# From Screen to Scene

## Building VR Experiments with Python

A hands-on workshop for researchers

**Chunyu Qu, Artyom Zinchenko** & **Zhuanghua Shi**

LMU Munich · April 2026

Erasmus+ KA210-VET · xr4vet.eu

15/04/2026

Co-funded by the European Union

# Workshop Overview

## Morning

- **10:00** M1: Welcome & Setup Check
- **10:20** M2: Ursina Fundamentals
- **11:05** Break (20 min)
- **11:25** M3: Interaction & Input
- **12:05** M4: Experiment Paradigm Design
- **12:50** Q&A & Hands-on
- **13:00** Lunch (60 min)

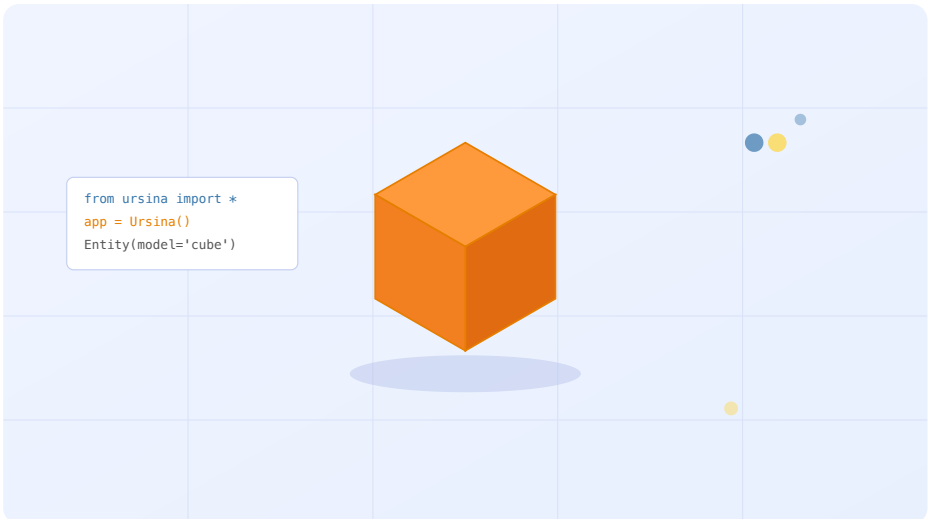
## Afternoon

- **14:00** M5: Capstone – MazeWalker-Py
- **14:45** M6: VR Roadmap
- **15:10** Break (20 min)
- **15:30** M7: Beyond Primitives
- **16:00** Wrap-up & Homework
- **16:30** break (20 min)
- **16:50** Unity VR in Research (Artyom)

**Two tracks**: *Builders* (design & code) · *Runners* (configure & operate)

# Module 1

## Welcome & Setup Check



# Who’s in the Room?

Quick poll – raise your hand:

- Who has written a PsychoPy experiment?
- Who has used Unity or Unreal?
- Who has never touched 3D programming?
- Who has a gamepad plugged in right now?

> This workshop assumes **basic Python** (variables, functions, classes). No 3D or VR experience needed.

# The Landscape: Why Ursina?

| Tool | Language | VR Ready | Science Use |
|---|---|---|---|
| PsychoPy | Python | Limited | Gold standard 2D |
| Unity | C# | Full | Dominant in VR |
| **Ursina** | Python | Via Panda3D | **Sweet spot** |
| Vizard | Python | Full | Commercial |

**Why Ursina?**

- Pure Python – reuse your analysis skills
- Built on Panda3D – production-grade 3D engine
- ~20 extra lines to go from desktop to headset VR

# Setup Check

Run this in your terminal:

