M2 AR VR easy steps

Stage 0: Create Project & Scene (Topics: 1, 4 – Unity Editor Interface & Layout, Creating & Managing Scenes)

Overview: This foundational stage sets up your Unity project and introduces scenes as self-contained levels or screens. Understanding scenes is core to organizing game content, as each holds GameObjects, assets, and logic.

Exact Steps:

1. Open Unity Hub.
2. Click **New Project**.
3. Select the **3D (URP)** or **3D (Built-in)** template (URP for better performance/lighting in modern projects).
4. Name the project **UnityPlaygroundLab** and choose a save location.
5. Click **Create Project**. Unity Editor opens with a default scene.

What Happens: Unity generates a new project folder with core assets (e.g., SampleScene.unity). The Hierarchy shows default objects like Main Camera (for rendering) and Directional Light (for basic lighting). This establishes the project's structure for adding scenes later via File > New Scene or Scene Management window.

Check & Pitfalls:

- **Check**: In the Project window, see SampleScene; Hierarchy lists Main Camera and Directional Light. Press Play (Ctrl+P) to preview an empty scene.
- **Pitfalls**: If the project doesn't load, ensure Unity Hub is updated. Avoid non-LTS versions for stability.

Stage 1: Editor Interface & Layout (Topic: 1 – Unity Editor Interface & Layout)

Overview: Familiarizes you with Unity's UI panels for efficient workflow. The Editor is your workspace for building, testing, and debugging—mastering it speeds up development and avoids confusion.

Exact Steps:

1. In the Editor, identify key panels:
    - **Scene View** (center): Edit 3D space; use mouse to orbit/pan (Alt+drag).
    - **Game View** (next to Scene): Preview play mode; toggle with tabs.
    - **Hierarchy** (left): List of scene objects.
    - **Inspector** (right): Edit selected object's properties/components.
    - **Project** (bottom): File browser for assets.
    - **Console** (bottom tab): View logs/errors.
2. To customize: Drag panel edges to resize; undock by dragging tabs. Go to **Window > Layouts > Default** to reset, or **Window > Layouts > Save Layout** to store your setup.

3. Practice tools: Press **W** (Move), **E** (Rotate), **R** (Scale), **T** (Rect) in Scene View. Press **F** to focus on selected objects.

What Happens: Panels interact dynamically—e.g., selecting a Hierarchy object populates the Inspector. Custom layouts persist across sessions, streamlining tasks like scene editing vs. asset management.

Check & Pitfalls:

- **Check**: Select Main Camera in Hierarchy; Inspector shows its Transform, Camera component. Play mode renders in Game View.
- **Pitfalls**: Accidentally moving the Scene camera (hold right-click to orbit properly). Use **F** if lost; reset layout if panels vanish.

Stage 2: GameObjects & Components (Topics: 2, 16 – GameObjects & Components, Scene Hierarchy & Naming Conventions)

Overview: GameObjects are scene entities (e.g., characters); Components add functionality (e.g., physics). Hierarchy organizes them like folders for readability, essential for complex scenes.

Exact Steps:

1. Create Ground: Right-click Hierarchy > **3D Object > Plane**. In Inspector Transform: Set Scale (X:4, Y:1, Z:4), Position (0,0,0).
2. Create Ramp: Right-click Hierarchy > **3D Object > Cube**. Transform: Scale (6,1,3), Position (4,0.5,0), Rotation (X:25, Y:0, Z:0).
3. Create Player: Right-click > **3D Object > Cube**. Transform: Position (0,0.5,0). In Inspector: **Add Component > Physics > Rigidbody**, then **Add Component > Box Collider**.
4. Create Crate_A: Right-click > **3D Object > Cube**. Transform: Position (2,0.5,0). Add Rigidbody and Box Collider.
5. Create Spinner_B: Right-click > **3D Object > Cylinder**. Transform: Position (0,1,5). It has a default Capsule Collider.
6. Optional organization: Right-click Hierarchy > **Create Empty**. Name it _Environment; drag Ground/Ramp as children. Repeat for _Gameplay (Player, Crate_A, Spinner_B) and _UI (empty for later).
7. Ensure Ground has a Box Collider (Add if missing via Add Component).

