Photoreal Physics Engine + Harmonic Algebra (HA)

A clean, runnable Blender 4.x project that simulates physics (rigid bodies & cloth) and renders **photoreal** videos with **Cycles**. It's wired to a minimal Harmonic Algebra layer so your resonance fields become Blender force emitters, and a safety operator clamps extremes. No cloud keys needed.

Quick Start

Windows 1. Install **Blender 4.x**. 2. Create a folder (e.g., `PhotorealPhysicsEngine_HA/`). 3. Reproduce the file tree below and paste each file's code into place. 4. Edit `run_blender_windows.bat`: set the path to your `blender.exe`. 5. Double-click `run_blender_windows.bat`. Videos render to `output/`.

macOS / Linux

```
chmod +x run_blender_unix.sh
./run_blender_unix.sh
```

One-liner smoke test (fast):

```
blender -b -P scripts/blender_sim.py -- --scenario rigid --duration 2 --fps 24
--res 1280 720 --samples 32 --out output/smoke_test.mp4
```

Folder Structure

```
PhotorealPhysicsEngine_HA/
├─ README.md
├─ run_blender_windows.bat
├─ run_blender_unix.sh
├─ scripts/
│  └─ blender_sim.py
├─ harmonic/
│  ├─ __init__.py
│  ├─ harmonic_core.py
│  └─ blender_bridge.py
└─ configs/
   ├─ default_rigid.json
   ├─ default_cloth.json
   ├─ only_ha.json
```

```
    ├─ ha_rigid.json
    └─ ha_cloth.json
```

