

VPython Program Rewrite Request

<https://gemini.google.com/app/5af526a1858aa4b5>

User prompt: I would like to rewrite a VPythom program.

Response: Share your current VPython code and let me know what you'd like to change or improve.

Common rewrite goals include:

- **Modernizing legacy code** (e.g., updating old `visual` module syntax to modern VPython 7 / Web VPython).
- **Porting to another library** (e.g., converting 3D VPython simulations to Matplotlib, Pygame, or WebGL/Three.js).
- **Refactoring & Performance** (e.g., cleaning up event loops, organizing into classes, or smoothing animation frame rates).
- **Adding features** (e.g., adding user controls like sliders/buttons, camera controls, or new physics behavior).

Whenever you're ready, paste the code and your goal!

User prompt: The code is for the QuarkNet online version of the QuarkNet Rolling with Rutherford program, found here: <https://www.glowscript.org/#/user/jeremy.wegner/folder/MyPrograms/program/1RollWithRuthSimple/edit> . I would like to remove the visual barrier, reduce the number of targets to 4, arranged at corners of a square, and have the 10 probes come from different random directions. That is a start.

Response: Here is a clean, modern **Web VPython** rewrite of the **Rolling with Rutherford** simulation incorporating all your specifications:

1. **Visual Barrier Removed:** Full visibility of all targets and incoming probes.
2. **4 Corner Targets:** Positioned at the corners of a square centered at the origin.
3. **10 Probes from Random Directions:** Spawned around the perimeter of the simulation area at random angles, launched toward the central target region.
4. **Collision Physics & Scattering Trails:** Probes leave trails showing how they scatter upon colliding with the target spheres.

Modern Web VPython Code

Web VPython 3.2

```
# --- SCENE SETUP ---
scene = canvas(
    title="<b>Rolling with Rutherford (4 Targets)</b>",
    width=800, height=600,
    center=vec(0, 0, 0),
    background=color.gray(0.1)
)

# Reference ground grid/plate
ground = box(pos=vec(0, -0.2, 0), size=vec(24, 0.1, 24), color=color.gray(0.3))

# --- TARGET CONFIGURATION ---
# 4 targets arranged at corners of a square (side length = 4 units)
square_half_side = 2.0
target_radius = 0.6

target_positions = [
    vec(-square_half_side, 0, -square_half_side),
    vec( square_half_side, 0, -square_half_side),
    vec( square_half_side, 0,  square_half_side),
    vec(-square_half_side, 0,  square_half_side)
]

targets = []
for pos_vec in target_positions:
    t = sphere(
```

```

        pos=pos_vec,
        radius=target_radius,
        color=color.red,
        shininess=0.8
    )
    targets.append(t)

# Visual outline for the target square bounding area
square_border = curve(
    pos=[
        vec(-square_half_side, 0.01, -square_half_side),
        vec( square_half_side, 0.01, -square_half_side),
        vec( square_half_side, 0.01,  square_half_side),
        vec(-square_half_side, 0.01,  square_half_side),
        vec(-square_half_side, 0.01, -square_half_side)
    ],
    color=color.yellow,
    radius=0.03
)

# --- PROBE CONFIGURATION ---
num_probes = 10
probe_radius = 0.25
launch_radius = 10.0    # Distance from center where probes spawn
probe_speed = 4.0       # Speed of probes

probes = []

for i in range(num_probes):
    # Random angle around the perimeter
    angle = random() * 2 * pi

    # Starting position on the perimeter
    spawn_pos = vec(launch_radius * cos(angle), 0, launch_radius * sin(angle))

    # Aim toward a point inside/near the central target square (with slight randomness)
    target_aim = vec((random() - 0.5) * 4, 0, (random() - 0.5) * 4)
    velocity_dir = hat(target_aim - spawn_pos)

    p = sphere(
        pos=spawn_pos,
        radius=probe_radius,
        color=color.cyan,
        make_trail=True,
        trail_type="curve",
        trail_radius=0.05,
        retain=100
    )
    p.v = velocity_dir * probe_speed
    p.has_collided = False
    probes.append(p)

# --- SIMULATION LOOP ---
dt = 0.02
total_hits = 0

label_hits = label(pos=vec(0, 8, 0), text="Total Target Hits: 0", height=16, box=False, color=color.white)

while True:
    rate(50)

    for probe in probes:
        # Move probe
        probe.pos = probe.pos + probe.v * dt

        # Check collision with targets
        for target in targets:

```

```

# Distance vector between probe and target center
r_vec = probe.pos - target.pos
dist = mag(r_vec)

# Check hard-sphere elastic collision threshold
if dist <= (probe.radius + target.radius):
    # Normal unit vector from target center to probe center
    normal = hat(r_vec)

    # Reflect probe velocity vector (elastic rebound)
    probe.v = probe.v - 2 * dot(probe.v, normal) * normal

    # Push probe slightly outside target radius to prevent sticking/double hits
    probe.pos = target.pos + normal * (probe.radius + target.radius + 0.02)

    if not probe.has_collided:
        total_hits += 1
        probe.has_collided = True
        label_hits.text = f"Total Target Hits: {total_hits}"

# Reset probe if it travels too far outside the grid
if mag(probe.pos) > launch_radius + 2:
    # Respawn from a new random location
    new_angle = random() * 2 * pi
    probe.pos = vec(launch_radius * cos(new_angle), 0, launch_radius * sin(new_angle))
    target_aim = vec((random() - 0.5) * 4, 0, (random() - 0.5) * 4)
    probe.v = hat(target_aim - probe.pos) * probe_speed
    probe.has_collided = False
    probe.clear_trail()

```

Key Features & How It Works:

- 1. Target Geometry:** The 4 targets are placed at $(-2, 0, -2)$, $(2, 0, -2)$, $(2, 0, 2)$, and $(-2, 0, 2)$ forming a 4×4 square grid outline.
- 2. Omnidirectional Probe Generation:** Angles are generated using $\text{random()} * 2 * \pi$ along a perimeter of radius 10. The probe velocities are directed inward toward the target square.
- 3. Hard-Sphere Elastic Scattering:** When a probe hits a target sphere, its velocity vector reflects across the collision normal vector ($\mathbf{v}' = \mathbf{v} - 2(\mathbf{v} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}$).
- 4. Respawning Loop:** When probes exit the viewing arena, they automatically reset and re-launch from a new random direction, providing continuous data collection.

User prompt: Nice but the visual rendering needs to be rotated 90 degrees.

Response: To rotate the visual perspective by 90° around the vertical axis, set `scene.forward` (the direction vector the camera points).

Here is the updated code with the camera rotated 90° to view the scene from the side:

Web VPython 3.2

```

# --- SCENE SETUP ---
scene = canvas(
    title="<b>Rolling with Rutherford (4 Targets)</b>",
    width=800, height=600,
    center=vec(0, 0, 0),
    background=color.gray(0.1),
    forward=vec(-1, -0.6, 0), # Rotated camera view 90 degrees
    up=vec(0, 1, 0)
)

# Reference ground grid/plate
ground = box(pos=vec(0, -0.2, 0), size=vec(24, 0.1, 24), color=color.gray(0.3))

```

```

# --- TARGET CONFIGURATION ---
# 4 targets arranged at corners of a square (side length = 4 units)
square_half_side = 2.0
target_radius = 0.6

target_positions = [
    vec(-square_half_side, 0, -square_half_side),
    vec( square_half_side, 0, -square_half_side),
    vec( square_half_side, 0,  square_half_side),
    vec(-square_half_side, 0,  square_half_side)
]

targets = []
for pos_vec in target_positions:
    t = sphere(
        pos=pos_vec,
        radius=target_radius,
        color=color.red,
        shininess=0.8
    )
    targets.append(t)

# Visual outline for the target square bounding area
square_border = curve(
    pos=[
        vec(-square_half_side, 0.01, -square_half_side),
        vec( square_half_side, 0.01, -square_half_side),
        vec( square_half_side, 0.01,  square_half_side),
        vec(-square_half_side, 0.01,  square_half_side),
        vec(-square_half_side, 0.01, -square_half_side)
    ],
    color=color.yellow,
    radius=0.03
)

# --- PROBE CONFIGURATION ---
num_probes = 10
probe_radius = 0.25
launch_radius = 10.0    # Distance from center where probes spawn
probe_speed = 4.0       # Speed of probes

probes = []

for i in range(num_probes):
    # Random angle around the perimeter
    angle = random() * 2 * pi

    # Starting position on the perimeter
    spawn_pos = vec(launch_radius * cos(angle), 0, launch_radius * sin(angle))

    # Aim toward a point inside/near the central target square
    target_aim = vec((random() - 0.5) * 4, 0, (random() - 0.5) * 4)
    velocity_dir = hat(target_aim - spawn_pos)

    p = sphere(
        pos=spawn_pos,
        radius=probe_radius,
        color=color.cyan,
        make_trail=True,
        trail_type="curve",
        trail_radius=0.05,
        retain=100
    )
    p.v = velocity_dir * probe_speed
    p.has_collided = False
    probes.append(p)

