

""""

Tetris Game with Math Quiz, Difficulty Selection, High Bomb Frequency AND AUDIO

UPDATED: Fix Timeout showing '0' answer. Now shows REAL answer.

""""

```
import tkinter as tk
```

```
from tkinter import messagebox, ttk
```

```
import random
```

```
import os
```

```
try:
```

```
    from PIL import Image, ImageTk
```

```
    PIL_AVAILABLE = True
```

```
except Exception:
```

```
    PIL_AVAILABLE = False
```

```
try:
```

```
    import cv2
```

```
    OPENCV_AVAILABLE = True
```

```
except Exception:
```

```
    OPENCV_AVAILABLE = False
```

```
# -----
```

```
# AUDIO CONFIGURATION
```

```
# -----
```

```
AUDIO_FILES = {
```

```
    # Pastikan path file audio anda betul di sini
```

```
    'bgm':    r"C:\Users\User\Downloads\AUDIO GAME\relax-summer-397430.mp3",
```

```
    'move':   r"C:\Users\User\Downloads\AUDIO GAME\cinematic-piano-note-362716.mp3",
```

```
    'rotate': r"C:\Users\User\Downloads\AUDIO GAME\cinematic-piano-note-362716.mp3",
```

```

'drop': r"C:\Users\User\Downloads\AUDIO GAME\dropping-of-the-ring-404874.mp3",
'explode': r"C:\Users\User\Downloads\AUDIO GAME\loud-explosion-425457.mp3",
'gameover': r"C:\Users\User\Downloads\AUDIO GAME\game-over-417465.mp3",
'correct': r"C:\Users\User\Downloads\AUDIO GAME\correct-156911.mp3",
'wrong': r"C:\Users\User\Downloads\AUDIO GAME\wrong-answer-126515.mp3"
}

```

try:

```
import pygame
```

```
pygame.mixer.init()
```

```
AUDIO_ENABLED = True
```

except ImportError:

```
print("Pygame not found. Audio will be disabled. Run 'pip install pygame'")
```

```
AUDIO_ENABLED = False
```

```
# -----
```

```
# GAME CONFIGURATION
```

```
# -----
```

```
# UI: change BLOCK_SIZE to scale the tile size (increases on-screen width/height).
```

```
# To expand game width, increase BLOCK_SIZE (e.g., 40 or 48) or increase BOARD_WIDTH.
```

```
BLOCK_SIZE = 40 # was 30; change this value to make everything larger
```

```
BOARD_WIDTH = 10
```

```
BOARD_HEIGHT = 20
```

```
# Window/layout constants (adjust to change main window size/padding)
```

```
INFO_PANEL_WIDTH = 300
```

```
WINDOW_EXTRA_HEIGHT = 10
```

```
# Slight reduction applied to the game window height so it's a little shorter but still fits all elements
```

```
GAME_WINDOW_HEIGHT_REDUCTION = 60
```

```
# Increased menu width/height so main menu elements (difficulty cards, etc.) aren't cut off
```

```
MENU_WINDOW_WIDTH = 1000
```

```
MENU_WINDOW_HEIGHT = 540
```

```
COLORS = {
```

```
    'I': '#00F0F0', 'O': '#F0F000', 'T': '#A000F0', 'S': '#00F000',
```

```
    'Z': '#F00000', 'J': '#0000F0', 'L': '#F0A000', 'BOMB': '#FF4500'
```

```
}
```

```
SHAPES = {
```

```
    'I': [[1, 1, 1, 1]],
```

```
    'O': [[1, 1], [1, 1]],
```

```
    'T': [[0, 1, 0], [1, 1, 1]],
```

```
    'S': [[0, 1, 1], [1, 1, 0]],
```

```
    'Z': [[1, 1, 0], [0, 1, 1]],
```

```
    'J': [[1, 0, 0], [1, 1, 1]],
```

```
    'L': [[0, 0, 1], [1, 1, 1]]
```

```
}
```

```
DIFFICULTY_CONFIG = {
```

```
    'Easy': {'speed': 600, 'bomb_chance': 8, 'math_type': 'add_sub', 'score_mult': 1.0},
```

```
    'Normal': {'speed': 450, 'bomb_chance': 6, 'math_type': 'mult', 'score_mult': 1.5},
```

```
    'Hard': {'speed': 300, 'bomb_chance': 4, 'math_type': 'mixed', 'score_mult': 2.0}
```

```
}
```

```
# -----
```

```
# LEADERBOARD (file-based, 3 permanent notepads)
```

```
# -----
```

```
LEADERBOARD_DIR = os.path.join(os.path.dirname(__file__), 'leaderboards')
```

```
os.makedirs(LEADERBOARD_DIR, exist_ok=True)
```

```
LEADERBOARD_FILES = {  
    'Easy': os.path.join(LEADERBOARD_DIR, 'leaderboard_easy.txt'),  
    'Normal': os.path.join(LEADERBOARD_DIR, 'leaderboard_normal.txt'),  
    'Hard': os.path.join(LEADERBOARD_DIR, 'leaderboard_hard.txt'),  
}
```

```
def ensure_leaderboard_files_exist():
```

```
    for path in LEADERBOARD_FILES.values():  
        if not os.path.exists(path):  
            # create an empty permanent notepad (text file)  
            with open(path, 'w', encoding='utf-8') as f:  
                f.write("")
```

```
def append_score_to_leaderboard(difficulty, name, score):
```

```
    """Append player's name and integer score to the corresponding difficulty file."""  
    ensure_leaderboard_files_exist()  
    path = LEADERBOARD_FILES.get(difficulty, LEADERBOARD_FILES['Normal'])  
    name = str(name).replace(',', ' ').strip() or 'Anonymous'  
    with open(path, 'a', encoding='utf-8') as f:  
        f.write(f"{name},{int(score)}\n")
```

```
def load_leaderboard(difficulty):
```

```
    """Return list of (name, score) sorted by score desc. Players with same score will be grouped  
    together."""  
    ensure_leaderboard_files_exist()  
    path = LEADERBOARD_FILES.get(difficulty, LEADERBOARD_FILES['Normal'])  
    entries = []  
    with open(path, encoding='utf-8') as f:
```

```

for line in f:
    line = line.strip()
    if not line:
        continue
    parts = line.rsplit(',', 1)
    if len(parts) != 2:
        continue
    name = parts[0].strip()
    try:
        score = int(parts[1].strip())
    except ValueError:
        continue
    entries.append((name, score))

# Sort descending by score, then by name; equal scores will be contiguous (grouped)
entries.sort(key=lambda x: (-x[1], x[0].lower()))

return entries

```

```

def show_leaderboard_window(root, difficulty=None):

```

```

    ensure_leaderboard_files_exist()
    win = tk.Toplevel(root)
    win.title("Leaderboards")
    win.geometry("520x520")
    win.configure(bg='#2C3E50')
    win.transient(root)
    win.minsize(420, 360)

    # Header
    header = tk.Frame(win, bg='#1ABC9C')
    header.pack(fill='x')