What Happens: GameObjects act as containers; adding Components like Rigidbody enables physics simulation. Children inherit parent transforms. Naming (e.g., Crate_A) and grouping prevent clutter in large hierarchies.

Check & Pitfalls:

- **Check**: Select Player; Inspector shows Rigidbody/Box Collider. Press Play: Player/Crate fall onto Ground (physics activates).
- **Pitfalls**: No collider on Ground? Objects fall through. Non-uniform scaling on physics objects can glitch colliders—keep scales even.

Stage 3: Transforms (Topic: 3 – Transforms: Position, Rotation, Scale)

Overview: Every GameObject has a Transform component defining its 3D pose. Mastering this is key, as all spatial manipulations (movement, animation) rely on it.

Exact Steps:

1. Select any object (e.g., Ramp) in Hierarchy.
2. In Scene View, press **W** to activate Move tool; drag arrows to adjust Position (X,Y,Z axes).
3. Press **E** for Rotate; drag arcs for Rotation (Euler angles in degrees).
4. Press **R** for Scale; drag boxes to resize uniformly.
5. For precision: In Inspector Transform section, manually edit fields (e.g., Position X: 5).
6. Practice on multiple objects, focusing with **F**.

What Happens: Transform updates world-space position/rotation/scale. Local values are relative to parents (e.g., child offset from parent). Tools provide visual gizmos; direct edits override for exact values.

Check & Pitfalls:

- **Check**: Move Player to (0,2,0); it lifts off Ground. Rotate Ramp Y:90°; it faces sideways.
- **Pitfalls**: Non-uniform scale (e.g., X:2, Y:1, Z:3) warps physics colliders oddly—use uniform for Rigidbody objects. Gimbal lock rare in Unity but avoid extreme rotations.

Stage 4: Create Trigger Zone (Topics: 2, 11 – GameObjects & Components, Trigger Events: OnTriggerEnter/OnTriggerExit)

Overview: Triggers detect overlaps without physical collision, ideal for zones (e.g., power-ups). This builds on GameObjects by adding non-physical interaction.

Exact Steps:

1. Right-click Hierarchy > **Create Empty**. Name it TriggerZone; Transform: Position (-5,0,0).
2. Right-click TriggerZone > **3D Object > Cube**; name child TriggerArea. Transform: Scale (2,1,2).
3. In Inspector for TriggerArea: Select Box Collider component > Check **Is Trigger**.
4. Create a material: Project window > Right-click > Create > Material. Name it TriggerMat; drag to TriggerArea's Mesh Renderer. Set Albedo color (e.g., yellow) for visibility.

What Happens: Is Trigger makes the collider "ghostly"—objects pass through, but OnTriggerEnter/Exit fires if a Rigidbody overlaps. The empty parent organizes the zone.

Check & Pitfalls:

- **Check**: Play mode; move Player (manually via Transform) into/out of TriggerArea—no bounce, but later scripts will log events.
- **Pitfalls**: No Rigidbody on entering object? No trigger events. Test with Debug.Log in a temp script attached to TriggerArea.

Stage 5: Scripting Basics (Topic: 5 – Intro to C# Scripting in Unity)

Overview: Scripts (C# classes inheriting MonoBehaviour) add custom logic via lifecycle methods (Awake for init, Start for setup, Update for per-frame). This bridges Editor setup to interactive behavior.

Exact Steps:

1. In Project window: Right-click > Create > Folder; name it Scripts.
2. Double-click a script (or create new: Right-click Scripts > Create > C# Script). Unity opens in your code editor.
3. For now, copy-paste Appendix scripts into new files (e.g., PlayerController.cs). Save and return to Unity—it auto-compiles.
4. Attach: Drag script from Project to target GameObject in Hierarchy (e.g., later stages).