---

scripts/blender_sim.py

```
# blender_sim.py — Blender 4.x
# Run examples:
#   blender -b -P scripts/blender_sim.py -- --config configs/ha_rigid.json
#   blender -b -P scripts/blender_sim.py -- --scenario rigid --duration 6 --fps
24
#           --res 1920 1080 --samples 256 --out output/rigid_demo.mp4

import bpy, sys, os, json, random
from math import radians
from mathutils import Euler

# ------------------ arg parsing ------------------
argv = sys.argv
argv = argv[argv.index("--") + 1:] if "--" in argv else []

def pop_flag(flag, default=None, cast=None, nargs=1):
    if flag in argv:
        i = argv.index(flag)
        vals = argv[i+1:i+1+nargs]
        # remove flag + args
        for _ in range(nargs + 1):
            del argv[i]
        if cast:
            return cast(vals[0]) if nargs == 1 else [cast(v) for v in vals]
        return vals[0] if nargs == 1 else vals
    return default

config_path = pop_flag("--config", None, str)
scenario    = pop_flag("--scenario", "rigid", str)
duration    = pop_flag("--duration", 5.0, float)
fps         = pop_flag("--fps", 24, int)
res_w, res_h= pop_flag("--res", [1920,1080], int, nargs=2)
samples     = pop_flag("--samples", 256, int)
out_path    = pop_flag("--out", "output/shot.mp4", str)
vol_density = pop_flag("--volumetrics", 0.0, float)
seed        = pop_flag("--seed", 42, int)
ha_cfg_path = pop_flag("--ha-config", None, str)

ha_cfg_inline = None
```

```
if config_path and os.path.exists(config_path):
    with open(config_path, "r", encoding="utf-8") as f:
        cfg = json.load(f)
    scenario    = cfg.get("scenario", scenario)
    duration    = cfg.get("duration", duration)
    fps         = cfg.get("fps", fps)
    res_w       = cfg.get("res_w", res_w)
    res_h       = cfg.get("res_h", res_h)
    samples     = cfg.get("samples", samples)
    out_path    = cfg.get("out_path", out_path)
    vol_density = cfg.get("volumetrics", vol_density)
    seed        = cfg.get("seed", seed)
    if "ha" in cfg:
        ha_cfg_inline = cfg["ha"]

if ha_cfg_path and os.path.exists(ha_cfg_path):
    with open(ha_cfg_path, "r", encoding="utf-8") as f:
        ha_file_cfg = json.load(f)
    if "ha" in ha_file_cfg:
        ha_cfg_inline = ha_file_cfg["ha"]

random.seed(seed)

# ------------------ utils ------------------
def safe_clear_scene():
    bpy.ops.wm.read_factory_settings(use_empty=True)


def set_cycles_engine(samples, res_w, res_h, fps):
    scene = bpy.context.scene
    scene.render.engine = 'CYCLES'
    # Try GPU first, fall back to CPU
    try:
        prefs = bpy.context.preferences.addons['cycles'].preferences
        prefs.get_devices()  # ensure devices are enumerated
        for d in prefs.devices:
            d.use = True
        scene.cycles.device = 'GPU'
    except Exception:
        scene.cycles.device = 'CPU'

    scene.cycles.samples = samples
    scene.cycles.use_adaptive_sampling = True
    scene.cycles.max_bounces = 12
    scene.cycles.use_denoising = True
    scene.cycles.blur_glossy = 1.0

    scene.render.resolution_x = res_w
```

```
    scene.render.resolution_y = res_h
    scene.render.fps = fps

    scene.view_settings.view_transform = 'Filmic'
    scene.view_settings.look = 'Medium High Contrast'
    scene.view_settings.exposure = 0.0
    scene.view_settings.gamma = 1.0

    scene.render.use_motion_blur = True
    scene.render.motion_blur_shutter = 0.5

    scene.render.image_settings.file_format = 'FFMPEG'
    scene.render.ffmpeg.format = 'MPEG4'
    scene.render.ffmpeg.codec = 'H264'
    scene.render.ffmpeg.constant_rate_factor = 'HIGH'
    scene.render.ffmpeg.ffmpeg_preset = 'GOOD'
    scene.render.ffmpeg.gopsize = 12


def ensure_parent_dir(path):
    d = os.path.dirname(path)
    if d and not os.path.exists(d):
        os.makedirs(d, exist_ok=True)


def set_output(path, total_frames):
    scene = bpy.context.scene
    scene.frame_start = 1
    scene.frame_end = int(total_frames)
    scene.render.filepath = bpy.path.abspath("//" + path)
    ensure_parent_dir(scene.render.filepath)


def add_world(sky_strength=1.0, volumetric_density=0.0):
    world = bpy.data.worlds.new("World") if not bpy.data.worlds else
bpy.data.worlds[0]
    bpy.context.scene.world = world
    world.use_nodes = True
    nt = world.node_tree
    nt.nodes.clear()

    out = nt.nodes.new("ShaderNodeOutputWorld")
    bg  = nt.nodes.new("ShaderNodeBackground")
    sky = nt.nodes.new("ShaderNodeTexSky")