```

```

tk.Label(header, text='Leaderboards', font=('Arial', 18, 'bold'), bg='#1ABC9C',
fg='white').pack(side='left', padx=12, pady=10)

close_btn = tk.Button(header, text='Close', command=win.destroy, bg='#C0392B', fg='white',
font=('Arial', 10, 'bold'))

close_btn.pack(side='right', padx=12, pady=8)

# Difficulty toggle buttons

btn_frame = tk.Frame(win, bg='#2C3E50')

btn_frame.pack(fill='x', pady=(10, 6))

selected_var = tk.StringVar(value=difficulty or 'Easy')

def make_btn(text):
    def _on():
        selected_var.set(text)
        _show(text)
        # update styles
        for b in btn_frame.winfo_children():
            if getattr(b, 'diff', None) == text:
                b.config(bg='#8E44AD', fg='white')
            else:
                b.config(bg='#34495E', fg='white')

    b = tk.Button(btn_frame, text=text, width=8, font=('Arial', 11, 'bold'), bg='#34495E', fg='white',
relief='flat', command=_on)

    b.diff = text

    return b

btn_easy = make_btn('Easy'); btn_easy.pack(side='left', padx=8)

btn_normal = make_btn('Normal'); btn_normal.pack(side='left', padx=8)

btn_hard = make_btn('Hard'); btn_hard.pack(side='left', padx=8)

```

```

# Content area with scroll

container = tk.Frame(win, bg='#2C3E50')
container.pack(fill='both', expand=True, padx=12, pady=(6, 12))

canvas = tk.Canvas(container, bg='#2C3E50', highlightthickness=0)
scrollbar = tk.Scrollbar(container, orient='vertical', command=canvas.yview)
scroll_frame = tk.Frame(canvas, bg='#2C3E50')

canvas.create_window((0, 0), window=scroll_frame, anchor='nw')

def _on_config(event):
    canvas.configure(scrollregion=canvas.bbox('all'))
scroll_frame.bind('<Configure>', _on_config)

canvas.configure(yscrollcommand=scrollbar.set)
canvas.pack(side='left', fill='both', expand=True)
scrollbar.pack(side='right', fill='y')

# Column headers

header_row = tk.Frame(scroll_frame, bg='#2C3E50')
header_row.pack(fill='x', pady=(0, 6))

tk.Label(header_row, text='Rank', width=6, anchor='w', bg='#2C3E50', fg='#ECF0F1', font=('Arial', 11,
'bold')).pack(side='left')

tk.Label(header_row, text='Name(s)', anchor='w', bg='#2C3E50', fg='#ECF0F1', font=('Arial', 11,
'bold')).pack(side='left', fill='x', expand=True, padx=(6, 0))

tk.Label(header_row, text='Score', width=8, anchor='e', bg='#2C3E50', fg='#ECF0F1', font=('Arial', 11,
'bold')).pack(side='right')

def _show(diff):

```

```

# update toggle button styles
for b in btn_frame.winfo_children():
    if getattr(b, 'diff', None) == diff:
        b.config(bg='#8E44AD', fg='white')
    else:
        b.config(bg='#34495E', fg='white')

# Clear existing rows
for w in scroll_frame.winfo_children():
    if w is header_row:
        continue
    w.destroy()

entries = load_leaderboard(diff)
if not entries:
    tk.Label(scroll_frame, text=f"No scores yet for {diff}", bg='#2C3E50', fg='white').pack(pady=12)
    return

# Group by score
grouped = {}
for name, score in entries:
    grouped.setdefault(score, []).append(name)

rank = 1
row_bg_toggle = False
for score in sorted(grouped.keys(), reverse=True):
    names = grouped[score]
    names_text = ', '.join(names)

```

```

row_bg = '#2C3E50' if row_bg_toggle else '#22313A'
row_bg_toggle = not row_bg_toggle

row = tk.Frame(scroll_frame, bg=row_bg, pady=6)
row.pack(fill='x', padx=2, pady=2)

# Rank with trophy for first place
rank_text = f"{rank}."
if rank == 1:
    rank_text = "🏆 1"

tk.Label(row, text=rank_text, width=6, anchor='w', bg=row_bg, fg='#F39C12', font=('Arial', 11,
'bold')).pack(side='left')

# Names (wrap long lists)
name_lbl = tk.Label(row, text=names_text, anchor='w', justify='left', bg=row_bg, fg='white',
font=('Arial', 11), wraplength=300)
name_lbl.pack(side='left', fill='x', expand=True, padx=(6, 0))

# Score
tk.Label(row, text=str(score), width=8, anchor='e', bg=row_bg, fg='#F39C12', font=('Arial', 11,
'bold')).pack(side='right', padx=(0, 6))

rank += len(names)

# Initial show
to_show = difficulty or 'Easy'
selected_var.set(to_show)
_show(to_show)

# Bottom actions

```

```
footer = tk.Frame(win, bg='#2C3E50')

footer.pack(fill='x', pady=(0, 12))

tk.Button(footer, text='Refresh', command=lambda: _show(selected_var.get()), bg='#2980B9',
fg='white', font=('Arial', 11, 'bold')).pack(side='left', padx=12)

tk.Button(footer, text='Close', command=win.destroy, bg='#C0392B', fg='white', font=('Arial', 11,
'bold')).pack(side='right', padx=12)
```

```
# -----
# END LEADERBOARD
# -----
```

```
class SoundManager:

    def __init__(self):

        self.sounds = {}

        if not AUDIO_ENABLED:

            return

        for name, filename in AUDIO_FILES.items():

            if name != 'bgm':

                if os.path.exists(filename):

                    try:

                        self.sounds[name] = pygame.mixer.Sound(filename)

                        self.sounds[name].set_volume(0.4)

                    except Exception as e:

                        print(f"Error loading {filename}: {e}")

        def play(self, name):

            if AUDIO_ENABLED and name in self.sounds:

                self.sounds[name].play()
```

```

def start_music(self):
    if AUDIO_ENABLED and os.path.exists(AUDIO_FILES['bgm']):
        try:
            pygame.mixer.music.load(AUDIO_FILES['bgm'])
            pygame.mixer.music.set_volume(0.2)
            pygame.mixer.music.play(-1)
        except Exception as e:
            print(f"Error loading music: {e}")

```

```

def stop_music(self):
    if AUDIO_ENABLED:
        pygame.mixer.music.stop()

```

class Tetromino:

```

def __init__(self, shape_type=None, is_bomb=False):
    if shape_type is None:
        shape_type = random.choice(list(SHAPES.keys()))
    self.shape_type = shape_type
    self.shape = [row[:] for row in SHAPES[shape_type]]
    self.is_bomb = is_bomb
    self.color = COLORS['BOMB'] if is_bomb else COLORS[shape_type]
    self.x = BOARD_WIDTH // 2 - len(self.shape[0]) // 2
    self.y = 0

```

```

def rotate(self):
    self.shape = [[self.shape[j][i] for j in range(len(self.shape) - 1, -1, -1)]
                  for i in range(len(self.shape[0]))]