What Happens: MonoBehaviour hooks into Unity's loop: Awake runs on load, Start before first Update, etc. Compilation errors show in Console—fix syntax for green status.

Check & Pitfalls:

- **Check**: Create a test script with void Start() { Debug.Log("Hello"); }; attach to Player. Play: Console shows "Hello".
- **Pitfalls**: Syntax errors (e.g., missing semicolon) halt compilation—check Console. Use [RequireComponent(typeof(Rigidbody))] to auto-add dependencies.

Stage 6: Movement, Update vs FixedUpdate, Frame-rate Independence (Topics: 6, 7, 8 – Update() vs FixedUpdate(), Keyboard Input, Frame-rate Independent Movement: Time.deltaTime)

Overview: Scripts handle input and physics. Update() for input (variable frames), FixedUpdate() for physics (fixed timestep). Time.deltaTime scales movement for consistent speed across hardware.

Exact Steps:

1. Create PlayerController.cs in Scripts (copy from Appendix). Set public fields in Inspector: moveSpeed=6, jumpForce=5, maxSpeed=10.
2. Drag PlayerController to Player GameObject.
3. Optional Camera follow: Create CameraFollow.cs (Appendix). Drag to Main Camera; drag Player to "Target" field in Inspector. Make Main Camera child of Player (drag in Hierarchy).
4. Play and test.

What Happens: Update() reads Input.GetAxisRaw for WASD (normalized vector), checks Space for jump (AddForce impulse). FixedUpdate() applies velocity changes (forces) and clamps speed. Time.deltaTime multiplies rotations/translations for FPS independence. LateUpdate() in CameraFollow lerps position post-frame.

Check & Pitfalls:

- **Check**: Play: WASD moves Player smoothly, Space jumps (grounded check via velocity.y). Camera follows without jitter.

- **Pitfalls**: Direct transform.position in Update fights physics—use AddForce in FixedUpdate. High FPS without deltaTime causes fast movement.

Stage 7: Rigidbody & Colliders (Topic: 9 – Rigidbody & Colliders)

Overview: Rigidbody simulates real physics (gravity/forces); Colliders define shape. Static colliders (no Rigidbody) are efficient for environments.

Exact Steps:

1. Confirm: Player and Crate_A have Rigidbody (mass=1 default) + Collider.
2. Ground: Only Box Collider (no Rigidbody)—set as static.
3. In Inspector Rigidbody: Uncheck "Use Gravity" if unwanted; adjust Drag for air resistance.

What Happens: Rigidbody integrates forces over time; collisions resolve via impulse. At least one Rigidbody per pair enables OnCollisionEnter. Static colliders don't move, saving CPU.

Check & Pitfalls:

- **Check**: Play: Player/Crate rest on Ground with gravity; push Crate (manually) and it rolls realistically.
- **Pitfalls**: Both objects without Rigidbody? No physics. Overlapping colliders pre-play cause penetration—reset positions.

Stage 8: Collisions & Triggers + UI Feedback (Topics: 10, 11, 15, 20 – Collision Detection: OnCollisionEnter, Trigger Events: OnTriggerEnter/OnTriggerExit, UI Feedback on Events, Debugging & Console Messages)

Overview: Events like OnCollisionEnter detect impacts; triggers for overlaps. UI (TMP) and Debug.Log provide runtime feedback for testing.

Exact Steps:

1. Hierarchy > Right-click > **UI > Text - TextMeshPro**. Name it HUD_Text1; position bottom-right (drag anchors in RectTransform).
    - This auto-creates HUD_Canvas (rename if needed).
2. Create CollisionReporter.cs (Appendix). Drag to Player; assign HUD_Text1 to "Hud" field.
3. Create TriggerZone.cs (Appendix). Drag to TriggerArea; in Inspector, drag objects (e.g., Crate_A, Spinner_B) to "Color Targets" array. Set Enter Color (green), Exit Color (white).
4. Select Player in Hierarchy; Inspector > Tag dropdown > Add Tag... > Create "Player" > Assign to Player.