    sky.sun_elevation = 0.9
    sky.sun_rotation  = 1.2
    bg.inputs["Strength"].default_value = sky_strength
```

```
    nt.links.new(sky.outputs["Color"], bg.inputs["Color"])
    nt.links.new(bg.outputs["Background"], out.inputs["Surface"])

    if volumetric_density > 0:
        vol = nt.nodes.new("ShaderNodeVolumePrincipled")
        vol.inputs["Density"].default_value = volumetric_density
        nt.links.new(vol.outputs["Volume"], out.inputs["Volume"])


def add_sun_light(strength=3.0):
    data = bpy.data.lights.new(name="Sun", type='SUN')
    obj  = bpy.data.objects.new(name="Sun", object_data=data)
    bpy.context.collection.objects.link(obj)
    obj.rotation_euler = Euler((radians(35), radians(-10), radians(25)), 'XYZ')
    data.energy = strength
    data.angle = radians(1.0)


def make_material_ground():
    mat = bpy.data.materials.new("GroundPBR")
    mat.use_nodes = True
    nt = mat.node_tree
    bsdf = nt.nodes.get("Principled BSDF")

    tex = nt.nodes.new("ShaderNodeTexNoise")
    tex.inputs["Scale"].default_value = 6.0

    bump = nt.nodes.new("ShaderNodeBump")
    bump.inputs["Strength"].default_value = 0.3

    colramp = nt.nodes.new("ShaderNodeValToRGB")
    colramp.color_ramp.elements[0].color = (0.25, 0.25, 0.25, 1)
    colramp.color_ramp.elements[1].color = (0.12, 0.12, 0.12, 1)

    nt.links.new(tex.outputs["Fac"], bump.inputs["Height"])
    nt.links.new(bump.outputs["Normal"], bsdf.inputs["Normal"])
    nt.links.new(tex.outputs["Fac"], colramp.inputs["Fac"])
    nt.links.new(colramp.outputs["Color"], bsdf.inputs["Base Color"])

    bsdf.inputs["Roughness"].default_value = 0.9
    return mat


def add_ground(size=30):
    bpy.ops.mesh.primitive_plane_add(size=size, location=(0, 0, 0))
    plane = bpy.context.active_object
    plane.name = "Ground"
    plane.data.materials.append(make_material_ground())
```

```
    bpy.ops.rigidbody.object_add(type='PASSIVE')
    plane.rigid_body.friction = 0.6
    plane.rigid_body.restitution = 0.0


def add_camera_path(total_frames, radius=12.0, height=6.0):
    bpy.ops.curve.primitive_bezier_circle_add(radius=radius, location=(0, 0,
height))
    path = bpy.context.active_object
    path.name = "CamPath"

    bpy.ops.object.camera_add(location=(radius, 0, height))
    cam = bpy.context.active_object
    cam.data.lens = 45
    cam.data.dof.use_dof = True
    cam.data.dof.focus_distance = 8.0
    cam.data.dof.aperture_fstop = 2.8

    con = cam.constraints.new(type='FOLLOW_PATH')
    con.target = path
    con.use_curve_follow = True

    bpy.context.scene.frame_set(1)
    con.offset_factor = 0.0
    con.keyframe_insert(data_path="offset_factor", frame=1)
    con.offset_factor = 1.0
    con.keyframe_insert(data_path="offset_factor", frame=total_frames)

    empty = bpy.data.objects.new("LookAt", None)
    empty.location = (0, 0, 1.25)
    bpy.context.collection.objects.link(empty)

    tcon = cam.constraints.new(type='TRACK_TO')
    tcon.target = empty
    tcon.track_axis = 'TRACK_NEGATIVE_Z'
    tcon.up_axis = 'UP_Y'


def rough_metal_material(name="Metal"):
    mat = bpy.data.materials.new(name)
    mat.use_nodes = True
    bsdf = mat.node_tree.nodes.get("Principled BSDF")
    bsdf.inputs["Metallic"].default_value = 0.9
    bsdf.inputs["Roughness"].default_value = 0.3
    return mat
```

```
def colored_plastic_material(name="Plastic"):
    mat = bpy.data.materials.new(name)
    mat.use_nodes = True
    bsdf = mat.node_tree.nodes.get("Principled BSDF")
    rc = (
        0.45 + 0.5 * random.random(),
        0.45 + 0.5 * random.random(),
        0.45 + 0.5 * random.random(),
        1.0,
    )
    bsdf.inputs["Base Color"].default_value = rc
    bsdf.inputs["Roughness"].default_value = 0.35
    return mat


def add_rigid_stack(n=22, spread=8.0):
    for _ in range(n):
        t = random.random()
        x = (random.random() - 0.5) * spread
        y = (random.random() - 0.5) * spread
        z = 4 + random.random() * 6
        if t < 0.5:
            bpy.ops.mesh.primitive_uv_sphere_add(
                segments=48, ring_count=32, radius=0.4 + random.random() * 0.6,
                location=(x, y, z)
            )
        else:
            bpy.ops.mesh.primitive_cube_add(
                size=0.8 + random.random() * 0.8, location=(x, y, z)
            )
        obj = bpy.context.active_object
        obj.data.materials.append(colored_plastic_material())