```

class TetrisGame:

```

def __init__(self, root, difficulty, player_name=None):
    self.root = root

    self.difficulty = difficulty

    # store and normalize player name early so we can autosave when game ends
    self.player_name = (player_name.strip() if isinstance(player_name, str) else None) or "Anonymous"
    self.score_saved = False

    self.settings = DIFFICULTY_CONFIG[difficulty]

    self.root.title(f"Tetris Math - {difficulty} Mode — Player: {self.player_name}")

    # instance-level block size that will be updated on resize
    self.block_size = BLOCK_SIZE

    self.sound = SoundManager()
    self.sound.start_music()

    self.board = [[None for _ in range(BOARD_WIDTH)] for _ in range(BOARD_HEIGHT)]
    self.current_piece = None
    self.next_piece = Tetromino()
    self.score = 0
    self.lines_cleared = 0
    self.level = 1
    self.game_over = False
    self.paused = False
    self.blocks_placed = 0

    self.setup_ui()
    self.new_piece()
    self.update_game()

    # Ensure the outer window size matches the content (avoids unusable void areas)

```

```

# Run after a short delay so geometry managers finish updating
self.root.after(50, self.adjust_window_to_content)

def setup_ui(self):
    """Create a polished in-game UI: game canvas + stylized info panel with stats, preview, and
    controls."""

    self.main_frame = tk.Frame(self.root, bg='#0f1620')
    self.main_frame.pack(fill='both', expand=True)

    # Game canvas (left) — put inside a holder frame so we can center it when window expands
    canvas_holder = tk.Frame(self.main_frame, bg='#0f1620')
    canvas_holder.grid(row=0, column=0, sticky='nsew', padx=(10, 14), pady=12)

    self.canvas = tk.Canvas(canvas_holder, width=self.block_size * BOARD_WIDTH,
                             height=self.block_size * BOARD_HEIGHT,
                             bg='#08141a', highlightthickness=0)

    # Pack centered inside holder so it remains centered when the window is wide
    self.canvas.pack(expand=True, anchor='center')

    # Info panel (right) — stylized card
    info_frame = tk.Frame(self.main_frame, bg='#0b1620', width=INFO_PANEL_WIDTH, bd=0)
    info_frame.grid(row=0, column=1, sticky='nsew', padx=(0, 12), pady=12)

    # Layout weights
    self.main_frame.grid_rowconfigure(0, weight=1, minsize=self.block_size * BOARD_HEIGHT)
    self.main_frame.grid_columnconfigure(0, weight=1)
    self.main_frame.grid_columnconfigure(1, weight=0)

    # Responsive resize
    self.root.bind('<Configure>', self.on_root_resize)

```

```

# Header: difficulty + player

header = tk.Frame(info_frame, bg='#0b1620')

header.pack(fill='x', pady=(4, 10))

tk.Label(header, text=f"{self.difficulty} MODE", font=('Segoe UI', 12, 'bold'), bg='#0b1620',
fg='#F8C471').pack(anchor='w')

tk.Label(header, text=f"Player: {self.player_name}", font=('Segoe UI', 10), bg='#0b1620',
fg='#95A5A6').pack(anchor='w')


# Stats cards

stats_frame = tk.Frame(info_frame, bg='#0b1620')

stats_frame.pack(fill='x')


def stat_card(parent, title, color, value_text):

    card = tk.Frame(parent, bg='#08141a', bd=0, relief='ridge', padx=8, pady=8)

    card.pack(fill='x', pady=6)

    top = tk.Frame(card, bg=color, height=4)

    top.pack(fill='x', side='top', pady=(0, 6))

    tk.Label(card, text=title, bg=card.cget('bg'), fg='#BDC3C7', font=('Segoe UI', 9)).pack(anchor='w')

    val = tk.Label(card, text=value_text, bg=card.cget('bg'), fg='white', font=('Segoe UI', 18, 'bold'))

    val.pack(anchor='w')

    return card, val


_, self.score_label = stat_card(stats_frame, 'SCORE', '#F39C12', '0')

_, self.lines_label = stat_card(stats_frame, 'LINES', '#3498DB', '0')

level_card, self.level_label = stat_card(stats_frame, 'LEVEL', '#E74C3C', '1')


# Level progress

prog_frame = tk.Frame(level_card, bg=level_card.cget('bg'))

```

```

prog_frame.pack(fill='x', pady=(8, 0))

tk.Label(prog_frame, text='Progress', bg=level_card.cget('bg'), fg='#BDC3C7', font=('Segoe UI',
8)).pack(anchor='w')

# Styled progress bar and numeric label

style = ttk.Style()

try:
    style.theme_use('default')
except Exception:
    pass

style.configure('Level.Horizontal.TProgressbar', troughcolor='#17202A', background='#F39C12',
bordercolor='#17202A', lightcolor='#F39C12', darkcolor='#D68910')

bar_row = tk.Frame(prog_frame, bg=prog_frame.cget('bg'))

bar_row.pack(fill='x', pady=(4, 0))

self.level_progress = ttk.Progressbar(bar_row, style='Level.Horizontal.TProgressbar',
orient='horizontal', mode='determinate', maximum=10)

self.level_progress.pack(side='left', fill='x', expand=True)

self.progress_label = tk.Label(bar_row, text='0/10', bg=prog_frame.cget('bg'), fg='#95A5A6',
font=('Segoe UI', 9))

self.progress_label.pack(side='right', padx=(8, 0))


# Next piece preview

nxt = tk.Frame(info_frame, bg='#0b1620')

nxt.pack(fill='x', pady=(12, 6))

tk.Label(nxt, text='NEXT', bg='#0b1620', fg='#BDC3C7', font=('Segoe UI', 10,
'bold')).pack(anchor='w')

self.next_canvas = tk.Canvas(nxt, width=self.block_size * 5, height=self.block_size * 5,
bg='#08141a', highlightthickness=1, highlightbackground='#22313A')

self.next_canvas.pack(pady=(6, 0))


# Controls

ctrl = tk.Frame(info_frame, bg='#0b1620')

```

```

ctrl.pack(fill='x', pady=(12, 6))

self.pause_btn = tk.Button(ctrl, text='Pause', bg='#34495E', fg='white', font=('Segoe UI', 10, 'bold'),
relief='flat', command=self.toggle_pause)

self.pause_btn.pack(side='left')

self.music_on = AUDIO_ENABLED

self.music_btn = tk.Button(ctrl, text='Music: On' if self.music_on else 'Music: Off', bg='#34495E',
fg='white', font=('Segoe UI', 10, 'bold'), relief='flat', command=self.toggle_music)

self.music_btn.pack(side='left', padx=8)

tk.Button(ctrl, text='Leaderboards', bg='#8E44AD', fg='white', font=('Segoe UI', 10, 'bold'),
relief='flat', command=lambda: show_leaderboard_window(self.root, self.difficulty)).pack(side='left',
padx=6)

menu_btn = tk.Button(ctrl, text='Menu', bg='#C0392B', fg='white', font=('Segoe UI', 10, 'bold'),
relief='flat', command=self.show_menu_prompt)

menu_btn.pack(side='right')

# Controls legend

legend = tk.Frame(info_frame, bg='#0b1620')

legend.pack(fill='x', pady=(10, 0))

controls = ["← → : Move  Space : Drop", "↓ : Soft Drop  P : Pause", "↑ : Rotate"]

tk.Label(legend, text='CONTROLS', font=('Segoe UI', 9, 'bold'), bg='#0b1620',
fg='#BDC3C7').pack(anchor='w')

for control in controls:

    tk.Label(legend, text=control, font=('Segoe UI', 9), bg='#0b1620', fg='#95A5A6').pack(anchor='w')

# Key bindings

self.root.bind('<Left>', lambda e: self.move_piece(-1, 0))

self.root.bind('<Right>', lambda e: self.move_piece(1, 0))