What Happens: OnCollisionEnter logs name and updates TMP text with bold markup. OnTriggerEnter/Exit checks tag, changes material.color on targets. Debug.Log outputs to Console; TMP renders rich text.

Check & Pitfalls:

- **Check**: Play: Collide with Crate_A → Console log + HUD "Collision with Crate_A". Enter TriggerArea → Targets green, HUD "Entered trigger..."; exit → white + "Exited...".

- **Pitfalls**: No tag? Trigger ignores non-Player. Missing Rigidbody? No events. TMP missing? Import Essentials.

Stage 9: Object Interaction: Move/Rotate/Color (Topic: 12 – Object Interaction: Move, Rotate, Color Change)

Overview: Scripts enable dynamic behaviors like rotation or oscillation. Combined with triggers, they create interactive responses.

Exact Steps:

1. Create Rotator.cs (Appendix). Drag to Spinner_B; set eulerPerSecond (0,90,0) for Y-axis spin.
2. Create MoverPingPong.cs (Appendix). Drag to Crate_A; set localAxis (1,0,0) for X-movement, distance=2, speed=1.
3. Use existing TriggerZone.cs: Ensure colorTargets include Spinner_B/Crate_A.
4. Optional challenge: Edit Rotator.cs—add bool isRotating=true; in Update(), if(!isRotating) return;. Toggle via TriggerZone OnTriggerEnter (find Rotator via GetComponent, set false/true).

What Happens: Update() in Rotator applies Rotate(euler * deltaTime, Space.World) for smooth spin. MoverPingPong uses Sin(Time.time) for ping-pong oscillation. Trigger manipulates Renderer.material.color.

Check & Pitfalls:

- **Check**: Play: Spinner rotates 90°/sec; Crate slides back-forth 2 units. Enter trigger → colors change while inside.
- **Pitfalls**: Space.Self vs. World? Use World for absolute rotation. For toggle: Ensure script finds via FindObjectOfType<Rotator>() on same object.

Stage 10: TextMesh Pro & Dynamic Text (Topics: 13, 14, 15 – Intro to TextMesh Pro (TMP), Displaying Dynamic Text, UI Feedback on Collision/Trigger)

Overview: TMP renders high-quality, markup-enabled UI text (e.g., **bold**). Dynamic updates provide score/HUD feedback, essential for player info.

Exact Steps:

1. Hierarchy > Right-click > **UI > Canvas**; set Render Mode: Screen Space - Overlay.
2. Right-click Canvas > **UI > Text - TextMeshPro**. Name TMP_ScoreText; position top-left (RectTransform anchors).
3. Repeat for TMP_InfoText (top-right).
4. Right-click Canvas > **Create Empty**; name HUD_Controls. Create TextHud.cs (Appendix); drag to HUD_Controls. In Inspector, drag TMP_ScoreText to "Score Text", TMP_InfoText to "Info Text".
5. Optional: Create ScorePickup.cs (Appendix); drag to Crate_A, set value=5.

What Happens: Start() initializes text. SetInfo/AddScore update TMP_Text.text with markup. Collision in ScorePickup finds TextHud via FindFirstObjectByType, adds score, impulses Crate upward.

Check & Pitfalls:

- **Check**: Play: HUD shows "Welcome! WASD...". Collide Crate → Score +5, Info "Picked up: +5".
- **Pitfalls**: TMP shows "Missing"? Re-import Essentials. Canvas not overlay? Text scales wrong—set to Overlay for screen-fixed UI.

Stage 11: Naming & Hierarchy Conventions (Topic: 16 – Scene Hierarchy & Naming Conventions)

Overview: Consistent naming/hierarchy scales projects—prefixes group logically, reducing search time in teams.

Exact Steps:

1. Rename objects descriptively: e.g., Player → Player_Cube, Crate_A stays.
2. Use prefixes: Ensure _Environment (statics), _Gameplay (dynamics), _UI (Canvases).
3. Scripts: PascalCase (e.g., PlayerController.cs).
4. Drag related objects under empty parents (e.g., all UI under _UI).