        bpy.ops.rigidbody.object_add(type='ACTIVE')
        obj.rigid_body.friction = 0.5
        obj.rigid_body.restitution = 0.1
        obj.rigid_body.linear_damping = 0.01
        obj.rigid_body.angular_damping = 0.01
        obj.rigid_body.mass = 0.5 + random.random() * 2.0


def add_statue_and_cloth():
    bpy.ops.mesh.primitive_monkey_add(size=1.5, location=(0, 0, 1.5))
    su = bpy.context.active_object
    su.name = "Statue"
    su.rotation_euler = Euler((radians(0), radians(0), radians(35)), 'XYZ')
    su.data.materials.append(rough_metal_material())

    bpy.ops.object.modifier_add(type='COLLISION')
```

```
    su.modifiers["Collision"].settings.thickness_outer = 0.02

    bpy.ops.mesh.primitive_plane_add(size=6.0, location=(0, 0, 4.5))
    cloth = bpy.context.active_object
    cloth.name = "Cloth"

    bpy.ops.object.mode_set(mode='EDIT')
    bpy.ops.mesh.subdivide(number_cuts=60)
    bpy.ops.object.mode_set(mode='OBJECT')

    mat = bpy.data.materials.new("ClothMat")
    mat.use_nodes = True
    bsdf = mat.node_tree.nodes.get("Principled BSDF")
    bsdf.inputs["Base Color"].default_value = (0.7, 0.05, 0.05, 1.0)
    bsdf.inputs["Roughness"].default_value = 0.85
    cloth.data.materials.append(mat)

    bpy.ops.object.modifier_add(type='CLOTH')
    cloth.modifiers["Cloth"].settings.quality = 8
    cloth.modifiers["Cloth"].settings.time_scale = 1.0
    cloth.modifiers["Cloth"].settings.air_damping = 1.0
    cloth.modifiers["Cloth"].collision_settings.use_self_collision = True
    cloth.modifiers["Cloth"].collision_settings.self_friction = 5.0
    cloth.modifiers["Cloth"].collision_settings.distance_min = 0.005

    # Pin corner vertices
    vg = cloth.vertex_groups.new(name="Pin")
    to_pin = []
    for i, v in enumerate(cloth.data.vertices):
        co = cloth.matrix_world @ v.co
        if abs(co.x) > 2.7 and abs(co.y) > 2.7:
            to_pin.append(i)
    vg.add(to_pin, 1.0, 'REPLACE')
    cloth.modifiers["Cloth"].settings.vertex_group_mass = "Pin"
    cloth.modifiers["Cloth"].settings.pin_stiffness = 0.8

    # Add a wind effector
    bpy.ops.object.effector_add(type='WIND', location=(2.0, -3.0, 2.0))
    wind = bpy.context.active_object
    wind.rotation_euler = Euler((radians(10), radians(0), radians(25)), 'XYZ')
    wind.field.strength = 50.0
    wind.field.flow = 0.5


def simulate_and_render():
    # Baking can improve cloth quality; okay to proceed if calls are missing
    try:
        bpy.ops.ptcache.free_bake_all()
```

```
    except Exception:
        pass
    try:
        bpy.ops.ptcache.bake_all(bake=True)
    except Exception:
        pass
    bpy.ops.render.render(animation=True)

# ------------------ HA glue ------------------
def apply_harmonic_if_any(duration, fps):
    if ha_cfg_inline:
        try:
            import sys as _sys, os as _os
            _sys.path.append(_os.path.abspath("."))
            from harmonic.blender_bridge import apply_ha
            info = apply_ha(bpy.context.scene, ha_cfg_inline, duration, fps)
            print(f"[HA] coherence={info['coherence']:.4f},
suggested_samples={info['suggested_samples']}")
        except Exception as e:
            print("[HA] Failed to apply harmonic config:", e)

# ------------------ main ------------------
def main():
    total_frames = int(duration * fps)
    safe_clear_scene()
    set_cycles_engine(samples, res_w, res_h, fps)
    set_output(out_path, total_frames)
    add_world(sky_strength=1.0, volumetric_density=vol_density)
    add_sun_light(3.0)
    add_ground(30)
    add_camera_path(total_frames, 12.0, 6.0)

    # Apply HA BEFORE building bodies so fields influence simulation
    apply_harmonic_if_any(duration, fps)

    if scenario == "rigid":
        if not bpy.context.scene.rigidbody_world:
            bpy.ops.rigidbody.world_add()
        bpy.context.scene.rigidbody_world.steps_per_second = 240
        bpy.context.scene.rigidbody_world.solver_iterations = 20
        add_rigid_stack(n=22, spread=8.0)
    elif scenario == "cloth":
        add_statue_and_cloth()
    else:
        print(f"Unknown scenario '{scenario}', defaulting to rigid.")
        if not bpy.context.scene.rigidbody_world:
            bpy.ops.rigidbody.world_add()
        add_rigid_stack(n=18, spread=6.0)
```

```
    simulate_and_render()
    print("✅ Done. Video saved to:", bpy.context.scene.render.filepath)

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

---

harmonic/init.py