```

```

self.root.bind('<Down>', lambda e: self.move_piece(0, 1))

self.root.bind('<Up>', lambda e: self.rotate_piece())

self.root.bind('<space>', lambda e: self.hard_drop())

self.root.bind('p', lambda e: self.toggle_pause())

self.root.bind('P', lambda e: self.toggle_pause())


# small initial info update
self.update_info_panel()


def adjust_window_to_content(self):
    """Resize the root window so the game canvas and info panel fit comfortably.
    This helps avoid clipped UI elements after entering a match."""
    try:
        desired_w = self.block_size * BOARD_WIDTH + INFO_PANEL_WIDTH + 60

        desired_h = max(300, self.block_size * BOARD_HEIGHT + WINDOW_EXTRA_HEIGHT + 80 -
GAME_WINDOW_HEIGHT_REDUCTION)

        # Apply geometry safely
        try:
            self.root.geometry(f"{desired_w}x{desired_h}")
        except Exception:
            pass

        try:
            self.root.minsize(480, 360)
        except Exception:
            pass

    except Exception as e:
        print(f"adjust_window_to_content error: {e}")


def new_piece(self):

```

```

self.current_piece = self.next_piece

chance = self.settings['bomb_chance']

if random.randint(1, chance) == 1:
    self.next_piece = Tetromino(is_bomb=True)
else:
    self.next_piece = Tetromino()

self.draw_next_piece()


if self.check_collision(self.current_piece, 0, 0):
    self.handle_game_over()


def toggle_music(self):
    """Toggle background music on/off and update the button text."""
    if not AUDIO_ENABLED:
        messagebox.showinfo('Audio', 'Audio not available. Install pygame to enable audio.')
        return
    self.music_on = not self.music_on
    if self.music_on:
        self.sound.start_music()
        self.music_btn.config(text='Music: On')
    else:
        self.sound.stop_music()
        self.music_btn.config(text='Music: Off')


def update_info_panel(self):
    """Refresh score, lines, level and progress bar visuals."""
    try:
        self.score_label.config(text=str(int(self.score)))
        self.lines_label.config(text=str(self.lines_cleared))

```

```

self.level_label.config(text=str(self.level))

# progress toward next level (10 lines per level)
progress = self.lines_cleared % 10
self.level_progress['value'] = progress
self.progress_label.config(text=f"{progress}/10")

# force visual refresh so users see immediate change
try:
    self.level_progress.update_idletasks()
    self.level_progress.update()
except Exception:
    pass
except Exception:
    pass

def handle_game_over(self):
    self.game_over = True
    if hasattr(self, 'after_id'):
        self.root.after_cancel(self.after_id)

    # Auto-save score to the appropriate leaderboard immediately
    try:
        if not getattr(self, 'score_saved', False):
            append_score_to_leaderboard(self.difficulty, getattr(self, 'player_name', 'Anonymous') or
            'Anonymous', self.score)

            self.score_saved = True
    except Exception as e:
        print(f"Failed to save leaderboard entry: {e}")

    self.sound.stop_music()
    self.sound.play('gameover')
    self.root.unbind('<Left>')

```

```

self.root.unbind('<Right>')

self.root.unbind('<Down>')

self.root.unbind('<Up>')

self.root.unbind('<space>')

self.main_frame.destroy()

self.show_game_over_screen()

def show_game_over_screen(self):

    go_frame = tk.Frame(self.root, bg='#2C3E50')

    go_frame.pack(fill='both', expand=True)

    tk.Label(go_frame, text="GAME OVER", font=('Arial', 30, 'bold'), bg='#2C3E50',
fg='#E74C3C').pack(pady=(20, 10))

    tk.Label(go_frame, text="Final Score", font=('Arial', 16), bg='#2C3E50', fg='#BDC3C7').pack()

    tk.Label(go_frame, text=f"{int(self.score)}", font=('Arial', 40, 'bold'), bg='#2C3E50',
fg='#F39C12').pack(
        pady=(0, 20))

    tk.Label(go_frame, text=f"Score saved as {getattr(self, 'player_name', 'Anonymous')}", font=('Arial',
12), bg='#2C3E50', fg='#BDC3C7').pack(pady=(0, 10))

    tk.Button(go_frame, text="View Leaderboard", command=lambda:
show_leaderboard_window(self.root, self.difficulty), font=('Arial', 12, 'bold'), bg='#8E44AD', fg='white',
width=20).pack(pady=(0, 6))

    tk.Button(go_frame, text="MAIN MENU", font=('Arial', 14, 'bold'), bg='#27AE60', fg='white',
width=20, height=2, command=lambda: self.return_to_menu(go_frame)).pack(pady=(10, 0))

def return_to_menu(self, frame_to_destroy):

    frame_to_destroy.destroy()

    # Resize window back to menu size and lock resizing to menu layout

    self.root.geometry(f"{MENU_WINDOW_WIDTH}x{MENU_WINDOW_HEIGHT}")

```

```

self.root.resizable(False, False)

MainMenu(self.root)


def show_menu_prompt(self):
    """Show a small, non-blocking menu prompt allowing the player to resume or return to main
    menu."""

    # Pause the game while the prompt is active; remember previous paused state
    prev_paused = self.paused
    self.paused = True

    # Stop music temporarily without changing saved state
    try:
        self.sound.stop_music()
    except Exception:
        pass

    prompt = tk.Toplevel(self.root)
    prompt.title("Menu")
    prompt.transient(self.root)
    prompt.attributes('-topmost', True)
    prompt.geometry("300x120")
    tk.Label(prompt, text="Return to main menu?", font=('Arial', 12, 'bold')).pack(pady=(12, 6))


def resume():
    prompt.destroy()
    self.paused = prev_paused
    if getattr(self, 'music_on', False):
        try:
            self.sound.start_music()
        except Exception:

```

```

pass

self.canvas.delete('pause')


def to_menu():

    prompt.destroy()

    self.return_to_menu(self.main_frame)


btn_frame = tk.Frame(prompt)

btn_frame.pack(pady=(6, 8))

tk.Button(btn_frame, text='Resume Game', bg='#27AE60', fg='white', font=('Arial', 10, 'bold'),
command=resume).pack(side='left', padx=8)

tk.Button(btn_frame, text='Return to Menu', bg='#C0392B', fg='white', font=('Arial', 10, 'bold'),
command=to_menu).pack(side='left', padx=8)

# If the player closes the prompt window, behave like Resume

prompt.protocol('WM_DELETE_WINDOW', resume)


def check_collision(self, piece, offset_x, offset_y):

    for y, row in enumerate(piece.shape):

        for x, cell in enumerate(row):

            if cell:

                new_x = piece.x + x + offset_x

                new_y = piece.y + y + offset_y

                if new_x < 0 or new_x >= BOARD_WIDTH or new_y >= BOARD_HEIGHT:

                    return True

                if new_y >= 0:

                    if self.board[new_y][new_x]:

                        return True

    return False