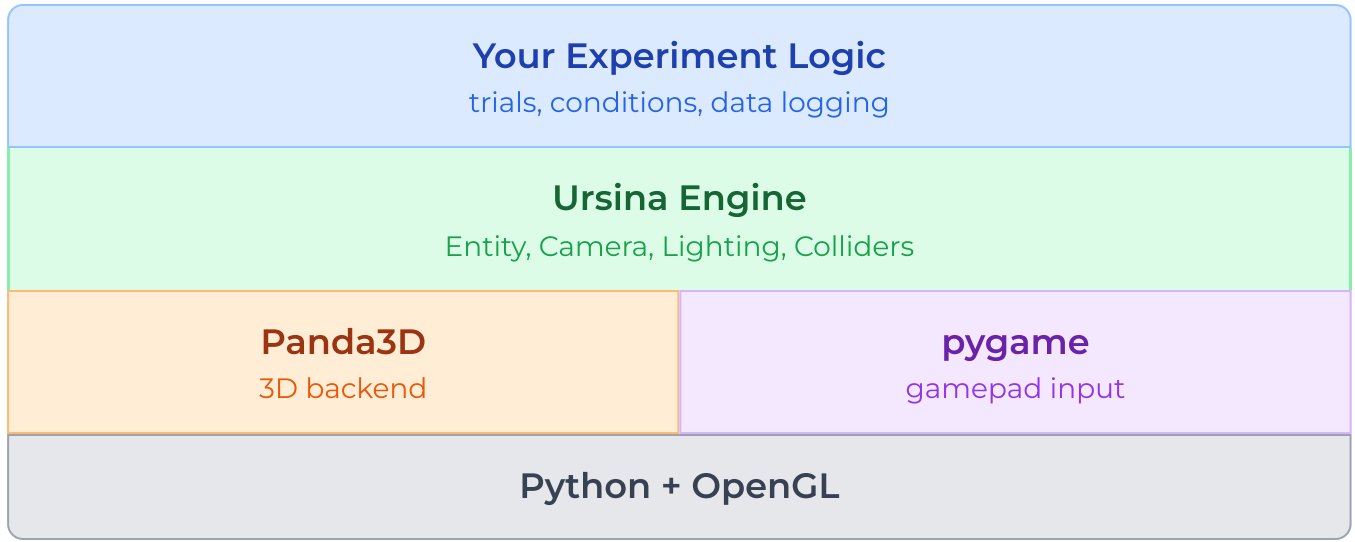
```
cd vr-tutorial
source .venv/bin/activate
python exercises/ex1_hello_ursina/hello_cube.py
```

A window with a **spinning orange cube**.
Middle-click to orbit, scroll to zoom.

If it works – you are ready.

```
from ursina import *

app = Ursina()
Entity(model='cube',
       color=color.orange,
       texture='white_cube')
EditorCamera()
app.run()
```

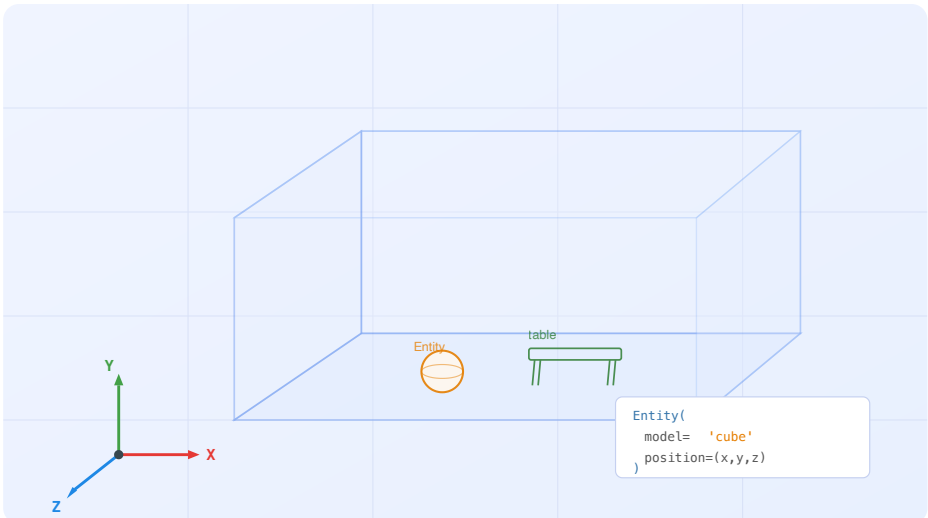
# Architecture Overview




Three layers: **Ursina** (rendering, physics) · **pygame** (gamepad input) · **Your code** (experiment logic, trials, data)

# Module 2

## Ursina Fundamentals



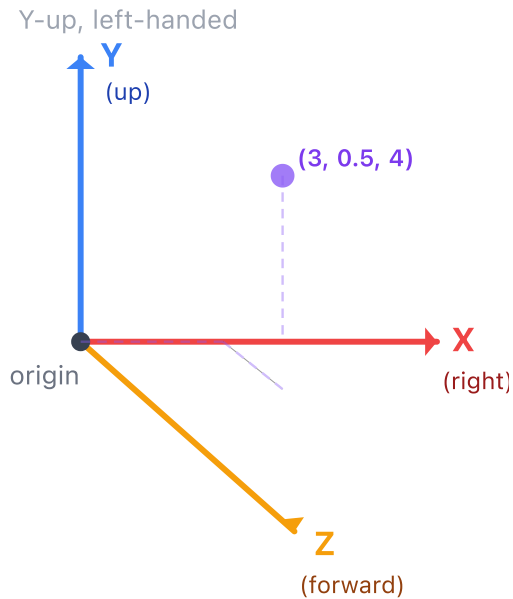
# The Entity: Everything Is One

```python
from ursina import *

app = Ursina()

cube = Entity(
    model='cube',
    color=color.orange,
    texture='white_cube',
    position=(0, 1, 0),
    scale=(2, 1, 1),
    rotation=(0, 45, 0),
    collider='box',
)

app.run()
```

An Entity = **model** + **position** + **appearance** + optional **physics**

> Walls, floors, furniture, collectibles, even the player – they're all Entities with different properties.

