="audio/mp3", autoplay=True, loop=True)

    st.markdown("""
    <link
     href="https://fonts.googleapis
     .com/css?family=Open+Sans:400,700&display=swap" rel="stylesheet">
    <style>
    /* Global font similar to Cayman */
    html, body, [class*="css"] {
        font-family: "Open Sans","Helvetica Neue",Helvetica,Arial,sans-serif;
    }
    .cayman-header {
        color: #fff;
        text-align: center;
        padding: 2.5rem 1rem;
        background-color: #159957;
        background-image: linear-gradient(120deg, #155799, #159957);
        border-radius: 20px;
        margin-bottom: 1.25rem;
    }
    </style>
    """, unsafe_allow_html=True)

    st.markdown("""
    <link
     href="https://fonts.googleapis
     .com/css?family=Open+Sans:400,600,700&display=swap" rel="stylesheet">
    <style>
    /* Base font */
    :root, html, body, [class*="css"] {
        font-family: "Open Sans","Helvetica Neue",Helvetica,Arial,sans-serif;
    }
    /* Sidebar container */
    section[data-testid="stSidebar"] {
        text-align: left;
        background-color: #159957;
        background-image: linear-gradient(120deg, #155799, #159957);

        border-right: 1px solid rgba(27,31,35,0.1);
```

```
}
</style>
""", unsafe_allow_html=True)

with st.sidebar:
    st.markdown(
        """
        <div class="cayman-sb-header">
            <h2>Puzzle Controls</h2>
        </div>
        """,
        unsafe_allow_html=True
    )

st.markdown('<div class="cayman-header"><h1>🧩 Puzzle Your Image 🧩</h1><p>Upload an image to play!</p></div>', unsafe_allow_html=True)

# Puzzle logic
GOAL = (1, 2, 3, 4, 5, 6, 7, 8, 0)

def index_to_rc(idx):
    return divmod(idx, 3)  # (row, col)

def rc_to_index(r, c):
    return r * 3 + c

def neighbors(state):
    """Return list of neighbor states reachable by moving the blank."""
    zero_idx = state.index(0)
    r, c = index_to_rc(zero_idx)
    moves = []
    for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
        nr, nc = r + dr, c + dc
        if 0 <= nr < 3 and 0 <= nc < 3:
            nidx = rc_to_index(nr, nc)
            ns = list(state)
            ns[zero_idx], ns[nidx] = ns[nidx], ns[zero_idx]
            moves.append(tuple(ns))
    return moves

def manhattan(s, goal=GOAL):
    dist = 0
    for idx, tile in enumerate(s):
        if tile == 0:
            continue
        goal_idx = goal.index(tile)
        r1, c1 = index_to_rc(idx)
        r2, c2 = index_to_rc(goal_idx)
        dist += abs(r1 - r2) + abs(c1 - c2)
    return dist
```

```
def is_solvable(state):
    """Check inversions ignoring blank for 8-puzzle."""
    flat = [x for x in state if x != 0]
    inv = sum(1 for i in range(len(flat)) for j in range(i + 1, len(flat))
     if flat[i] > flat[j])
    return inv % 2 == 0

def astar(start, goal=GOAL, max_nodes=200000):
    """Return list of states from start to goal (inclusive) or None if not
     found."""
    start = tuple(start)
    if start == goal:
        return [start]
    open_heap = []
    g = {start: 0}
    fstart = manhattan(start, goal)
    heapq.heappush(open_heap, (fstart, 0, start))
    came_from = {}
    closed = set()
    counter = 1
    nodes = 0
    while open_heap:
        _, _, current = heapq.heappop(open_heap)
        nodes += 1
        if nodes > max_nodes:
            return None  # fail gracefully for very large search
        if current == goal:
            path = [current]
            while current in came_from:
                current = came_from[current]
                path.append(current)
            return list(reversed(path))
        closed.add(current)
        for nb in neighbors(list(current)):
            if nb in closed:
                continue
            tentative_g = g[current] + 1
            if nb not in g or tentative_g < g[nb]:
                came_from[nb] = current
                g[nb] = tentative_g
                f = tentative_g + manhattan(nb, goal)
                counter += 1
                heapq.heappush(open_heap, (f, counter, nb))
    return None