```

```

def move_piece(self, dx, dy):
    if self.game_over or self.paused: return

    if not self.check_collision(self.current_piece, dx, dy):
        self.current_piece.x += dx
        self.current_piece.y += dy
        if dx != 0 or dy != 0:
            self.sound.play('move')
        self.draw_board()
        return True

    elif dy > 0:
        self.lock_piece()

    return False


def rotate_piece(self):
    if self.game_over or self.paused: return

    original_shape = [row[:] for row in self.current_piece.shape]
    self.current_piece.rotate()

    if self.check_collision(self.current_piece, 0, 0):
        self.current_piece.shape = original_shape
    else:
        self.sound.play('rotate')
        self.draw_board()


def hard_drop(self):
    if self.game_over or self.paused: return

    drop_distance = 0

    while not self.check_collision(self.current_piece, 0, 1):
        self.current_piece.y += 1
        drop_distance += 1

```

```
self.score += drop_distance * 2 * self.settings['score_mult']

self.score_label.config(text=str(int(self.score)))

self.update_info_panel()

self.sound.play('drop')

self.lock_piece()
```

```
def lock_piece(self):

    bomb_positions = []

    for y, row in enumerate(self.current_piece.shape):

        for x, cell in enumerate(row):

            if cell:

                board_y = self.current_piece.y + y

                board_x = self.current_piece.x + x

                if board_y >= 0:

                    if self.current_piece.is_bomb:

                        self.board[board_y][board_x] = 'BOMB'

                        bomb_positions.append((board_y, board_x))

                    else:

                        self.board[board_y][board_x] = self.current_piece.color
```

```
if bomb_positions:

    self.explode_bomb(bomb_positions)
```

```
self.score += 10 * self.settings['score_mult']

self.score_label.config(text=str(int(self.score)))

self.blocks_placed += 1

self.clear_lines()
```

```
if self.blocks_placed >= 5:
```

```

        self.blocks_placed = 0

        self.paused = True

        self.show_math_quiz()
    else:
        self.new_piece()

def explode_bomb(self, bomb_positions):
    self.sound.play('explode')

    for bomb_y, bomb_x in bomb_positions:
        adjacent = [
            (bomb_y - 1, bomb_x), (bomb_y + 1, bomb_x), (bomb_y, bomb_x - 1), (bomb_y, bomb_x + 1),
            (bomb_y - 1, bomb_x - 1), (bomb_y - 1, bomb_x + 1), (bomb_y + 1, bomb_x - 1), (bomb_y + 1,
bomb_x + 1)
        ]

        self.board[bomb_y][bomb_x] = None

        for adj_y, adj_x in adjacent:
            if 0 <= adj_y < BOARD_HEIGHT and 0 <= adj_x < BOARD_WIDTH:
                if self.board[adj_y][adj_x]:
                    self.board[adj_y][adj_x] = None

                    self.score += 50 * self.settings['score_mult']

        self.score_label.config(text=str(int(self.score)))

    self.update_info_panel()

    self.apply_gravity()

def apply_gravity(self):
    changed = True

    while changed:
        changed = False

        for y in range(BOARD_HEIGHT - 2, -1, -1):

```

```
for x in range(BOARD_WIDTH):
    if self.board[y][x] and not self.board[y + 1][x]:
        self.board[y + 1][x] = self.board[y][x]
        self.board[y][x] = None
        changed = True
```

```
def clear_lines(self):
    lines_cleared = 0
    y = BOARD_HEIGHT - 1
    while y >= 0:
        if all(self.board[y]):
            del self.board[y]
            self.board.insert(0, [None for _ in range(BOARD_WIDTH)])
            lines_cleared += 1
        else:
            y -= 1
```

```
if lines_cleared > 0:
    base_points = [0, 100, 300, 500, 800]
    self.score += base_points[lines_cleared] * self.level * self.settings['score_mult']
    self.lines_cleared += lines_cleared
    self.level = self.lines_cleared // 10 + 1
    self.score_label.config(text=str(int(self.score)))
    self.lines_label.config(text=str(self.lines_cleared))
    self.level_label.config(text=str(self.level))
    self.update_info_panel()
```

```
def show_math_quiz(self):
    math_mode = self.settings['math_type']
```

```

# --- LOGIC MATEMATIK ---

if math_mode == 'add_sub':
    a = random.randint(1, 20 + (self.level * 2))
    b = random.randint(1, 20 + (self.level * 2))
    op = random.choice(['+', '-'])
    answer = a + b if op == '+' else a - b
    question = f"{a} {op} {b}"
elif math_mode == 'mult':
    limit = 9 + (self.level // 2)
    a = random.randint(2, limit)
    b = random.randint(2, limit)
    answer = a * b
    question = f"{a} × {b}"
else: # HARD MODE
    a = random.randint(5, 15 + self.level)
    b = random.randint(2, 10)
    c = random.randint(1, 10)
    op1 = random.choice(['+', '-', 'x'])
    op2 = random.choice(['+', '-'])

    # Kira langkah demi langkah
    if op1 == '+':
        temp = a + b
    elif op1 == '-':
        temp = a - b
    else: # Darab
        temp = a * b

```

```

if op2 == '+':
    answer = temp + c
else: # Tolak
    answer = temp - c

question = f"({a} {op1} {b}) {op2} {c}"

# --- QUIZ WINDOW (non-modal, unobtrusive) ---
quiz_window = tk.Toplevel(self.root)
quiz_window.title("Math Quiz")

# small size and positioned near the top-right of the game canvas
try:
    self.root.update_idletasks()

    cx = self.root.winfo_rootx() + self.canvas.winfo_x() + max(0, self.canvas.winfo_width() - 300)
    cy = self.root.winfo_rooty() + self.canvas.winfo_y() + 20
    quiz_window.geometry(f"280x150+{cx}+{cy}")
except Exception:
    quiz_window.geometry(f"{BLOCK_SIZE * 6}x{BLOCK_SIZE * 4}")

quiz_window.configure(bg='#2C3E50')
quiz_window.transient(self.root)
quiz_window.attributes('-topmost', True)

# allow it to be closed; treat close as wrong answer
quiz_window.protocol("WM_DELETE_WINDOW", lambda: [quiz_window.destroy(),
self.handle_quiz_result(False, answer)])

self.time_left = 5 if self.difficulty == 'Hard' else 8
self.timer_running = True

```

```

def update_timer():
    if not self.timer_running: return

    # If the window was closed externally, do nothing
    if not quiz_window.winfo_exists():
        self.timer_running = False
        return

    self.time_left -= 1

    if self.time_left <= 0:
        if quiz_window.winfo_exists():
            quiz_window.destroy()

            self.timer_running = False

            self.handle_quiz_result(False, answer)
        else:
            timer_label.config(text=f"Time: {self.time_left}s", fg='#E74C3C' if self.time_left < 3 else
'#F39C12')

            self.root.after(1000, update_timer)

    timer_label = tk.Label(quiz_window, text=f"Time: {self.time_left}s", font=('Arial', 12, 'bold'),
bg='#2C3E50', fg='#F39C12')

    timer_label.pack(pady=(8, 4))

    tk.Label(quiz_window, text="Solve:", font=('Arial', 11), bg='#2C3E50', fg='#BDC3C7').pack()

    tk.Label(quiz_window, text=question + " = ?", font=('Arial', 16, 'bold'), bg='#2C3E50',
fg='#ECF0F1').pack(pady=6)

    answer_var = tk.StringVar()

    entry = tk.Entry(quiz_window, textvariable=answer_var, font=('Arial', 14), justify='center', width=8)

    entry.pack(pady=(4, 8))

    entry.focus()

