Build & Shake Game

January 2, 2026

[ ]:
```
"""
Build & Shake (MVP)
- Small vibrating tower "game": 1-DOF mass-spring model per floor
- Computes natural frequencies and mode shapes
- Animates the first mode shape (with a torsion indicator)
- Interface built with Matplotlib sliders (no Tkinter / customtkinter required)

Requirements:
  pip install numpy scipy matplotlib

Run:
  python build_and_shake.py
"""
```

[ ]:
```
# ---------------------------- Imports ---------------------------- #
import numpy as np
from dataclasses import dataclass
from scipy.linalg import eigh
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from matplotlib.widgets import Slider, Button, RadioButtons
```

[ ]:
```
# ---------------------------- Model ---------------------------- #
@dataclass
class TowerParams:
    n: int = 20                # nb étages
    m_floor: float = 1.0       # masse par étage (valeur relative)
    k_story: float = 10.0      # rigidité par étage (valeur relative)
    asym: float = 0.0          # asymétrie (0 = symétrique, >0 ajoute␣
 ↪"torsion-like")
    crack: float = 0.0         # "fissuration" (0..0.8) -> réduit K
    damping: float = 0.02      # amortissement visuel (pas physique exact,␣
 ↪juste pour smooth)
    amp: float = 1.0           # amplitude d'animation
```

[ ]:
```
# -------------------- Definitions / Inputs -------------------- #
def build_MK(p: TowerParams):
```

```
    """
    Système shear building:
      M = diag(m_i)
      K = tridiagonale (k_story par étage), diminuée par crack
    On ajoute une "composante torsion-like" via asym -> mélange des déplacements
    (ce n'est pas une vraie torsion 3D, mais un indicateur fun pour le jeu).
    """
    n = p.n
    m = np.full(n, max(1e-6, p.m_floor), dtype=float)
    M = np.diag(m)

    k_eff = max(1e-6, p.k_story) * (1.0 - np.clip(p.crack, 0.0, 0.95))
    k = np.full(n, k_eff, dtype=float)

    K = np.zeros((n, n), dtype=float)
    for i in range(n):
        if i == 0:
            K[i, i] += k[i]
        else:
            K[i, i] += k[i] + k[i-1]
            K[i, i-1] -= k[i-1]
            K[i-1, i] -= k[i-1]

    # "torsion-like" coupling matrix (fun)
    # plus asym est grand, plus on mélange les étages voisins -> mode shape␣
↪plus "twisty"
    asym = np.clip(p.asym, 0.0, 1.0)
    if asym > 0:
        C = np.zeros((n, n), dtype=float)
        for i in range(n):
            if i > 0:
                C[i, i-1] += 1.0
            if i < n-1:
                C[i, i+1] += 1.0
            C[i, i] -= ( (i > 0) + (i < n-1) )
        # petite perturbation de K
        K = K + (0.15 * asym) * (k_eff * C)

    return M, K
```

[ ]:
```
def modal_analysis(M, K):
    """
    Résout K phi = w^2 M phi
    Retourne fréquences (Hz-like), modes (normalisés)
    """
    # eigh pour matrices symétriques
    w2, V = eigh(K, M)
```

```
    # filtre numérique
    w2 = np.clip(w2, 0.0, None)
    idx = np.argsort(w2)
    w2 = w2[idx]
    V = V[:, idx]

    # supprimer les 0 (si jamais)
    eps = 1e-10
    valid = w2 > eps
    w2 = w2[valid]
    V = V[:, valid]

    w = np.sqrt(w2)

    # "Hz-like" : ici c'est relatif (pas de vraies unités), on garde f = w/(2pi)
    f = w / (2*np.pi)

    # normalisation mode: max abs = 1
    Vn = V / np.max(np.abs(V), axis=0, keepdims=True)
    return f, Vn
```