```
# Harmonic Algebra integration layer for Blender-based physics
```

harmonic/harmonic_core.py

```
from dataclasses import dataclass, field
from typing import List, Dict, Any, Tuple
import math

Vec3 = Tuple[float, float, float]

def clamp(x: float, lo: float, hi: float) -> float:
    return max(lo, min(hi, x))

@dataclass
class SafetyOperatorS:
    max_force: float = 200.0
    max_density: float = 0.05
    max_wind: float = 200.0
    max_torque: float = 200.0

    def force(self, f: float) -> float:
        return clamp(f, -self.max_force, self.max_force)

    def density(self, d: float) -> float:
        return clamp(d, 0.0, self.max_density)

    def wind(self, w: float) -> float:
        return clamp(w, -self.max_wind, self.max_wind)

    def torque(self, t: float) -> float:
        return clamp(t, -self.max_torque, self.max_torque)

@dataclass
class ResonanceComponent:
```

```
    type: str
    params: Dict[str, Any]

    def value_at(self, p: Vec3, t: float) -> float:
        """Toy analytic fields; swap with your real HA equations."""
        if self.type == "resonant_vortex":
            cx, cy, cz = self.params.get("center", (0, 0, 1.5))
            r     = self.params.get("radius", 5.0)
            freq  = self.params.get("frequency", 0.3)
            phase = self.params.get("phase", 0.0)
            dx, dy = p[0] - cx, p[1] - cy
            rho = math.hypot(dx, dy) / max(1e-6, r)
            amp = self.params.get("strength", 35.0) * math.exp(-rho * rho)
            return amp * math.sin(2 * math.pi * freq * t + phase)
        if self.type == "turbulence":
            seed = self.params.get("seed", 13)
            s = math.sin((p[0] + 1.7) * (p[1] - 2.3) * (p[2] + 0.5) * math.pi +
t * 1.618 + seed)
            return self.params.get("strength", 2.5) * s
        if self.type == "radial_pulse":
            cx, cy, cz = self.params.get("center", (0, 0, 1.0))
            speed  = self.params.get("speed", 1.0)
            period = self.params.get("period", 4.0)
            d = math.dist(p, (cx, cy, cz))
            return math.sin(2 * math.pi * (t / period - d / max(1e-4, speed)))
        return 0.0

@dataclass
class HarmonicField:
    components: List[ResonanceComponent] = field(default_factory=list)

    def signal(self, p: Vec3, t: float) -> float:
        return sum(c.value_at(p, t) for c in self.components)

    def coherence(self, samples: List[Tuple[Vec3, float]]) -> float:
        vals = [self.signal(pos, t) for (pos, t) in samples]
        if not vals:
            return 0.0
        mu = sum(vals) / len(vals)
        var = sum((v - mu) ** 2 for v in vals) / len(vals)
        return 1.0 / (1e-6 + var)

@dataclass
class HarmonicScheduler:
    low: int = 192
    mid: int = 384
    high: int = 768
    mid_threshold: float = 2.0
```

```
    high_threshold: float = 10.0

    def choose_samples(self, coh: float) -> int:
        if coh >= self.high_threshold:
            return self.high
        if coh >= self.mid_threshold:
            return self.mid
        return self.low

@dataclass
class AHDE:
    """Adaptive Harmonic Decision Engine (policy hooks)."""
    def choose_scenario(self, baseline: str, coherence: float) -> str:
        return "cloth" if coherence > 5.0 else baseline

@dataclass
class HAConfig:
    safety: SafetyOperatorS
    field: HarmonicField
    scheduler: HarmonicScheduler
    ahde: AHDE

def build_from_config(cfg: dict) -> HAConfig:
    S = SafetyOperatorS(**cfg.get("safety", {}))
    comps = [ResonanceComponent(type=c.get("type", "resonant_vortex"), params=c)
             for c in cfg.get("fields", [])]
    HF = HarmonicField(components=comps)
    HS = HarmonicScheduler(**cfg.get("scheduler", {}))
    A  = AHDE()
    return HAConfig(safety=S, field=HF, scheduler=HS, ahde=A)
```

harmonic/blender_bridge.py

```
from typing import Dict, Any
from .harmonic_core import build_from_config, HAConfig
import math

try:
    import bpy
    from mathutils import Euler
except Exception:  # not in Blender
    bpy = None
    Euler = None


def _need_bpy():
```

```
    if bpy is None:
        raise RuntimeError("Run inside Blender (bpy unavailable).")


def apply_ha(scene, ha_cfg: Dict[str, Any], duration: float, fps: int):
    """Build HA from config and apply to Blender world/fields. Returns
metrics."""
    _need_bpy()
    ha: HAConfig = build_from_config(ha_cfg)