```

```

# --- SIMULATION LOOP ---
dt = 0.02
total_hits = 0

label_hits = label(pos=vec(0, 8, 0), text="Total Target Hits: 0", height=16, box=False, color=color.white)

while True:
    rate(50)

    for probe in probes:
        # Move probe
        probe.pos = probe.pos + probe.v * dt

        # Check collision with targets
        for target in targets:
            r_vec = probe.pos - target.pos
            dist = mag(r_vec)

            # Hard-sphere elastic collision
            if dist <= (probe.radius + target.radius):
                normal = hat(r_vec)

                # Reflect probe velocity
                probe.v = probe.v - 2 * dot(probe.v, normal) * normal

                # Push probe outside target radius to prevent sticking
                probe.pos = target.pos + normal * (probe.radius + target.radius + 0.02)

                if not probe.has_collided:
                    total_hits += 1
                    probe.has_collided = True
                    label_hits.text = f"Total Target Hits: {total_hits}"

        # Reset probe if it travels too far outside the grid
        if mag(probe.pos) > launch_radius + 2:
            new_angle = random() * 2 * pi
            probe.pos = vec(launch_radius * cos(new_angle), 0, launch_radius * sin(new_angle))
            target_aim = vec((random() - 0.5) * 4, 0, (random() - 0.5) * 4)
            probe.v = hat(target_aim - probe.pos) * probe_speed
            probe.has_collided = False
            probe.clear_trail()

```

Camera Angles Options:

- **Current setting:** forward=vec(-1, -0.6, 0) looks from the right side facing left.
- **Opposite 90° angle:** Change to forward=vec(1, -0.6, 0) if you want to look from the left side facing right instead.
- **Top-down 90° tilt:** Set forward=vec(0, -1, 0) with up=vec(0, 0, -1) for an overhead view.

User prompt: Very nice. Modifications: 1. We do not need the trails after the probes. 2. Can you make the targets shake in place for about 1 sec after they are hit?

Response: Here is the updated Web VPython script with probe trails removed and a 1-second shake animation added to the targets upon collision:

Web VPython 3.2

```

# --- SCENE SETUP ---
scene = canvas(
    title="<b>Rolling with Rutherford (4 Targets)</b>",
    width=800, height=600,
    center=vec(0, 0, 0),
    background=color.gray(0.1),