# Initialize session_state
defaults = {
    "state": GOAL,
    "history": [],
    "solution": None,
```

```
    "sol_index": 0,
    "move_count": 0,
    "start_time": None,
    "auto_play": False,
}
for key, val in defaults.items():
    if key not in st.session_state:
        st.session_state[key] = val

if "last_upload_id" not in st.session_state:
    st.session_state.last_upload_id = None

# Image slicing
def crop_center_square(img: Image.Image) -> Image.Image:
    w, h = img.size
    m = min(w, h)
    left = (w - m) // 2
    top = (h - m) // 2
    return img.crop((left, top, left + m, top + m))

def slice_into_tiles(img: Image.Image):
    """Return dict mapping tile number (1..8) and 0->blank image"""
    img = crop_center_square(img)
    img = img.resize((450, 450), Image.LANCZOS)
    tiles = {}
    size = img.size[0] // 3
    num = 1
    for r in range(3):
        for c in range(3):
            box = (c * size, r * size, (c + 1) * size, (r + 1) * size)
            tile_img = img.crop(box)
            tiles[num] = tile_img
            num += 1
    blank = Image.new("RGB", (size, size), (255, 255, 255))
    draw = ImageDraw.Draw(blank)
    draw.rectangle([1, 1, size - 2, size - 2], outline=(200, 200, 200))
    tiles[0] = blank
    return tiles

# Session state tiles default
if "tiles" not in st.session_state:
    default = Image.new("RGB", (450, 450), (21, 109, 153))
    draw = ImageDraw.Draw(default)
    st.session_state.tiles = slice_into_tiles(default)
if "history" not in st.session_state:
    st.session_state.history = []
if "auto_play" not in st.session_state:
    st.session_state.auto_play = False
if "start_time" not in st.session_state:
    st.session_state.start_time = None
```

```
# Upload
uploaded = st.file_uploader("", type=["png", "jpg", "jpeg"])
if uploaded is not None:
    file_bytes = uploaded.getvalue()
    upload_fingerprint = (uploaded.name, len(file_bytes),
     hashlib.md5(file_bytes).hexdigest())
    if st.session_state.last_upload_id != upload_fingerprint:
        img = Image.open(BytesIO(file_bytes)).convert("RGB")
        st.session_state.tiles = slice_into_tiles(img)
        st.session_state.state = GOAL
        st.session_state.solution = None
        st.session_state.sol_index = 0
        st.session_state.last_upload_id = upload_fingerprint
        st.success("Image loaded and sliced. Start shuffling or play from
         the goal!")

# Controls
col1, col2, col3, col4 = st.columns([1, 1, 1, 1])

with col1:
    if st.sidebar.button("Shuffle pieces", key="btn_shuffle"):
        s = list(GOAL)
        moves = random.randint(20, 60)
        last = None
        for _ in range(moves):
            nbs = neighbors(tuple(s))
            if last and len(nbs) > 1:
                nbs = [nb for nb in nbs if nb != last]
            nxt = random.choice(nbs)
            last = tuple(s)
            s = list(nxt)
        st.session_state.state = tuple(s)
        st.session_state.history = []
        st.session_state.solution = None
        st.session_state.sol_index = 0
        st.session_state.move_count = 0
        st.session_state.start_time = time.time()

with col2:
    if st.sidebar.button("Reset to goal", key="btn_reset"):
        st.session_state.state = GOAL
        st.session_state.history = []
        st.session_state.solution = None
        st.session_state.sol_index = 0
        st.session_state.move_count = 0
        st.session_state.start_time = time.time()

with col3:
    if st.sidebar.button("Solve (A* — optimal)", key="btn_solve"):
        if not is_solvable(st.session_state.state):
            st.error("This configuration is not solvable!")
```

```
        else:
            with st.spinner("Running A*..."):
                t0 = time.time()
                path = astar(st.session_state.state, GOAL)
                t1 = time.time()
            if path is None:
                st.error("A* failed to find a solution within limits.")
            else:
                st.session_state.state = GOAL
                st.session_state.solution = path
                st.session_state.sol_index = 0
                st.session_state.auto_play = False
                st.success(f"Found solution in {len(path)-1} moves (time
                 {t1 - t0:.2f}s).")