```

```

def check_answer():
    self.timer_running = False

    try:
        user_answer = int(answer_var.get())

        if quiz_window.winfo_exists():
            quiz_window.destroy()

        if user_answer == answer:
            self.sound.play('correct')

            self.handle_quiz_result(True, 200 * self.level * self.settings['score_mult'])
        else:
            self.sound.play('wrong')

            self.handle_quiz_result(False, answer)
    except ValueError:
        self.timer_running = True

        messagebox.showwarning("Invalid", "Please enter a number!")

        entry.focus()

tk.Button(quiz_window, text="Submit", command=check_answer, font=('Arial', 11, 'bold'),
bg='#27AE60', fg='white', padx=12).pack(pady=6)

entry.bind('<Return>', lambda e: check_answer())

update_timer()

def handle_quiz_result(self, correct, value):
    color = '#27AE60' if correct else '#E74C3C'

    title = "Correct! 🎉" if correct else "Wrong ❌ "

    # Kalau Salah/Timeout, 'value' sekarang adalah jawaban sebenar, bukan 0.

    msg = f"Bonus: +{int(value)} points" if correct else f"Correct answer was {value}\nPenalty: -500 points"

    if correct:

```

```

        self.score += value
    else:
        self.score = max(0, self.score - 500)
    self.score_label.config(text=str(int(self.score)))
    self.update_info_panel()

# Non-modal, unobtrusive result popup positioned near the canvas
result_window = tk.Toplevel(self.root)
result_window.overrideredirect(False)
result_window.configure(bg=color)
try:
    self.root.update_idletasks()
    cx = self.root.winfo_rootx() + self.canvas.winfo_x() + max(0, self.canvas.winfo_width() - 220)
    cy = self.root.winfo_rooty() + self.canvas.winfo_y() + 20
    result_window.geometry(f"220x90+{cx}+{cy}")
except Exception:
    result_window.geometry("300x150")
result_window.attributes('-topmost', True)

tk.Label(result_window, text=title, font=('Arial', 14, 'bold'), bg=color, fg='white').pack(pady=(10, 4))
tk.Label(result_window, text=msg, font=('Arial', 11), bg=color, fg='white').pack()

# Destroy after short time; resume game immediately so popup doesn't block gameplay
result_window.after(1800, result_window.destroy)

self.paused = False
self.canvas.delete('pause')
self.new_piece()
def toggle_pause(self):
    if not self.game_over:

```

```

        self.paused = not self.paused

        if self.paused:

            self.canvas.create_text(self.block_size * BOARD_WIDTH // 2, self.block_size * BOARD_HEIGHT
// 2, text="PAUSED",

                                   font=('Arial', 30, 'bold'), fill='#F39C12', tags='pause')

            self.sound.stop_music()

        else:

            self.canvas.delete('pause')

            self.sound.start_music()

def draw_board(self):

    self.canvas.delete('all')

    for y in range(BOARD_HEIGHT):

        for x in range(BOARD_WIDTH):

            if self.board[y][x]:

                self.draw_block(x, y, self.board[y][x])

    if self.current_piece:

        for y, row in enumerate(self.current_piece.shape):

            for x, cell in enumerate(row):

                if cell:

                    self.draw_block(self.current_piece.x + x, self.current_piece.y + y, self.current_piece.color,

                                    is_bomb=self.current_piece.is_bomb)

    for x in range(BOARD_WIDTH + 1):

        self.canvas.create_line(x * self.block_size, 0, x * self.block_size, BOARD_HEIGHT * self.block_size,

fill='#2C3E50',

                               width=1)

    for y in range(BOARD_HEIGHT + 1):

        self.canvas.create_line(0, y * self.block_size, BOARD_WIDTH * self.block_size, y * self.block_size,

fill='#2C3E50',

```

width=1)

```
def draw_block(self, x, y, color, is_bomb=False):
    if y >= 0:
        is_bomb_block = (color == 'BOMB') or is_bomb
        fill_color = COLORS['BOMB'] if is_bomb_block else color
        self.canvas.create_rectangle(x * self.block_size + 1, y * self.block_size + 1, (x + 1) * self.block_size
- 1,
                                   (y + 1) * self.block_size - 1, fill=fill_color, outline='#2C3E50', width=2)
        if is_bomb_block:
            font_size = max(8, self.block_size // 2)
            self.canvas.create_text(x * self.block_size + self.block_size // 2, y * self.block_size +
self.block_size // 2, text="💣",
                                   font=('Arial', font_size))

def draw_next_piece(self):
    # Draw the next piece centered and scaled to fit inside the preview canvas.
    self.next_canvas.delete('all')
    canvas_w = int(self.next_canvas.cget('width'))
    canvas_h = int(self.next_canvas.cget('height'))
    shape = self.next_piece.shape
    shape_h = len(shape)
    shape_w = len(shape[0])

    # Choose a preview block size that fits the canvas; cap at current block_size so it scales down if
needed
    preview_block = min(self.block_size, max(8, canvas_w // shape_w, canvas_h // shape_h))
    # If using very small canvases, ensure preview_block is at most the available floor division
    preview_block = min(preview_block, canvas_w // shape_w, canvas_h // shape_h)
```

```
offset_x = (canvas_w - shape_w * preview_block) // 2
```

```
offset_y = (canvas_h - shape_h * preview_block) // 2
```

```
for y, row in enumerate(shape):
```

```
    for x, cell in enumerate(row):
```

```
        if cell:
```

```
            color = COLORS['BOMB'] if self.next_piece.is_bomb else self.next_piece.color
```

```
            x1 = offset_x + x * preview_block
```

```
            y1 = offset_y + y * preview_block
```

```
            x2 = x1 + preview_block
```

```
            y2 = y1 + preview_block
```

```
            self.next_canvas.create_rectangle(x1 + 1, y1 + 1, x2 - 1, y2 - 1, fill=color,  
                                              outline='#2C3E50', width=2)
```

```
            if self.next_piece.is_bomb:
```

```
                font_size = max(8, preview_block // 2)
```

```
                self.next_canvas.create_text((x1 + x2) // 2, (y1 + y2) // 2, text="💣",  
                                              font=('Arial', font_size))
```

```
def update_game(self):
```

```
    if self.game_over: return # Stop update loop if game is over
```

```
    if not self.paused:
```

```
        self.move_piece(0, 1)
```

```
        self.draw_board()
```

```
    base_speed = self.settings['speed']
```

```
    speed = max(100, base_speed - (self.level - 1) * 40)
```

```
    self.root.after(speed, self.update_game)
```

```

def on_root_resize(self, event):

    # Recompute block size when the main window resizes so the game scales appropriately

    try:

        w = max(100, self.main_frame.winfo_width())

        h = max(100, self.main_frame.winfo_height())

    except Exception:

        return

    # Reserve space for the info panel width and a small gap

    available_w = max(100, w - INFO_PANEL_WIDTH - 10)

    available_h = max(100, h - 10)

    new_block = max(8, min(available_w // BOARD_WIDTH, available_h // BOARD_HEIGHT))

    if new_block != self.block_size:

        self.block_size = new_block

        # update canvas and preview sizes

        self.canvas.config(width=self.block_size * BOARD_WIDTH, height=self.block_size *
BOARD_HEIGHT)

        self.next_canvas.config(width=self.block_size * 4, height=self.block_size * 4)

        # redraw to reflect new sizes

        self.draw_board()

        self.draw_next_piece()

class MainMenu:

    def __init__(self, root):

        self.root = root

        self.root.title("CalcuTris")

        # Clear any existing widgets in root (important for return from game over)