# Coordinate System




- **x** – right (+) / left (-)
- **y** – up (+) / down (-)
- **z** – forward (+) / back (-)

`position=(3, 0.5, 4)` means: 3 right, 0.5 up, 4 forward

> Ursina uses a **Y-up, right-handed** coordinate system (same as Panda3D)

# Entity Properties at a Glance

| Property | Example |
| --- | --- |
| `model` | `'cube'`, `'sphere'` |
| `color` | `color.orange` |
| `texture` | `'brick'`, `'grass'` |
| `position` | `(0, 1, 0)` |
| `scale` | `(2, 1, 1)` or `0.5` |

| Property | Example |
| --- | --- |
| `rotation` | `(0, 45, 0)` |
| `collider` | `'box'`, `'mesh'` |
| `parent` | `camera.ui` |
| `enabled` | `True` / `False` |

> `model` and `position` are the minimum; everything else has sensible defaults.

# Cameras: Two Modes

## EditorCamera

Orbit, zoom, pan – great for **building**

```
EditorCamera()
# middle-click orbit, scroll zoom
```

- Free orbit movement
- No physics
- Use for scene building & debugging

## FirstPersonController

WASD + mouse look – great for **experiments**

```
from ursina.prefabs.first_person_controller \
    import FirstPersonController

player = FirstPersonController()
player.gravity = 0
player.position = (0, 1, 0)
```

- Gravity + collision
- Use for participant navigation

# Exercise 1: Build a Room (15 min)

## Steps

1. Open `exercises/ex2_build_a_room/template.py`
2. Create a floor (a flat quad)
3. Add four walls (position + rotate quads)
4. Place at least 3 furniture items
5. Add a `FirstPersonController` and walk around

## Key Hint

A flat floor = `quad` with `rotation_x=90`

```
floor = Entity(
    model='quad',
    scale=(20, 20),
    rotation_x=90,
    color=color.dark_gray,
    texture='white_cube'
)
```

# Exercise 2: Make It Real (10 min)

## Steps

1. Open `exercises/ex3_make_it_real/template.py`
2. Add `collider='box'` to walls and furniture
3. Replace plain colors with textures: `'brick'`, `'grass'`
4. Add `Sky()` for atmosphere
5. Add `DirectionalLight()` for depth

## Before vs After

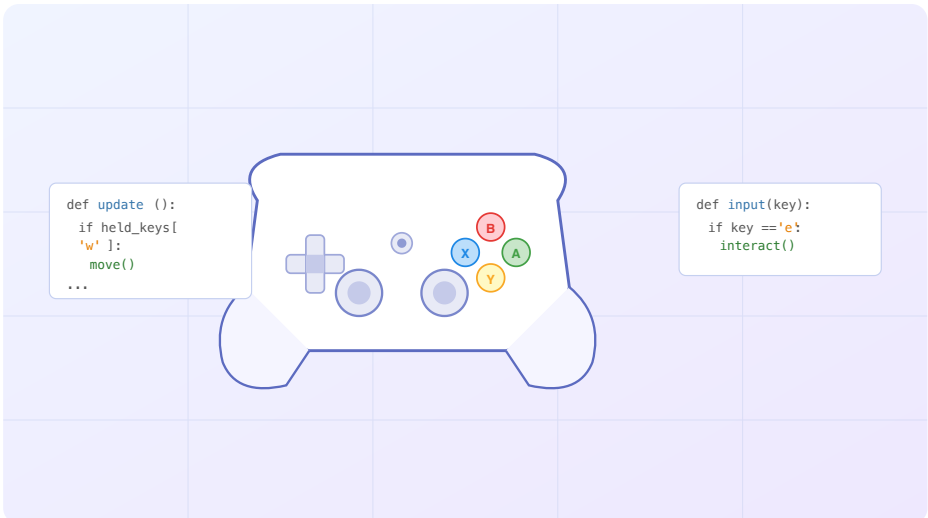
| Without | With |
| --- | --- |
| Walk through walls | Solid walls block you |
| Flat gray surfaces | Brick walls, grass floor |
| Black void background | Sky dome |

# Break

20 minutes – Back at **10:25**

# Module 3

## Adding Interaction & Input



# Ursina’s Input Model

```python
# 1. update() – every frame (~60 fps)
def update():
    if held_keys['w']:
        player.y += time.dt

# 2. input(key) – once per press/release
def input(key):
    if key == 'space':
        jump()

# 3. Entity methods – scoped
class Player(Entity):
    def update(self):
        pass
    def input(self, key):
        pass
```

| Method | Use case |
| --- | --- |
| `update()` | Continuous movement |
| `input()` | Discrete actions |
| `held_keys` | Sustained key holding |

`update()` runs 60x/s – for proximity checks.
`input()` fires once – for trial triggers.