What Happens: Hierarchy search/filter uses names; parents apply transforms hierarchically. Clean structure aids Scene view navigation.

Check & Pitfalls:

- **Check**: Search "Crate" in Hierarchy filter—finds Crate_A instantly. Expand _Gameplay—sees children organized.
- **Pitfalls**: Inconsistent names (e.g., cube1, Cube_One) confuse collaborators. Avoid deep nesting (>5 levels) for performance.

Stage 12: Build to Executable (Topics: 17, 18, 19 – Exporting & Building (.exe), Application Quit on ESC Key, Testing Executable Builds)

Overview: Builds package for standalone playtesting. Verifies Editor-only features work externally; ESC quit enhances UX.

Exact Steps:

1. File > **Build Settings**. Click **Add Open Scenes** (adds SampleScene).
2. Select **PC, Mac & Linux Standalone** > **Windows** (or your OS).
3. Click **Player Settings...** (gear icon): Company/Product Name as desired; close.
4. Click **Build**; choose/create Builds folder; name .exe (e.g., Playground.exe).
5. Create QuitOnEsc.cs (Appendix); drag to Canvas (always-active).
6. Build again if needed. Run .exe from file explorer.

What Happens: Build compiles to native executable, bundling assets/scripts. QuitOnEsc checks #if UNITY_EDITOR for Editor stop vs. Application.Quit() in build. ESC input triggers it.

Check & Pitfalls:

- **Check**: Run .exe: Movement/UI/physics work; ESC closes. Console inaccessible—use in-game HUD for debug.
- **Pitfalls**: Scenes not added? Black screen. Build fails? Check Console errors. Test on target OS/hardware for compatibility.

Practice iteratively—save often (Ctrl+S). For extensions (e.g., windmill), build on Rotator.cs. This lab covers 20 fundamentals; revisit Unity Docs for depth.

Appendix — Scripts

PlayerController.cs

```
using UnityEngine;

[RequireComponent(typeof(Rigidbody))]
public class PlayerController : MonoBehaviour
{
    public float moveSpeed = 6f;        // WASD movement
    public float jumpForce = 5f;        // Space jump
    public float maxSpeed = 10f;        // Clamp velocity

    Rigidbody rb;
    Vector3 input;

    void Awake() => rb = GetComponent<Rigidbody>();

    void Update() // Input & non-physics
    {
        float h = Input.GetAxisRaw("Horizontal");
        float v = Input.GetAxisRaw("Vertical");
        input = new Vector3(h, 0f, v).normalized;

        if (Input.GetKeyDown(KeyCode.Space) && Mathf.Abs(rb.velocity.y
) < 0.05f)
            rb.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
    }

    void FixedUpdate() // Physics
    {
        Vector3 targetVel = input * moveSpeed;
        Vector3 vel = rb.velocity;
        Vector3 velChange = (targetVel - new Vector3(vel.x, 0f, vel.z)
);
        rb.AddForce(velChange, ForceMode.VelocityChange);

        Vector3 horiz = new Vector3(rb.velocity.x, 0f, rb.velocity.z);
        if (horiz.magnitude > maxSpeed)
        {
            Vector3 clamped = horiz.normalized * maxSpeed;
            rb.velocity = new Vector3(clamped.x, rb.velocity.y, clampe
d.z);
        }
    }
}
```

CameraFollow.cs

```
using UnityEngine;

public class CameraFollow : MonoBehaviour
{
    public Transform target;
    public Vector3 offset = new Vector3(0f, 6f, -8f);
    public float smooth = 8f;

    void LateUpdate()
    {
        if (!target) return;
        transform.position = Vector3.Lerp(transform.position, target.p
osition + offset, smooth * Time.deltaTime);
        transform.LookAt(target);
    }
}
```