```

```

    forward=vec(-1, -0.6, 0), # Rotated perspective view
    up=vec(0, 1, 0)
)

# Reference ground grid/plate
ground = box(pos=vec(0, -0.2, 0), size=vec(24, 0.1, 24), color=color.gray(0.3))

# --- TARGET CONFIGURATION ---
square_half_side = 2.0
target_radius = 0.6

target_positions = [
    vec(-square_half_side, 0, -square_half_side),
    vec( square_half_side, 0, -square_half_side),
    vec( square_half_side, 0,  square_half_side),
    vec(-square_half_side, 0,  square_half_side)
]

targets = []
for pos_vec in target_positions:
    t = sphere(
        pos=pos_vec,
        radius=target_radius,
        color=color.red,
        shininess=0.8
    )
    t.base_pos = vec(pos_vec.x, pos_vec.y, pos_vec.z) # Store baseline position
    t.shake_timer = 0.0 # Timer for hit animation
    targets.append(t)

# Visual outline for target square bounding area
square_border = curve(
    pos=[
        vec(-square_half_side, 0.01, -square_half_side),
        vec( square_half_side, 0.01, -square_half_side),
        vec( square_half_side, 0.01,  square_half_side),
        vec(-square_half_side, 0.01,  square_half_side),
        vec(-square_half_side, 0.01, -square_half_side)
    ],
    color=color.yellow,
    radius=0.03
)

# --- PROBE CONFIGURATION ---
num_probes = 10
probe_radius = 0.25
launch_radius = 10.0 # Distance from center where probes spawn
probe_speed = 4.0 # Speed of probes

probes = []

for i in range(num_probes):
    angle = random() * 2 * pi
    spawn_pos = vec(launch_radius * cos(angle), 0, launch_radius * sin(angle))
    target_aim = vec((random() - 0.5) * 4, 0, (random() - 0.5) * 4)
    velocity_dir = hat(target_aim - spawn_pos)

    # Probes generated without visual trails
    p = sphere(
        pos=spawn_pos,
        radius=probe_radius,
        color=color.cyan
    )
    p.v = velocity_dir * probe_speed
    p.has_collided = False
    probes.append(p)

```

```

# --- SIMULATION LOOP ---
dt = 0.02
total_hits = 0

label_hits = label(pos=vec(0, 8, 0), text="Total Target Hits: 0", height=16, box=False, color=color.white)

while True:
    rate(50)

    # Handle target shake animations
    for target in targets:
        if target.shake_timer > 0:
            target.shake_timer -= dt
            shake_mag = 0.12 # Amplitude of shake
            offset = vec(
                (random() - 0.5) * shake_mag,
                (random() - 0.5) * shake_mag,
                (random() - 0.5) * shake_mag
            )
            target.pos = target.base_pos + offset
        else:
            target.pos = target.base_pos # Snap back to default resting position

    for probe in probes:
        # Move probe
        probe.pos = probe.pos + probe.v * dt

        # Check collision with targets (using base_pos for stable bounce physics)
        for target in targets:
            r_vec = probe.pos - target.base_pos
            dist = mag(r_vec)

            if dist <= (probe.radius + target.radius):
                normal = hat(r_vec)

                # Reflect probe velocity
                probe.v = probe.v - 2 * dot(probe.v, normal) * normal

                # Push probe outside target radius to prevent sticking
                probe.pos = target.base_pos + normal * (probe.radius + target.radius + 0.02)

                # Trigger target 1-second shake timer
                target.shake_timer = 1.0

                if not probe.has_collided:
                    total_hits += 1
                    probe.has_collided = True
                    label_hits.text = f"Total Target Hits: {total_hits}"

        # Reset probe if it travels too far outside the grid
        if mag(probe.pos) > launch_radius + 2:
            new_angle = random() * 2 * pi
            probe.pos = vec(launch_radius * cos(new_angle), 0, launch_radius * sin(new_angle))
            target_aim = vec((random() - 0.5) * 4, 0, (random() - 0.5) * 4)
            probe.v = hat(target_aim - probe.pos) * probe_speed
            probe.has_collided = False

```

What Changed:

1. **Removed Trails:** Stripped out `make_trail=True`, `trail_type`, `trail_radius`, `retain`, and `probe.clear_trail()` so probes now move cleanly without trailing lines.
2. **Target Shake Effect:**
 - Each target sphere tracks its fixed baseline position `base_pos` and a countdown timer `shake_timer`.
 - Upon impact, `shake_timer` is set to 1.0 second.

- On every step of the animation loop, targets with `shake_timer > 0` shift by a small random 3D displacement offset before resetting smoothly once the timer reaches zero.

User prompt: This is very nice. Now the interesting modification. Randomly, but on average for 1 in 20 probes, make the probe orange and about twice the size of a standard probe. When and if it hits a target, it should not bounce but pass through. The target should still bounce and turn yellow for 1 sec.