# The Pygame Sidecar Pattern (Optional)

**Problem:** Panda3D has unreliable gamepad support on **macOS**.

**Solution:** Use pygame alongside Ursina for joystick input.

```
import pygame

pygame.init()
pygame.joystick.init()
joystick = pygame.joystick.Joystick(0) \
    if pygame.joystick.get_count() > 0 \
    else None

DEADZONE = 0.2
```

```
def update():
    if joystick is None:
        return
    pygame.event.pump()

    # Left stick: movement
    lx = joystick.get_axis(0)
    ly = joystick.get_axis(1)

    if abs(lx) > DEADZONE:
        player.position += \
          player.right * lx * 5 * time.dt
    if abs(ly) > DEADZONE:
        player.position += \
          player.forward * -ly * 5 * time.dt
```

# Gamepad Integration: Full Pattern


```python
def update():
    if not joystick: return
    pygame.event.pump()

    # Left stick → movement
    lx = apply_deadzone(js.get_axis(0))
    ly = apply_deadzone(js.get_axis(1))
    # Right stick → camera
    rx = apply_deadzone(js.get_axis(2))
    ry = apply_deadzone(js.get_axis(3))

    speed = 5 * time.dt
    player.position += player.forward * -ly * speed
    player.position += player.right * lx * speed
    player.rotation_y += rx * 2
    camera.rotation_x = clamp(
        camera.rotation_x - ry * 2, -90, 90)
```

# Exercise 3: Pick Up the Star (15 min)

## Steps

1. Open `exercises/ex4_pick_up_star/template.py`
2. Create 3 gold star entities at different positions
3. Add a score counter on the HUD ( `Text` on `camera.ui` )
4. In `update()` , check distance to each star
5. When close enough: hide star, increment score
6. Show "Well done!" when all collected

## Key Pattern: Proximity Detection

```
COLLECT_DISTANCE = 2

def update():
    for star in stars:
        if star.enabled and \
           distance(player.position,
                    star.position) \
           < COLLECT_DISTANCE:
            star.enabled = False
            score += 1
```

Same pattern for "participant reached the target" in a navigation task.

# Input Abstraction

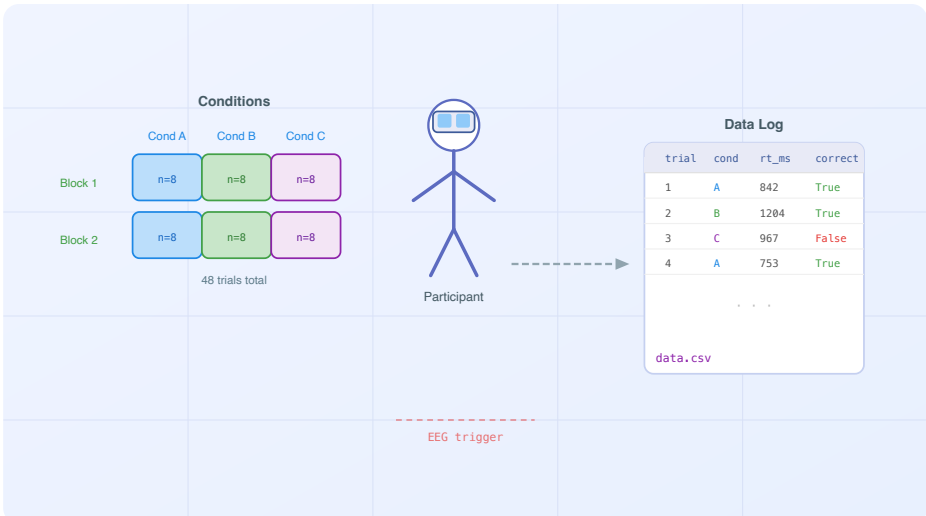
Why abstract input? So experiments work with keyboard **and** gamepad:

```
class InputManager:
    """Unified input: keyboard or gamepad."""

    def get_movement(self):
        if self.joystick:
            return (self.get_axis(0), self.get_axis(1))
        return (
            held_keys['d'] - held_keys['a'],
            held_keys['w'] - held_keys['s'],
        )

    def get_action(self, key):
        if self.joystick and self.joystick.get_button(0):
            return True
        return key == 'space'
```

Same experiment code works on desktop and in the lab.