```

```

for widget in self.root.winfo_children():
    widget.destroy()

self.frame = tk.Frame(root, bg='#2C3E50')
self.frame.pack(fill='both', expand=True)

bg_path = r"C:\Users\User\Downloads\AUDIO GAME\gambartetris.jpg"

# Always create a background label to hold an image or video frames behind widgets
self.bg_label = tk.Label(self.frame, bd=0)
self.bg_label.place(relx=0, rely=0, relwidth=1, relheight=1)
self.bg_label.lower()

if os.path.exists(bg_path):
    # If a video file and OpenCV + Pillow are available, play the video; otherwise fall back to an image
    video_exts = ('.mp4', '.avi', '.mov', '.mkv', '.webm')
    if OPENCV_AVAILABLE and PIL_AVAILABLE and bg_path.lower().endswith(video_exts):
        try:
            self._start_video(bg_path)
        except Exception as e:
            print(f"Failed to play video background: {e}. Falling back to image.")
            # fallback to static image below
            try:
                if PIL_AVAILABLE:
                    # Keep original image and resize it dynamically to the menu frame size
                    img = Image.open(bg_path).convert('RGBA')
                    self._bg_orig = img
                    if hasattr(Image, 'Resampling'):
                        self._bg_resample = Image.Resampling.LANCZOS
                    else:

```

```

        self._bg_resample = getattr(Image, 'LANCZOS', Image.BICUBIC)

    # initial draw

    try:

        self._update_bg_image()

    except Exception:

        # fall back to a single resized image if update fails

        img2 = img.resize((BLOCK_SIZE * BOARD_WIDTH + INFO_PANEL_WIDTH, BLOCK_SIZE *
BOARD_HEIGHT + WINDOW_EXTRA_HEIGHT), self._bg_resample)

        self.bg_image = ImageTk.PhotoImage(img2)

        self.bg_label.config(image=self.bg_image)

        self.bg_label.lower()

    # Ensure the background updates on frame resize

    try:

        self.frame.bind('<Configure>', lambda e: self._update_bg_image())

    except Exception:

        pass

    else:

        self.bg_image = tk.PhotoImage(file=bg_path)

        self.bg_label.config(image=self.bg_image)

        self.bg_label.lower()

    except Exception as e:

        print(f"Failed to load menu background: {e}")

    else:

        try:

            if PIL_AVAILABLE:

                # Keep original image and resize dynamically to the menu frame size

                img = Image.open(bg_path).convert('RGBA')

                self._bg_orig = img

                if hasattr(Image, 'Resampling'):

```

```

        self._bg_resample = Image.Resampling.LANCZOS
    else:
        self._bg_resample = getattr(Image, 'LANCZOS', Image.BICUBIC)

    # initial draw

    try:
        self._update_bg_image()

    except Exception:
        img2 = img.resize((BLOCK_SIZE * BOARD_WIDTH + INFO_PANEL_WIDTH, BLOCK_SIZE *
BOARD_HEIGHT + WINDOW_EXTRA_HEIGHT), self._bg_resample)

        self.bg_image = ImageTk.PhotoImage(img2)
        self.bg_label.config(image=self.bg_image)
        self.bg_label.lower()

    # Ensure the background updates on frame resize

    try:
        self.frame.bind('<Configure>', lambda e: self._update_bg_image())

    except Exception:
        pass

    else:
        self.bg_image = tk.PhotoImage(file=bg_path)

    # Place the image as a label and send it behind other widgets

    self.bg_label.config(image=self.bg_image)
    self.bg_label.lower()

except Exception as e:
    print(f"Failed to load menu background: {e}")

# --- Image-backed labels (polkadot) ---
polka_path = r"C:\Users\User\Downloads\AUDIO GAME\gambartetris.jpg"
self.title_bg = None

```

```

self.subtitle_bg = None

self.info_bg = None

if os.path.exists(polka_path) and PIL_AVAILABLE:
    try:
        polka_img = Image.open(polka_path).convert('RGBA')

        def tiled_photo(w, h):
            tile_w, tile_h = polka_img.size
            canvas_img = Image.new('RGBA', (w, h))
            for x in range(0, w, tile_w):
                for y in range(0, h, tile_h):
                    canvas_img.paste(polka_img, (x, y))
            return ImageTk.PhotoImage(canvas_img)

        self.title_bg = tiled_photo(400, 70)
        self.subtitle_bg = tiled_photo(360, 44)
        self.info_bg = tiled_photo(380, 100)
    except Exception as e:
        print(f"Failed to load polkadot background: {e}")
elif os.path.exists(polka_path):
    try:
        # Try tk.PhotoImage for PNG/GIF; JPEG will usually fail here
        temp = tk.PhotoImage(file=polka_path)
        self.title_bg = temp
        self.subtitle_bg = temp
        self.info_bg = temp
    except Exception:
        pass

```

```

# --- Header (stylized title with subtle shadow) ---

title_container = tk.Frame(self.frame, bg='#2C3E50')

title_container.pack(fill='x', pady=(8, 6))

tk.Label(title_container, text="CalcuTris", font=('Helvetica', 34, 'bold'), bg='#2C3E50',
fg='#FF6B6B').pack()

tk.Label(title_container, text="A math-powered Tetris experience", font=('Arial', 12), bg='#2C3E50',
fg='#BDC3C7').pack()


# --- Difficulty cards area ---

cards_frame = tk.Frame(self.frame, bg='#2C3E50')

cards_frame.pack(pady=(16, 8), padx=12, fill='x')


def make_card(parent, diff, color, desc, emoji):

    card = tk.Frame(parent, bg='#22313A', bd=0, relief='flat', padx=10, pady=10)

    card.pack(side='left', expand=True, fill='both', padx=8)


def on_enter(e):

    card.config(bg='#2C3E50')

def on_leave(e):

    card.config(bg='#22313A')


card.bind('<Enter>', on_enter)

card.bind('<Leave>', on_leave)

card.config(cursor='hand2')

card.bind('<Button-1>', lambda e, d=diff: self.start_game(d))


# header

hdr = tk.Frame(card, bg=color)