    # World volumetrics via Safety S
    if scene.world and scene.world.node_tree:
        nd = ha.safety.density(ha_cfg.get("world_volumetrics", 0.0))
        if nd > 0:
            nt = scene.world.node_tree
            out = next((n for n in nt.nodes if n.bl_idname ==
"ShaderNodeOutputWorld"), None)
            vol = next((n for n in nt.nodes if n.bl_idname ==
"ShaderNodeVolumePrincipled"), None)
            if not out:
                out = nt.nodes.new("ShaderNodeOutputWorld")
            if not vol:
                vol = nt.nodes.new("ShaderNodeVolumePrincipled")
            vol.inputs["Density"].default_value = nd
            nt.links.new(vol.outputs["Volume"], out.inputs["Volume"])

    # Force fields
    _add_forcefields(ha, duration, fps)

    # Coherence → render samples
    coh = _coherence(ha)
    scene.cycles.samples = ha.scheduler.choose_samples(coh)
    return {"coherence": coh, "suggested_samples": scene.cycles.samples}


def _coherence(ha: HAConfig) -> float:
    samples = []
    for x in (-4, 0, 4):
        for y in (-4, 0, 4):
            for t in (0.0, 0.5, 1.0, 2.0):
                samples.append(((x, y, 1.0), t))
    return ha.field.coherence(samples)


def _add_forcefields(ha: HAConfig, duration: float, fps: int):
    _need_bpy()
    total = int(duration * fps)
```

```
    parent = bpy.data.objects.new("HA_ForceFields", None)
    bpy.context.collection.objects.link(parent)

    for comp in ha.field.components:
        t = comp.type
        if t == "resonant_vortex":
            center   = comp.params.get("center", (0, 0, 1.5))
            strength = ha.safety.torque(comp.params.get("strength", 35.0))
            radius   = comp.params.get("radius", 5.0)
            freq     = comp.params.get("frequency", 0.3)

            bpy.ops.object.effector_add(type='VORTEX', location=center)
            obj = bpy.context.active_object
            obj.parent = parent
            obj.field.flow = 1.0
            obj.field.distance_max = radius

            # Subtle animated oscillation of strength
            for fr in (1, total // 2, total):
                tsec = fr / fps
                obj.field.strength = ha.safety.torque(
                    strength * (0.85 + 0.3 * math.sin(2 * math.pi * freq *
tsec))
                )
                obj.keyframe_insert(data_path="field.strength", frame=fr)

        elif t == "turbulence":
            strength = ha.safety.force(comp.params.get("strength", 2.5))
            size     = comp.params.get("size", 3.0)
            bpy.ops.object.effector_add(type='TURBULENCE', location=(0, 0, 2.0))
            obj = bpy.context.active_object
            obj.parent = parent
            obj.field.strength = strength
            obj.field.size = size

        elif t == "radial_pulse":
            center = comp.params.get("center", (0, 0, 1.0))
            base   = ha.safety.force(comp.params.get("amplitude", 40.0))
            speed  = comp.params.get("speed", 1.0)
            bpy.ops.object.effector_add(type='FORCE', location=center)
            obj = bpy.context.active_object
            obj.parent = parent

            step = max(1, total // 24)
            for fr in range(1, total + 1, step):
                tsec = fr / fps
                obj.field.strength = ha.safety.force(
```

```
                    base * math.sin(2 * math.pi * tsec / max(0.01, speed))
                )
                obj.keyframe_insert(data_path="field.strength", frame=fr)
        else:
            bpy.ops.object.effector_add(type='FORCE', location=(0, 0, 1.0))
            obj = bpy.context.active_object
            obj.parent = parent
            obj.field.strength = ha.safety.force(5.0)
```

---

configs/default_rigid.json

```
{
  "scenario": "rigid",
  "duration": 6,
  "fps": 24,
  "res_w": 1920,
  "res_h": 1080,
  "samples": 256,
  "out_path": "output/rigid_demo.mp4",
  "volumetrics": 0.0,
  "seed": 42
}
```

configs/default_cloth.json

```
{
  "scenario": "cloth",
  "duration": 8,
  "fps": 24,
  "res_w": 2560,
  "res_h": 1440,
  "samples": 384,
  "out_path": "output/cloth_demo.mp4",
  "volumetrics": 0.012,
  "seed": 42
}
```

configs/only_ha.json

```
{
  "ha": {
```

```
    "world_volumetrics": 0.015,
    "safety": { "max_force": 180, "max_density": 0.03 },
    "fields": [
      { "type": "resonant_vortex", "center": [0, 0, 1.25], "strength": 36,
"radius": 6.0, "frequency": 0.3 },
      { "type": "turbulence",      "seed": 11, "strength": 2.2, "size": 2.8 }
    ],
    "scheduler": { "low": 192, "mid": 384, "high": 640, "mid_threshold": 1.8,
"high_threshold": 8.0 }
  }
}
```

configs/ha_rigid.json