# Module 4

## Experiment Paradigm Design



# The Experiment Layer

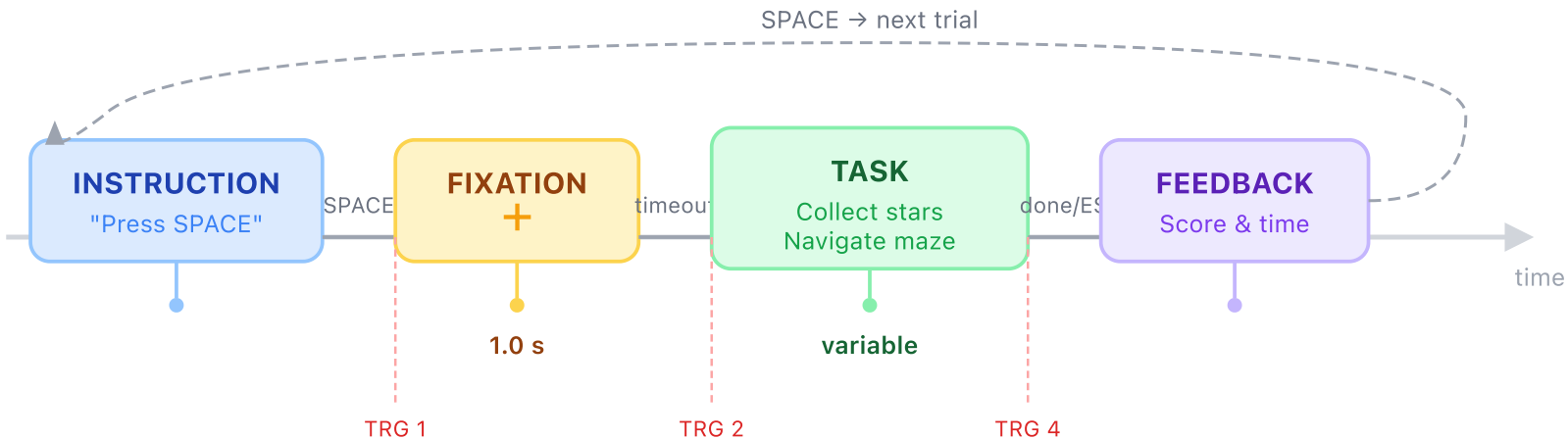
Your 3D scene is the **stimulus**. On top of it, you need:

- **State machine** – what phase of the trial are we in?
- **Trial sequencing** – condition, repeats, randomization
- **Timing** – fixation duration, response windows, ITI
- **Data logging** – what happened, when, under which condition
- **Triggers** – event markers for EEG/fMRI synchronization

> Ursina handles the *scene*. You handle the *experiment*.

# State Machine

Every trial follows a sequence of states:




Each state controls:

- What the participant **sees** (text, scene, fixation cross)
- What **input** is accepted (SPACE to advance, ESC to skip)
- What gets **logged** (triggers, timestamps — red markers below the timeline)

# Implementing the State Machine

```
class Experiment(Entity):
    def __init__(self):
        super().__init__()
        self.state = 'INSTRUCTION'
        self.current_trial = 0
        # ... setup trials, UI, player

    def input(self, key):
        if key == 'space':
            if self.state == 'INSTRUCTION':
                self.show_fixation()
            elif self.state == 'FEEDBACK':
                self.next_trial()
        elif key == 'escape' \
                and self.state == 'TASK':
            self.end_task(completed=False)

    def update(self):
        if self.state != 'TASK':
            return
        # Check proximity, collect stars...
```

**Key Idea**

Subclassing `Entity` gives you `update()` and `input()` scoped to the experiment.

The `state` variable controls which code path runs – clean separation of concerns.

# Trial Sequencing

```
# Define conditions and build trial list
trials = []
for _ in range(2):  # 2 repeats
    trials.append(
      {'condition': 'easy', 'n_stars': 1})
    trials.append(
      {'condition': 'hard', 'n_stars': 3})

random.shuffle(trials)
```

**For real experiments:** use balanced Latin squares, blocked randomization, or counterbalancing as needed.

| Trial | Condition | Stars |
|---|---|---|
| 1 | hard | 3 |
| 2 | easy | 1 |
| 3 | easy | 1 |
| 4 | hard | 3 |

The pattern: build a list of dicts, shuffle, iterate.

# Data Logging

```python
import csv

csv_file = open('experiment_data.csv',
                'w', newline='')
writer = csv.writer(csv_file)
writer.writerow([
    'trial', 'condition', 'n_stars',
    'collected', 'duration_s', 'completed'
])

# At the end of each trial:
writer.writerow([
    trial_number,
    trial['condition'],
    trial['n_stars'],
    score,
    f"{duration:.3f}",
    completed,
])
csv_file.flush()
```

**What to record:**

- Trial number & condition
- Stimulus parameters
- Response & accuracy
- Reaction time / duration
- Timestamps

**Important**

Always `flush()` after each trial – if the program crashes, you keep the data.