```

```

hdr.pack(fill='x', pady=(0, 8))

tk.Label(hdr, text=f"{emoji} {diff.upper()}", bg=color, fg='white', font=('Arial', 12,
'bold')).pack(padx=6, pady=6)

# description

tk.Label(card, text=desc, wraplength=260, justify='left', bg=card.cget('bg'), fg='#ECF0F1',
font=('Arial', 10)).pack(anchor='w')

# details

tk.Label(card, text=f"Speed: {DIFFICULTY_CONFIG[diff]['speed']} ms | Bomb chance:
{DIFFICULTY_CONFIG[diff]['bomb_chance']}%",
        bg=card.cget('bg'), fg='#BDC3C7', font=('Arial', 9)).pack(anchor='w', pady=(6, 0))

# action row

row = tk.Frame(card, bg=card.cget('bg'))

row.pack(fill='x', pady=(10, 0))

play_btn = tk.Button(row, text='Play', bg=color, fg='white', font=('Arial', 11, 'bold'), relief='flat',
command=lambda d=diff: self.start_game(d))

play_btn.pack(side='left')

hover_map = {'Easy': '#2ECC71', 'Normal': '#3498DB', 'Hard': '#E74C3C'}

play_btn.bind('<Enter>', lambda e, b=play_btn: b.config(bg=hover_map.get(diff, color),
relief='raised'))

play_btn.bind('<Leave>', lambda e, b=play_btn: b.config(bg=color, relief='flat'))

help_btn = tk.Button(row, text='How', bg='#34495E', fg='white', font=('Arial', 9), relief='flat',
command=lambda d=diff: messagebox.showinfo(f"How to play {d}", "Use arrows to move, Space to
drop, answer quizzes to gain bonuses."))

help_btn.pack(side='left', padx=8)

return card

```

```
make_card(cards_frame, 'Easy', '#27AE60', 'Addition and subtraction-based math tasks. Gentle pace for learning.', '□')
```

```
make_card(cards_frame, 'Normal', '#2980B9', 'Multiplication-focused problems with a moderate pace.', '●')
```

```
make_card(cards_frame, 'Hard', '#C0392B', 'Mixed operations, faster speed and more bombs. Dare to try?', '💣')
```

```
# --- Lower control row ---
```

```
controls = tk.Frame(self.frame, bg='#2C3E50')
```

```
controls.pack(fill='x', pady=(12, 6), padx=22)
```

```
lb_btn = tk.Button(controls, text='Leaderboards', bg='#8E44AD', fg='white', font=('Arial', 12, 'bold'), relief='flat', command=lambda: show_leaderboard_window(self.root))
```

```
lb_btn.pack(side='left')
```

```
opt_btn = tk.Button(controls, text='Settings', bg='#34495E', fg='white', font=('Arial', 12, 'bold'), relief='flat', command=lambda: messagebox.showinfo('Settings', 'No settings yet — coming soon!'))
```

```
opt_btn.pack(side='left', padx=8)
```

```
quit_btn = tk.Button(controls, text='Quit', bg='#C0392B', fg='white', font=('Arial', 12, 'bold'), relief='flat', command=self.root.quit)
```

```
quit_btn.pack(side='right')
```

```
# Footer note
```

```
tk.Label(self.frame, text='Tip: Answer quizzes during the game to earn bonus points!', font=('Arial', 10), bg='#2C3E50', fg='#BDC3C7').pack(pady=(10, 8))
```

```
def create_btn(self, text, color, command):
```

```
tk.Button(self.frame, text=text, bg=color, fg='white', font=('Arial', 12, 'bold'), width=15, height=2, borderwidth=0, command=command).pack(pady=5)
```

```

def start_game(self, difficulty):

    # Prompt player for a name before starting the game

    prompt = tk.Toplevel(self.root)

    prompt.title("Enter Name")

    prompt.transient(self.root)

    prompt.grab_set()

    prompt.geometry("360x140")

    tk.Label(prompt, text=f"Enter your name for {difficulty} mode:", font=('Arial', 12), bg='#2C3E50',
fg='#BDC3C7').pack(pady=(12, 6))

    name_var = tk.StringVar()

    name_entry = tk.Entry(prompt, textvariable=name_var, font=('Arial', 12))

    name_entry.pack(pady=(0, 8))

    name_entry.focus()


def _start():

    player_name = name_var.get().strip() or "Anonymous"

    try:

        self._stop_video()

    except Exception:

        pass

    # Resize window to game size and lock resizing for consistent gameplay

    game_width = BLOCK_SIZE * BOARD_WIDTH + INFO_PANEL_WIDTH

    game_height = max(300, BLOCK_SIZE * BOARD_HEIGHT + WINDOW_EXTRA_HEIGHT -
GAME_WINDOW_HEIGHT_REDUCTION)

    self.root.geometry(f"{game_width}x{game_height}")

    self.root.resizable(False, False)

    prompt.destroy()

    self.frame.destroy()

    TetrisGame(self.root, difficulty, player_name=player_name)

```

```
tk.Button(prompt, text="Start", command=_start, font=('Arial', 12, 'bold'), bg='#27AE60', fg='white',
width=12).pack(pady=(6, 8))
```

```
name_entry.bind('<Return>', lambda e: _start())
```

```
def _start_video(self, path):
```

```
    # OpenCV-based video playback loop; frames are scaled to the menu frame size.
```

```
    self._video_cap = cv2.VideoCapture(path)
```

```
    if not self._video_cap.isOpened():
```

```
        raise RuntimeError(f"Cannot open video file: {path}")
```

```
    fps = self._video_cap.get(cv2.CAP_PROP_FPS) or 24
```

```
    self._video_delay = int(1000 / fps)
```

```
    self._video_running = True
```

```
    self._video_photo = None
```

```
    # kick off the update loop
```

```
    self._update_video_frame()
```

```
def _update_video_frame(self):
```

```
    if not getattr(self, '_video_running', False):
```

```
        return
```

```
    cap = self._video_cap
```

```
    ret, frame = cap.read()
```

```
    if not ret:
```

```
        cap.set(cv2.CAP_PROP_POS_FRAMES, 0)
```

```
        ret, frame = cap.read()
```

```
        if not ret:
```

```
            return
```

```
    # convert to RGB
```

```
    frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
```

```

# desired size (use the menu frame size)
w = max(1, self.frame.winfo_width())
h = max(1, self.frame.winfo_height())

try:
    img = Image.fromarray(frame).resize((w, h), Image.ANTIALIAS)
except Exception:
    img = Image.fromarray(frame)
photo = ImageTk.PhotoImage(img)
self.bg_label.config(image=photo)
self.bg_label.image = photo

# schedule next
self.root.after(self._video_delay, self._update_video_frame)

def _stop_video(self):
    self._video_running = False
    cap = getattr(self, '_video_cap', None)
    if cap:
        try:
            cap.release()
        except Exception:
            pass
    self._video_cap = None

def _update_bg_image(self, event=None):
    """Resize the original background image to the current frame size and apply it edge-to-edge."""
    try:
        if not hasattr(self, '_bg_orig'):
            return
        w = max(1, self.frame.winfo_width())

```

```
h = max(1, self.frame.winfo_height())
img = self._bg_orig.resize((w, h), self._bg_resample)
self.bg_image = ImageTk.PhotoImage(img)
self.bg_label.config(image=self.bg_image)
self.bg_label.image = self.bg_image
self.bg_label.lower()
except Exception as e:
    print(f"Failed to update background image: {e}")
```

```
def main():
    root = tk.Tk()
    # Make the main window non-resizable for consistent layout
    root.resizable(False, False)
    # Set initial size for the menu
    root.geometry(f"{MENU_WINDOW_WIDTH}x{MENU_WINDOW_HEIGHT}")
    MainMenu(root)
    root.mainloop()

if __name__ == "__main__":
    main()
```