[ ]:
```
def compute_scores(p: TowerParams, f, mode1):
    """
    Scores fun:
      - confort: plus la fréquence 1 est basse, plus c'est "mou" -> confort␣
↪souvent pire sous vent
      - torsion-like: basé sur variation de pente du mode (plus ça zigzague,␣
↪plus "torsion")
      - coût fictif: augmente avec k_story + masse
    """
    f1 = f[0] if len(f) else 0.0

    # confort: on préfère une tour ni trop molle ni trop rigide (score en␣
↪cloche)
    # cible arbitraire (fun): f1_target ~ 0.25 (relatif)
    target = 0.25
    comfort = np.exp(-((f1 - target) / 0.18)**2)  # 0..1 environ

    # torsion-like: mesure de "zigzag" = norme de la dérivée seconde discrète
    x = mode1.copy()
    d2 = x[:-2] - 2*x[1:-1] + x[2:]
    twist = float(np.linalg.norm(d2) / max(1e-8, np.linalg.norm(x)))
    twist = np.clip(twist, 0.0, 2.0)

    # coût fictif
    cost = (p.n * (0.6*p.m_floor + 0.4*p.k_story)) * (1.0 + 0.5*p.asym)
```

```
    cost = float(cost)

    # score global (fun)
    # on récompense confort, on pénalise torsion et coût
    score = 100 * comfort - 25 * twist - 0.15 * cost
    return comfort, twist, cost, score
```