# EEG Triggers

```
# Define trigger codes
TRIG_FIXATION   = 1
TRIG_TASK_START = 2
TRIG_COLLECT    = 3
TRIG_TRIAL_END  = 4

def send_trigger(code):
    """Replace with LabJack/serial
    for real experiments."""
    print(f"[TRIGGER] code={code} "
          f"at {time.time():.3f}")
```

Send at key moments:

```
send_trigger(TRIG_FIXATION)
send_trigger(TRIG_TASK_START)
send_trigger(TRIG_COLLECT)
send_trigger(TRIG_TRIAL_END)
```

**In production:** replace `print` with a hardware call (LabJack, parallel port, serial). The trigger points stay the same.

# Exercise 4: Mini Experiment (20 min)

## Steps

1. Open `exercises/ex5_mini_experiment/template.`
2. Define 2 conditions (easy/hard) x 2 repeats
3. Implement state machine: INSTRUCTION → FIXATION → TASK → FEEDBACK
4. Log each trial to CSV
5. Add mock triggers at key events
6. Run it – complete all 4 trials and check CSV

## Two Tracks

**Builder Track**

Implement from the template hints

**Runner Track**

Read the solution, modify parameters (add a condition, change timing)

# Debrief: Mapping to Real Experiments

| Workshop concept | Real experiment equivalent |
| --- | --- |
| Collect stars | Navigate to target / reach object |
| Easy vs hard condition | Set size, distance, distractor load |
| Proximity detection | Response zone / collision trigger |
| CSV data logging | Behavioral data file |
| Mock triggers | EEG/fMRI event markers |
| State machine | Trial sequence controller |

The architecture scales: swap the 3D task, keep the experiment logic.

# Q&A & Hands-on

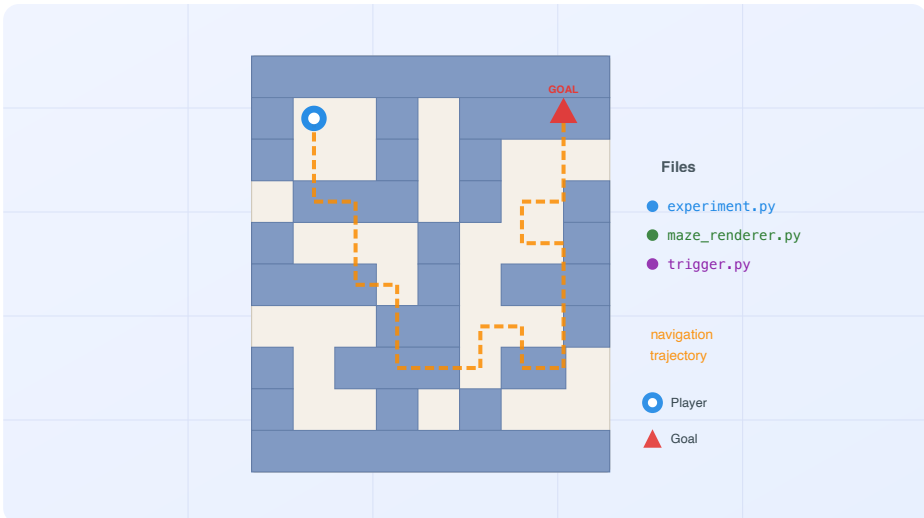
Catch up on exercises, ask questions, explore

# Lunch Break

Back at **13:00**

# Module 5

## Capstone: MazeWalker-Py



# MazeWalker-Py: Architecture

```
mazewalker-py/
├── experiment.py
├── maze_renderer.py
├── input_manager.py
├── trigger.py
├── config.yaml
├── conditions.csv
└── data/
```

- **experiment.py** – state machine + trials
- **maze_renderer.py** – Ursina scene
- **input_manager.py** – gamepad + keyboard
- **trigger.py** – EEG interface (LabJack)
- **config.yaml** – all parameters
- **conditions.csv** – trial list

Same patterns you learned today, at production scale.

# Your Code vs MazeWalker-Py

| Your Exercise 5 | MazeWalker-Py |
|---|---|
| State machine in one file | Separate `experiment.py` |
| Hard-coded conditions | External `conditions.csv` |
| Room from code | `maze_renderer.py` reads layout data |
| `print()` triggers | `trigger.py` with LabJack driver |
| Keyboard only | `input_manager.py` (keyboard + pad) |
| CSV logging inline | Dedicated logging with trajectories |

**The patterns are identical.** The production version separates concerns into files and uses configuration files instead of hard-coded values.

# Key Files Walkthrough

**experiment.py** – the state machine:

```
class MazeExperiment(Entity):
    states = ['INSTRUCTION', 'FIXATION',
              'NAVIGATION', 'FEEDBACK',
              'DONE']
    # Same pattern as your Exercise 5
```

**maze_renderer.py** – builds from data:

```
def build_maze(layout):
    for row, col in layout:
        Entity(model='cube',
               position=(col, 0.5, row),
               texture='brick',
               collider='box')
```

**trigger.py** – hardware abstraction:

```
class Trigger:
    def send(self, code):
        if self.device:
            self.device.write(code)
        else:
            print(f"[MOCK] {code}")
```

> Each file has a single responsibility. You can test each piece independently.