Response: Here is the updated Web VPython code featuring the special orange "heavy" probes and target color-changing logic:

Web VPython 3.2

```
# --- SCENE SETUP ---
scene = canvas(
    title="<b>Rolling with Rutherford (4 Targets)</b>",
    width=800, height=600,
    center=vec(0, 0, 0),
    background=color.gray(0.1),
    forward=vec(-1, -0.6, 0),
    up=vec(0, 1, 0)
)

# Reference ground grid/plate
ground = box(pos=vec(0, -0.2, 0), size=vec(24, 0.1, 24), color=color.gray(0.3))

# --- TARGET CONFIGURATION ---
square_half_side = 2.0
target_radius = 0.6

target_positions = [
    vec(-square_half_side, 0, -square_half_side),
    vec( square_half_side, 0, -square_half_side),
    vec( square_half_side, 0,  square_half_side),
    vec(-square_half_side, 0,  square_half_side)
]

targets = []
for pos_vec in target_positions:
    t = sphere(
        pos=pos_vec,
        radius=target_radius,
        color=color.red,
        shininess=0.8
    )
    t.base_pos = vec(pos_vec.x, pos_vec.y, pos_vec.z) # Baseline position
    t.shake_timer = 0.0                               # Timer for shake & color change
    targets.append(t)

# Visual outline for target square bounding area
square_border = curve(
    pos=[
        vec(-square_half_side, 0.01, -square_half_side),
        vec( square_half_side, 0.01, -square_half_side),
        vec( square_half_side, 0.01,  square_half_side),
        vec(-square_half_side, 0.01,  square_half_side),
        vec(-square_half_side, 0.01, -square_half_side)
    ],
    color=color.yellow,
    radius=0.03
)

# --- PROBE CONFIGURATION & HELPER ---
num_probes = 10
standard_radius = 0.25
launch_radius = 10.0
probe_speed = 4.0
```

```

def configure_probe(p):
    """Resets or initializes a probe with random position, direction, and 1-in-20 heavy type."""
    angle = random() * 2 * pi
    spawn_pos = vec(launch_radius * cos(angle), 0, launch_radius * sin(angle))
    target_aim = vec((random() - 0.5) * 4, 0, (random() - 0.5) * 4)

    p.pos = spawn_pos
    p.v = hat(target_aim - spawn_pos) * probe_speed
    p.targets_hit = [] # Tracks targets hit on current pass to prevent double-counting as it passes through

    # 1 in 20 chance (5%) for special orange heavy probe
    if random() < 0.05:
        p.is_heavy = True
        p.radius = standard_radius * 2.0
        p.color = color.orange
    else:
        p.is_heavy = False
        p.radius = standard_radius
        p.color = color.cyan

probes = []
for i in range(num_probes):
    p = sphere()
    configure_probe(p)
    probes.append(p)

# --- SIMULATION LOOP ---
dt = 0.02
total_hits = 0

label_hits = label(pos=vec(0, 8, 0), text="Total Target Hits: 0", height=16, box=False, color=color.white)

while True:
    rate(50)

    # Update target shake animations and color resets
    for target in targets:
        if target.shake_timer > 0:
            target.shake_timer -= dt
            shake_mag = 0.12
            offset = vec(
                (random() - 0.5) * shake_mag,
                (random() - 0.5) * shake_mag,
                (random() - 0.5) * shake_mag
            )
            target.pos = target.base_pos + offset
        else:
            target.pos = target.base_pos
            target.color = color.red # Revert back to red when shake timer completes

    for probe in probes:
        # Move probe
        probe.pos = probe.pos + probe.v * dt

        # Check collision with targets
        for target in targets:
            r_vec = probe.pos - target.base_pos
            dist = mag(r_vec)

            if dist <= (probe.radius + target.radius):
                # Ensure each target is registered only once per trajectory pass
                if target not in probe.targets_hit:
                    probe.targets_hit.append(target)
                    total_hits += 1
                    label_hits.text = f"Total Target Hits: {total_hits}"

```

```

target.shake_timer = 1.0 # Trigger 1-second shake

if probe.is_heavy:
    target.color = color.yellow # Turn yellow for heavy probe hit
    # Orange probe passes straight through without bouncing or altering trajectory
else:
    # Standard rebound elastic collision for cyan probes
    normal = hat(r_vec)
    probe.v = probe.v - 2 * dot(probe.v, normal) * normal
    probe.pos = target.base_pos + normal * (probe.radius + target.radius + 0.02)

# Reset probe when out of bounds
if mag(probe.pos) > launch_radius + 2:
    configure_probe(probe)