[ ]:
```
# ---------------------------- UI / Game ---------------------------- #
def main():
    p = TowerParams()

    # initial compute
    M, K = build_MK(p)
    f, V = modal_analysis(M, K)
    mode_idx = 0
    mode = V[:, mode_idx] if V.size else np.zeros(p.n)

    # figure
    plt.close("all")
    fig = plt.figure(figsize=(11, 6))
    ax = fig.add_axes([0.07, 0.12, 0.55, 0.80])
    ax.set_title("Build & Shake - Mode animation (MVP)")
    ax.set_xlabel("Déplacement latéral (relatif)")
    ax.set_ylabel("Étage")
    ax.grid(True, alpha=0.25)

    y = np.arange(1, p.n+1)

    # draw "tower spine"
    (line,) = ax.plot(np.zeros_like(y), y, lw=3)
    (pts,) = ax.plot(np.zeros_like(y), y, marker="o", lw=0)

    ax.set_ylim(0.5, p.n + 0.5)
    ax.set_xlim(-1.6, 1.6)

    # HUD text
    hud = fig.text(0.66, 0.86, "", fontsize=11, family="monospace")
    hint = fig.text(0.66, 0.20,
                    "But: un bon score = confortable, peu torsion, pas trop␣
 ↪cher.\n"
                    "Astuce: joue avec fissuration + asymétrie  ",
                    fontsize=10)

    # sliders area
    ax_n     = fig.add_axes([0.66, 0.76, 0.30, 0.03])
    ax_m     = fig.add_axes([0.66, 0.71, 0.30, 0.03])
    ax_k     = fig.add_axes([0.66, 0.66, 0.30, 0.03])
```

```
    ax_cr    = fig.add_axes([0.66, 0.61, 0.30, 0.03])
    ax_asym  = fig.add_axes([0.66, 0.56, 0.30, 0.03])
    ax_amp   = fig.add_axes([0.66, 0.51, 0.30, 0.03])

    s_n    = Slider(ax_n,    "Étages",    5, 80, valinit=p.n, valstep=1)
    s_m    = Slider(ax_m,    "Masse",     0.3, 4.0, valinit=p.m_floor)
    s_k    = Slider(ax_k,    "Rigidité",  1.0, 40.0, valinit=p.k_story)
    s_cr   = Slider(ax_cr,   "Fissure",   0.0, 0.85, valinit=p.crack)
    s_asym = Slider(ax_asym, "Asym",      0.0, 1.0, valinit=p.asym)
    s_amp  = Slider(ax_amp,  "Amp",       0.3, 2.5, valinit=p.amp)

    # radio for mode selection
    ax_mode = fig.add_axes([0.66, 0.30, 0.30, 0.16])
    ax_mode.set_title("Mode affiché")
    radio = RadioButtons(ax_mode, ("Mode 1", "Mode 2", "Mode 3"), active=0)

    # buttons
    ax_rand = fig.add_axes([0.66, 0.12, 0.14, 0.06])
    ax_reset = fig.add_axes([0.82, 0.12, 0.14, 0.06])
    b_rand = Button(ax_rand, "Random")
    b_reset = Button(ax_reset, "Reset")

    # animation state
    state = {
        "t": 0.0,
        "f": f,
        "V": V,
        "mode_idx": 0,
        "mode": mode,
        "p": p,
        "scores": (0, 0, 0, 0),
    }

    def recompute():
        # update params from sliders
        p2 = TowerParams(
            n=int(s_n.val),
            m_floor=float(s_m.val),
            k_story=float(s_k.val),
            crack=float(s_cr.val),
            asym=float(s_asym.val),
            amp=float(s_amp.val),
            damping=p.damping
        )
        M2, K2 = build_MK(p2)
        f2, V2 = modal_analysis(M2, K2)
```

```
        # ensure we have enough modes for selection
        mi = state["mode_idx"]
        mi = min(mi, max(0, (V2.shape[1]-1) if V2.size else 0))

        mode2 = V2[:, mi] if V2.size else np.zeros(p2.n)

        # update axes / y
        ax.set_ylim(0.5, p2.n + 0.5)

        # update state
        state["p"] = p2
        state["f"] = f2
        state["V"] = V2
        state["mode_idx"] = mi
        state["mode"] = mode2

        # update y arrays (new length)
        nonlocal y
        y = np.arange(1, p2.n+1)

        # update plot objects data length
        line.set_data(np.zeros_like(y), y)
        pts.set_data(np.zeros_like(y), y)

        # recompute scores on mode1 always
        if V2.size:
            mode1 = V2[:, 0]
            comfort, twist, cost, score = compute_scores(p2, f2, mode1)
        else:
            comfort, twist, cost, score = (0.0, 0.0, 0.0, -999.0)

        state["scores"] = (comfort, twist, cost, score)

    def update_hud():
        p2 = state["p"]
        f2 = state["f"]
        comfort, twist, cost, score = state["scores"]

        ftxt = "n/a"
        if len(f2) >= 1:
            ftxt = f"{f2[0]:.3f} (rel.)"

        hud.set_text(
            f"f1   {ftxt}\n"
            f"Confort: {comfort*100:5.1f}/100\n"
            f"Torsion:  {twist:5.2f} (↓ mieux)\n"
            f"Coût:     {cost:7.1f}\n"
```

```
            f"Score:    {score:7.1f}"
        )

    def on_slider(_):
        recompute()
        update_hud()

    for s in (s_n, s_m, s_k, s_cr, s_asym, s_amp):
        s.on_changed(on_slider)

    def on_mode(label):
        mi = {"Mode 1": 0, "Mode 2": 1, "Mode 3": 2}.get(label, 0)
        V2 = state["V"]
        if V2.size:
            mi = min(mi, V2.shape[1]-1)
            state["mode_idx"] = mi
            state["mode"] = V2[:, mi]
        else:
            state["mode_idx"] = 0
            state["mode"] = np.zeros(state["p"].n)
        update_hud()

    radio.on_clicked(on_mode)

    def on_random(_):
        # random fun settings
        s_n.set_val(int(np.random.randint(8, 60)))
        s_m.set_val(float(np.random.uniform(0.5, 3.0)))
        s_k.set_val(float(np.random.uniform(2.0, 35.0)))
        s_cr.set_val(float(np.random.uniform(0.0, 0.8)))
        s_asym.set_val(float(np.random.uniform(0.0, 1.0)))
        s_amp.set_val(float(np.random.uniform(0.6, 2.2)))

    def on_reset(_):
        s_n.reset()
        s_m.reset()
        s_k.reset()
        s_cr.reset()
        s_asym.reset()
        s_amp.reset()
        radio.set_active(0)

    b_rand.on_clicked(on_random)
    b_reset.on_clicked(on_reset)

    # init
    recompute()
```

```
    update_hud()

    # animation
    def animate(_frame):
        p2 = state["p"]
        mode2 = state["mode"]

        state["t"] += 0.06
        t = state["t"]

        # sinus animation along mode shape
        # damping is just a smoothing factor for "feel"
        A = p2.amp
        x = A * mode2 * np.sin(2*np.pi*0.7*t)

        # normalize to keep in view
        mx = max(1e-6, np.max(np.abs(x)))
        x = x / mx

        # update data
        line.set_data(x, y)
        pts.set_data(x, y)

        # dynamic xlim
        ax.set_xlim(-1.6, 1.6)
        return line, pts

    _anim = FuncAnimation(fig, animate, interval=30, blit=True)

    plt.show()
if __name__ == "__main__":
    main()
```