# Exercise 5: Make It Yours (20 min)

## Builder Track

Pick one:

- Add a third condition (medium: 2 stars)
- Add a time limit per trial (30s, auto-end)
- Log player position every 0.5s (trajectory)
- Add a practice trial before real trials

## Runner Track

Explore and configure:

- Read through MazeWalker-Py on GitHub
- Change `config.yaml` parameters (maze size, time limits)
- Run the experiment and examine output data

# Data Visualization: Trajectory Plots

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv('trajectory.csv')

plt.figure(figsize=(8, 8))
for trial in df.trial.unique():
    t = df[df.trial == trial]
    plt.plot(t.x, t.z, alpha=0.6,
             label=f'Trial {trial}')

plt.xlabel('X position')
plt.ylabel('Z position')
plt.title('Navigation Trajectories')
plt.legend()
plt.axis('equal')
plt.show()
```

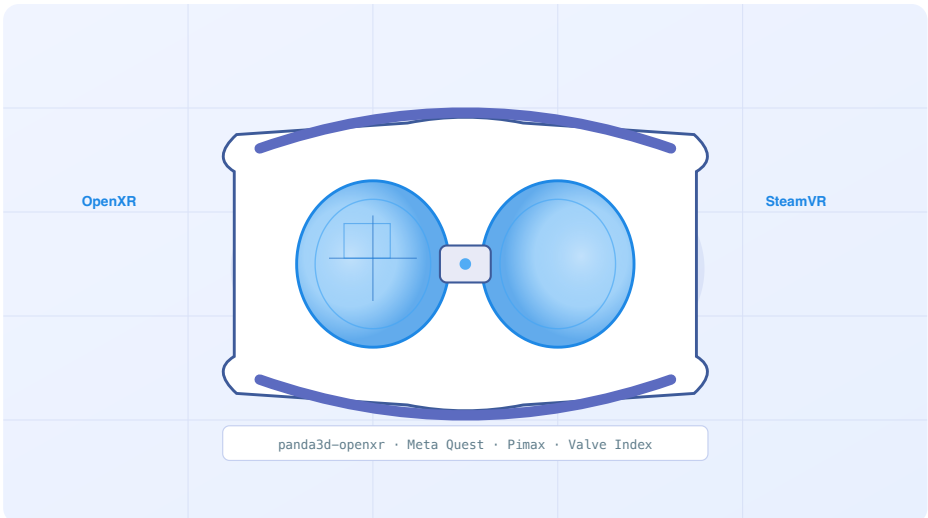
Trajectory data reveals **strategy** – not just whether they succeeded, but *how* they moved.

You can analyze:

- Path efficiency
- Hesitation points
- Strategy differences between conditions

# Module 6

## VR Roadmap & Wrap-up



# Desktop 3D vs Headset VR

## What Stays the Same

- Entity-based scene construction
- Experiment state machine
- Trial sequencing & data logging
- Trigger architecture
- Python as the main language

## What Changes

- Mouse + keyboard → **hand controllers + head tracking**
- Monitor at 60 Hz → **headset at 90 Hz** (per eye)
- `FirstPersonController` → **VR camera rig** (stereo)
- Must **manage comfort** (motion sickness)
- Need **OpenXR runtime**

~90% of your code stays the same. The scene, experiment logic, and data logging are identical.

# The VR Diff: ~20 Lines

```
# Desktop version (what you built today)
from ursina import *
app = Ursina()
player = FirstPersonController()
```

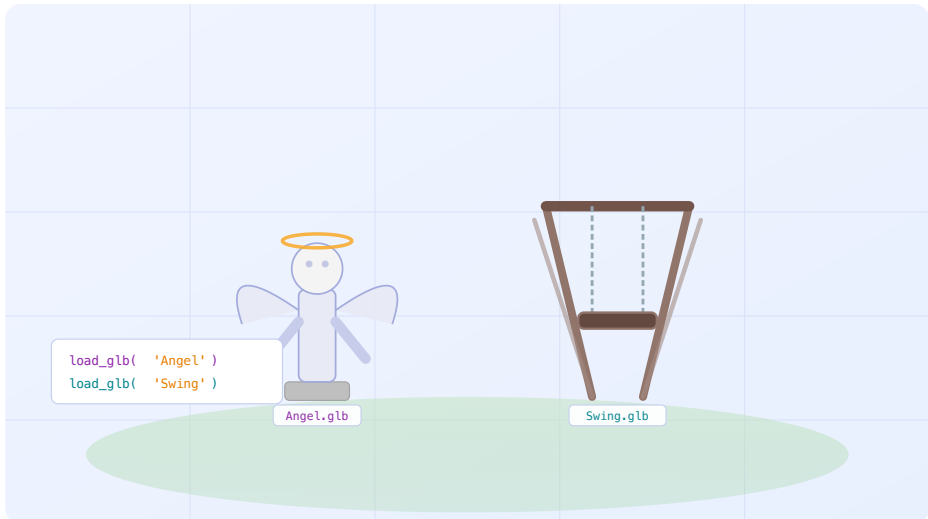
```
# VR version — pip install panda3d-openxr
from ursina import *
from p3dopenxr.p3dopenxr import P3DOpenXR

app = Ursina()
xr = P3DOpenXR()       # connects to headset
xr.init()              # starts stereo rendering