with col4:
    if "move_count" not in st.session_state:
        st.session_state.move_count = 0
    if "start_time" not in st.session_state:
        st.session_state.start_time = None

with st.sidebar:
    st.divider()

# Status
st.markdown(f"**Current state (solvable: {'Yes' if
 is_solvable(st.session_state.state) else 'No'})**")
moves_so_far = st.session_state.move_count
elapsed = int(time.time() - st.session_state.start_time) if
 st.session_state.start_time else 0
minutes, seconds = divmod(elapsed, 60)
st.write(f"**Moves made:** {moves_so_far}")
st.write(f"**Time elapsed:** {minutes:02d}:{seconds:02d}")

# Puzzle grid display
state = list(st.session_state.state)
tiles = st.session_state.tiles
for r in range(3):
    cols = st.columns(3)
    for c in range(3):
        idx = rc_to_index(r, c)
        tile_num = state[idx]
        with cols[c]:
            if st.button(f"{tile_num}", key=f"tile_{idx}"):
                zero_idx = state.index(0)
                zr, zc = index_to_rc(zero_idx)
                tr, tc = index_to_rc(idx)
                if abs(zr - tr) + abs(zc - tc) == 1:
                    state[zero_idx], state[idx] = state[idx],
                     state[zero_idx]
                    st.session_state.history.append(tuple(state))
```

```
                    st.session_state.state = tuple(state)
                    st.session_state.solution = None
                    st.session_state.move_count += 1
                    st.rerun()
            st.image(tiles[tile_num], use_container_width=True)

# Step-by-step playback
if st.session_state.solution:
    st.markdown("---")
    st.subheader("Solution playback")

    sol = st.session_state.solution
    n_steps = len(sol) - 1

    st.write(f"Optimal path length: {n_steps} moves")

    c1, c2, c3, c4 = st.columns([1, 1, 1, 1])
    with c1:
        if st.sidebar.button("Prev", key="pb_prev"):
            st.session_state.sol_index = max(0, st.session_state.sol_index
             - 1)
            st.rerun()
    with c2:
        if st.sidebar.button("Next", key="pb_next"):
            st.session_state.sol_index = min(len(sol) - 1,
             st.session_state.sol_index + 1)
            st.rerun()
    with c3:
        if st.sidebar.button("Go to start", key="pb_start"):
            st.session_state.sol_index = 0
            st.rerun()
    with c4:
        if st.sidebar.button("Go to end", key="pb_end"):
            st.session_state.sol_index = len(sol) - 1
            st.rerun()

    ap1, ap2 = st.columns([1, 1])
    with ap1:
        if st.sidebar.button("Auto-play", key="pb_auto"):
            st.session_state.auto_play = True
            st.rerun()
    with ap2:
        if st.sidebar.button("Stop", key="pb_stop"):
            st.session_state.auto_play = False
            st.rerun()

    step_idx = st.session_state.sol_index
    st.write(f"Step {step_idx} / {n_steps}")

    cur = sol[step_idx]
    for r in range(3):
```

```
            cols = st.columns(3)
            for c in range(3):
                i = rc_to_index(r, c)
                tnum = cur[i]
                with cols[c]:
                    st.image(tiles[tnum], use_container_width=True)

        if st.session_state.auto_play:
            if st.session_state.sol_index < len(sol) - 1:
                time.sleep(0.4)
                st.session_state.sol_index += 1
                st.rerun()
            else:
                st.session_state.auto_play = False
                st.success("Yay! You Did it!")

    st.markdown("---")
    st.markdown("""
    **Instructions**
    - Upload an image to be cropped to center and sliced into 9 tiles.
    - Shuffle to scramble the puzzle pieces (or move tiles by clicking their
     buttons).
    - Press *Solve (A\*)* to compute an optimal path, then use **Solution
     playback** to step through or auto-play.
    """)

with tab2:
    st.subheader("Puzzle Game Python Code")
    pdf_path = "./puzzlegamecode.pdf"
    with open(pdf_path, "rb") as f:
        base64_pdf = base64.b64encode(f.read()).decode('utf-8')
    pdf_display = f'<iframe src="data:application/pdf;base64,{base64_pdf}"
     width="100%" height="600"></iframe>'
    st.markdown(pdf_display, unsafe_allow_html=True)
```