```

Key Additions:

1. **5% Chance Orange Heavy Probe:** Handled inside `configure_probe()`. When `random() < 0.05`, the probe spawns with `radius = 0.5` (2x standard size) and `color.orange`.
2. **Pass-Through Behavior:** When an orange probe collides with a target sphere, it registers the hit without altering its velocity vector or pushing its position outward, letting it pass straight through.
3. **Yellow Target Reaction:** Heavy probe hits set `target.color = color.yellow` while triggering the 1-second shake. When `shake_timer` reaches 0, the target automatically reverts back to red.
4. **Collision Tracking:** Uses `probe.targets_hit` list to ensure that when a heavy probe passes straight through a target, it only counts as a single hit instead of re-triggering collision physics every frame.

User prompt: One last thing...Please explain how this simulation can be used to represent a dark matter search experiment such as LUX-ZEPLIN and how it falls short.

Response: How the Simulation Represents LUX-ZEPLIN (LZ)

1. **Ambient Dark Matter Flux:** The probes entering from random directions around the perimeter model the "dark matter wind." As Earth moves through the Milky Way's dark matter halo, an isotropic flux of Weakly Interacting Massive Particles (WIMPs) continuously streams through the detector.
2. **Target Nuclei Mass:** The four stationary targets represent the dense target volume of liquid xenon nuclei inside LZ's active Time Projection Chamber (TPC).
3. **Nuclear Recoil & Detector Signals:** The target shaking and turning yellow upon impact simulates **nuclear recoil**. When a dark matter candidate strikes a xenon nucleus, the deposited kinetic energy produces prompt scintillation light (S1) and drift electrons (S2), which PMTs detect as a "event signature."
4. **Scattering Cross-Section:** The probability of a probe colliding with a target visually demonstrates the concept of interaction cross-section (σ). Most space in the arena is empty, illustrating why massive target masses (LZ uses 7 tonnes of liquid xenon) are needed to catch rare events.

Where the Analogy Falls Short

1. **Hard-Sphere Classical vs. Quantum Weak Force:**
 - *Simulation:* Collisions rely on classical billiard-ball contact dynamics based on physical radii ($r_{\text{probe}} + r_{\text{target}}$).
 - *LUX-ZEPLIN:* WIMPs do not have physical "hard surfaces." Interactions are mediated by the weak nuclear force. The interaction probability depends on quantum scattering cross-sections and nuclear form factors rather than mechanical size.
2. **Extreme Rarity of Real Events:**
 - *Simulation:* Probes hit targets roughly 10–20% of the time because the targets occupy a large fraction of the central area.
 - *LUX-ZEPLIN:* WIMP interactions are extraordinarily rare. Billions of dark matter particles pass through LZ every second, but expected signals might yield fewer than a few events per year.
3. **Momentum Conservation and the "Orange Probe":**

- *Simulation*: The orange probe passes straight through a target without altering its path, yet still causes the target to shake and change color.
- *LUX-ZEPLIN*: In real particle physics, energy transfer requires momentum transfer. A dark matter particle cannot impart kinetic energy to recoil a heavy xenon nucleus without being deflected/scattered itself.

4. Background Interference:

- *Simulation*: Every target hit comes from a known probe, with zero false positives.
- *LUX-ZEPLIN*: The primary challenge in direct detection is background noise (gamma rays, solar neutrinos, and radon decays). LZ operates 4,850 feet underground at the Sanford Underground Research Facility (SURF) precisely to shield against cosmic rays that mimic dark matter recoils.

5. Planar vs. 3D Volumetric Fiducialization:

- *Simulation*: Probes move strictly on a 2D plane ($y = 0$).
- *LUX-ZEPLIN*: LZ uses a fully 3D cylindrical liquid volume. Reconstruction of the x, y, z coordinates allows scientists to "fiducialize" the detector—using the outer liquid xenon layer as a self-shielding boundary and analyzing only events in the ultra-clean core.

Would you like to add background noise particles or a fiducial boundary ring to the VPython script to explore how LZ filters out false triggers?