# Attach cubes to tracked hand controllers
Entity(model='cube', scale=0.1, color=color.azure,
       parent=xr.left_hand_anchor)
Entity(model='cube', scale=0.1, color=color.orange,
       parent=xr.right_hand_anchor)
```

Your room, your stars, your state machine – **unchanged**.

Works with any **OpenXR headset**: Meta Quest (via Link), HTC Vive, Valve Index, Pimax.

# Module 7

## Beyond Primitives: 3D Models



# Sourcing 3D Models

## Free Model Libraries

- **Sketchfab** – largest library, CC licensed
- **Poly Haven** – all CC0, no attribution
- **Objaverse** – 800K+ models, research use

> Prefer **.glb** (single file, textures embedded) over .obj

## AI Generation

- **Meshy.ai** – text → 3D in ~2 min, free tier
- **Tripo3D** – fast, has a Python API
- **HuggingFace TripoSR** – local, needs GPU

Pipeline: prompt → generate → preview → download .glb

> AI models for **prototyping**; curated models for final stimuli.

# Loading Models in Ursina

## Code

```
# Load a .glb model
chair = Entity(
    model='assets/chair.glb',
    scale=0.5,
    position=(2, 0, 3),
    rotation=(0, 90, 0),
    collider='box',
)

# Load a .obj model
table = Entity(
    model='assets/table.obj',
    texture='assets/table_diffuse.png',
    scale=1.0,
)
```

## Gotchas

- **Scale varies wildly** between sources – always check and adjust
- Some models face -Z – use `rotation_y=180`
- **.glb** embeds textures; **.obj** needs .mtl + texture files alongside
- Place models in an `assets/` folder

> Same Entity API you already know – just pass a file path instead of `'cube'` or `'sphere'`.

# Exercise 6: Load 3D Models (15 min)

## What You Do

1. Open `exercises/ex6_load_models/template.py`
2. Load `Angel.glb` and `Swing.glb` using `load_glb()`
3. Adjust scale, position, and rotation
4. Animate the swing with `math.sin()`
5. Add highlighting (yellow when near)

```
angel = Entity(
    model=load_glb('Angel', path=MODELS_DIR),
    scale=2, position=(-3, 0, 5),
)
```

## macOS Note

GLB models use PBR materials that need GLSL shaders macOS cannot compile. The `load_glb()` helper strips PBR shaders but keeps the base-color texture:

```
np.setShaderOff(2)   # ignore auto-shader
np.setLightOff(2)    # ignore scene lights
# keep Base Color, strip Metal Rough + Normal
```

Models render with original albedo textures. Room walls still receive lighting normally.

# Resources

## Documentation

- Ursina: ursinaengine.org
- Panda3D: panda3d.org
- pygame: pygame.org/docs
- panda3d-openxr: github.com/el-dee/panda3d-openxr
- panda3d-openvr: github.com/el-dee/panda3d-openvr

## This Workshop

- GitHub: `github.com/msenselab/vr-tutorial`
- All exercises with templates and solutions
- Slides available online
- MazeWalker-Py: `github.com/msenselab/MazeWalker-Py`
- PsychoPy-to-Ursina migration notes

# Unity VR in Research

- Unity is the industry standard for VR development
- Steeper learning curve than Ursina/Panda3D
- More complex build process for experiments
- Better support for advanced graphics, physics, and VR features

Artyom will cover Unity in the next session.

The basic tutorials we had already in Februrary. If you missed that, please check this tutorial website:

- https://eri-st.eu/VR-Workshop/en/

# Homework

Adapt the exercises for your own research question:

1. **Change the task** – replace star collection with your paradigm
2. **Add your conditions** – define meaningful experimental manipulations
3. **Design the CSV** – what dependent variables will you analyze?
4. **Add trajectory logging** – record position every N milliseconds
5. **Add 3D models** – replace primitives with realistic assets
6. **Test with a colleague** – have someone else run your experiment

> The best way to learn is to build something you actually need.

# Thank You!

## Questions?

**Chunyu Qu** · chunyu.qu@psy.lmu.de

**Zhuanghua Shi** · strongway@psy.lmu.de

LMU Munich

Workshop materials: `github.com/msenselab/vr-tutorial`

Erasmus+ KA210-VET · xr4vet.eu

Co-funded by the European Union