```
{
  "scenario": "rigid",
  "duration": 6,
  "fps": 24,
  "res_w": 1920,
  "res_h": 1080,
  "samples": 256,
  "out_path": "output/rigid_ha.mp4",
  "volumetrics": 0.0,
  "seed": 42,
  "ha": {
    "world_volumetrics": 0.015,
    "safety": { "max_force": 180, "max_density": 0.03 },
    "fields": [
      { "type": "resonant_vortex", "center": [0, 0, 1.25], "strength": 36,
"radius": 6.0, "frequency": 0.3 },
      { "type": "turbulence",      "seed": 11, "strength": 2.2, "size": 2.8 }
    ],
    "scheduler": { "low": 192, "mid": 384, "high": 640, "mid_threshold": 1.8,
"high_threshold": 8.0 }
  }
}
```

configs/ha_cloth.json

```
{
  "scenario": "cloth",
  "duration": 8,
  "fps": 24,
  "res_w": 2560,
```

```
  "res_h": 1440,
  "samples": 384,
  "out_path": "output/cloth_ha.mp4",
  "volumetrics": 0.012,
  "seed": 42,
  "ha": {
    "world_volumetrics": 0.02,
    "safety": { "max_force": 150, "max_density": 0.03 },
    "fields": [
      { "type": "resonant_vortex", "center": [0.2, -0.2, 1.4], "strength": 40,
"radius": 5.0, "frequency": 0.22 },
      { "type": "radial_pulse",    "center": [0, 0, 1.4],   "amplitude": 32,
"speed": 1.0 }
    ],
    "scheduler": { "low": 256, "mid": 512, "high": 768, "mid_threshold": 2.5,
"high_threshold": 12.0 }
  }
}
```

---

run_blender_windows.bat

```
@echo off
REM === Edit this to your Blender path ===
set BLENDER="C:\Program Files\Blender Foundation\Blender 4.1\blender.exe"

echo Running baseline rigid...
%BLENDER% -b -P scripts\blender_sim.py -- --config configs\default_rigid.json

echo.
echo Running baseline cloth...
%BLENDER% -b -P scripts\blender_sim.py -- --config configs\default_cloth.json

echo.
echo Running HA rigid...
%BLENDER% -b -P scripts\blender_sim.py -- --config configs\ha_rigid.json

echo.
echo Running HA cloth...
%BLENDER% -b -P scripts\blender_sim.py -- --config configs\ha_cloth.json

echo.
echo Done. Videos are in .\output
pause
```

run_blender_unix.sh

```
#!/usr/bin/env bash
BLENDER="${BLENDER:-blender}"

echo "Running baseline rigid..."
"$BLENDER" -b -P scripts/blender_sim.py -- --config configs/default_rigid.json

echo

echo "Running baseline cloth..."
"$BLENDER" -b -P scripts/blender_sim.py -- --config configs/default_cloth.json

echo

echo "Running HA rigid..."
"$BLENDER" -b -P scripts/blender_sim.py -- --config configs/ha_rigid.json

echo

echo "Running HA cloth..."
"$BLENDER" -b -P scripts/blender_sim.py -- --config configs/ha_cloth.json

echo

echo "Done. Videos are in ./output/"
```

Troubleshooting

- **GPU not used / slow renders** → open Blender GUI once: *Edit* → *Preferences* → *System* and enable your GPU for Cycles. Script will auto-use GPU if available.
- **"blender: command not found"** → on macOS/Linux, use the full path to Blender or add it to `PATH`.
- **No output file** → check `output/` relative to project root; ensure `out_path` is valid.
- **Weird lighting** → drop an HDRI in the World nodes for natural light, or raise `samples` to 512–1024.

Where to plug in your real HA

- Replace the toy fields in `harmonic/harmonic_core.py → ResonanceComponent.value_at()` with your true equations (QRTM, HRDE, etc.).
- Extend `HarmonicScheduler` to map coherence to render settings (samples, motion blur shutter, DOF, etc.).
- Drive cinematography from field gradients (e.g., camera radius/height/focus as functions of ∇field).

This build is intentionally minimal and stable, so you can slot your operators without fighting syntax gremlins. Enjoy the pretty physics while you wire up the deep math.