CollisionReporter.cs

```
using UnityEngine;

public class CollisionReporter : MonoBehaviour
{
    public TextMeshProUGUI hud; // assign in Inspector

    void OnCollisionEnter(Collision c)
    {
        Debug.Log($"Player collided with {c.gameObject.name}");
        if (hud) hud.SetInfo($"Collision with <b>{c.gameObject.name}</
b>");
    }

    void OnTriggerEnter(Collider other)
    {
        Debug.Log($"Entered trigger: {other.gameObject.name}");
        if (hud) hud.SetInfo($"Entered trigger: <b>{other.gameObject.n
ame}</b>");
    }

    void OnTriggerExit(Collider other)
    {
        Debug.Log($"Exited trigger: {other.gameObject.name}");
        if (hud) hud.SetInfo($"Exited trigger: <b>{other.gameObject.na
me}</b>");
    }
}
```

TriggerZone.cs

```
using UnityEngine;

public class TriggerZone : MonoBehaviour
{
    public Renderer[] colorTargets; // drop objects whose color you wa
nt to change
    public Color enterColor = Color.green;
    public Color exitColor = Color.white;

    void OnTriggerEnter(Collider other)
    {
        if (!other.CompareTag("Player")) return;
        foreach (var r in colorTargets)
            if (r) r.material.color = enterColor; // color change on e
nter
    }

    void OnTriggerExit(Collider other)
    {
        if (!other.CompareTag("Player")) return;
        foreach (var r in colorTargets)
            if (r) r.material.color = exitColor; // revert on exit
    }
}
```

Rotator.cs

```
using UnityEngine;

public class Rotator : MonoBehaviour
{
    public Vector3 eulerPerSecond = new Vector3(0f, 90f, 0f);

    void Update()
    {
        transform.Rotate(eulerPerSecond * Time.deltaTime, Space.World)
;
    }
}
```

MoverPingPong.cs

```
using UnityEngine;

public class MoverPingPong : MonoBehaviour
{
    public Vector3 localAxis = Vector3.right;
    public float distance = 2f;
    public float speed = 1f;
    Vector3 start;

    void Start() => start = transform.position;

    void Update()
    {
        float t = (Mathf.Sin(Time.time * speed) + 1f) * 0.5f; // 0..1
        transform.position = start + localAxis.normalized * (t * dista
nce);
    }
}
```

TextHud.cs

```
using UnityEngine;
using TMPro;

public class TextHud : MonoBehaviour
{
    public TMP_Text scoreText; // assign TMP_ScoreText
    public TMP_Text infoText;  // assign TMP_InfoText

    int score;

    void Start()
    {
        SetScore(0);
        SetInfo("Welcome! WASD to move, Space to jump.");
    }

    public void AddScore(int delta)
    {
        score += delta;
        SetScore(score);
        Debug.Log($"Score changed to {score}");
    }

    public void SetInfo(string message)
```

```
    {
        if (infoText) infoText.text = message;
    }

    void SetScore(int s)
    {
        if (scoreText) scoreText.text = $"Score: <b>{s}</b>";
    }
}
```

ScorePickup.cs

```
using UnityEngine;

public class ScorePickup : MonoBehaviour
{
    public int value = 5;

    void OnCollisionEnter(Collision c)
    {
        var hud = FindFirstObjectByType<TextHud>();
        if (hud)
        {
            hud.AddScore(value);
            hud.SetInfo($"Picked up: +{value}");
        }
        if (TryGetComponent<Rigidbody>(out var rb))
            rb.AddForce(Vector3.up * 2f, ForceMode.Impulse);
    }
}
```

QuitOnEsc.cs

```
using UnityEngine;

public class QuitOnEsc : MonoBehaviour
{
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Escape))
        {
#if UNITY_EDITOR
            Debug.Log("ESC pressed: would quit app (Editor detected)."
);
            UnityEditor.EditorApplication.isPlaying = false; // stop p
laymode
#else
            Debug.Log("ESC pressed: quitting application.");
```

```
            Application.Quit();
#endif
        }
    }
}
```

Optional Extensions

- Replace cylinder with a windmill and use Rotator for blades.
- Add pickups using trigger colliders (IsTrigger) and destroy them on collect.
- Add a simple menu scene and load Playground via SceneManager.