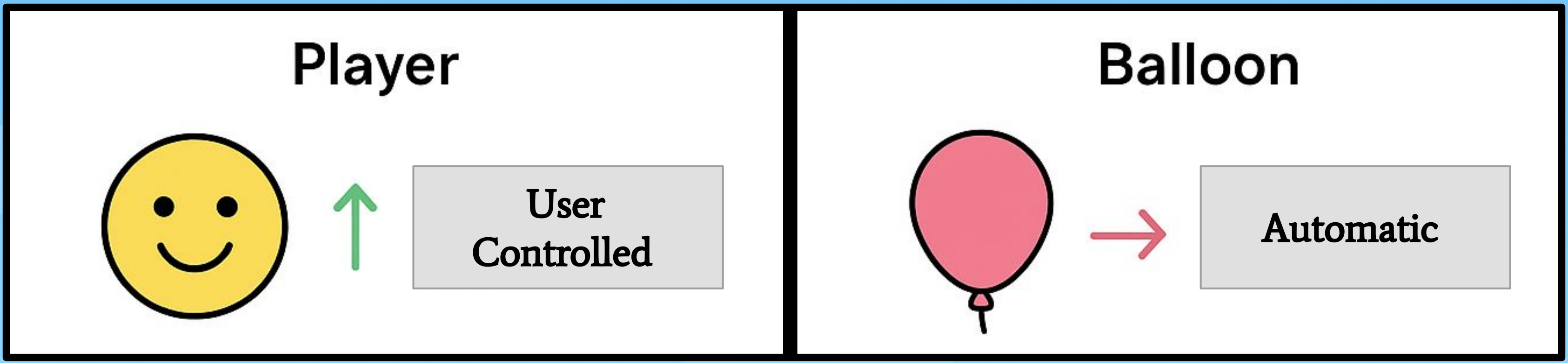
Introduction to Unity

Recap of Lab #1 & Lab #2



Player

User Controlled

Balloon

Automatic

Unity Editor Tour

- **Hierarchy**: Lists all GameObjects in the Scene; parent-child relationships matter
- **Scene View**: Visual workspace to place objects
- **Game View**: Player perspective of the game
- **Inspector**: Edit properties and components of selected objects
- **Project Window**: Asset management (scripts, prefabs, materials, audio)
- **Console**: Shows errors, warnings, and debug logs

Scenes & GameObjects

- **Scene** = a container for your level or environment
- **GameObject** = fundamental entity in Unity, can hold Components
- **Empty GameObject** = useful for organizing or as a parent
- Parent-Child hierarchy allows grouping and transforms inheritance

Components

- Components add behavior or properties to GameObjects
- Key examples:
  - **Transform** → position, rotation, scale (every GameObject has it)
  - **MeshRenderer** → renders object geometry visible in Scene
  - **Collider** → defines object boundaries for physics
  - **Rigidbody** → makes object interact with physics engine
  - **Script** → custom behavior using C#

First Script Example

- Create C# script and attach to a GameObject
- Example: Rotating a cube:

```
using UnityEngine;

public class Rotator : MonoBehaviour {
    void Update() {
        transform.Rotate(0, 100 * Time.deltaTime,
0);
    }
}
```

Unity Script Lifecycle

- **Awake()** → called when the object is initialized
- **Start()** → runs once before the first frame update
- **Update()** → called once per frame, use for dynamic behavior
- **FixedUpdate()** → called at fixed intervals, use for physics updates
- **OnCollisionEnter()/OnTriggerEnter()** → respond to collisions

Prefabs

- **Prefab** = reusable template of a GameObject
- **Benefits**: consistency across instances, central updates, easy duplication
- Use for enemies, pickups, projectiles, UI elements

Physics & Collisions

- **Rigidbody** → enables physics simulation (gravity, forces)
- **Collider** → defines the shape of the object for collision detection
- **Static Collider** = stationary object
- **Dynamic Collider** = moves with Rigidbody

Unity Labs #1 and #2 Recap

- **Unity Lab 1: Getting Started**
    - Familiarize with the Unity Editor and basic scene setup
    - Set Up Background
- **Unity Lab 2: Movement and User Input**
    - Prepare sprites for movement and interaction
    - Implement user controls for sprite movement (player)
    - Create an autonomous moving object (balloon)
    - Enhance visual feedback by flipping sprites
    - Keep the player within the visible screen area

Add a Background Image

- Goal: **Place a static image behind everything (like a sky or gradient).**
  - Option A: **Using a Sprite**
  - Option B: **Using the Camera**

Option A: Using a Sprite

- In your Assets folder, right-click → Create → Sprites → Square (or import your own image).
- Rename it Background.
- Drag it into the Scene view.
- Set Scale large enough to fill the screen (e.g., 20×20).
- In the Sprite Renderer:
  - Change Color or assign a background sprite texture.
  - Set Order in Layer = -10 (so it renders behind the player and balloon).

Option B: Using the Camera

- Select the Main Camera.
- In the Camera component, change Background Color to a gradient or solid sky color.
- Simpler backdrop with no extra objects.

Creating Sprite Renderer (sr) and Rigidbody2D (rb)

- Many objects in Unity need a Sprite Renderer and a Rigidbody2D to display correctly and interact with physics.
- Add a Sprite Renderer component:
  - Component → Rendering → Sprite Renderer
  - Assign a sprite image to Sprite.
- Add a Rigidbody2D component:
  - Component → Physics 2D → Rigidbody2D
  - Set Body Type to Dynamic for moving objects

Creating Sprite Renderer (sr) and Rigidbody2D (rb)

```
private SpriteRenderer sr;
private Rigidbody2D rb;


void Awake() {
    sr = GetComponent<SpriteRenderer>();
    rb = GetComponent<Rigidbody2D>();
}
```

Direction Vectors

- Direction vectors (Vector2 or Vector3) control where the object moves.
- For the player, moveInput is a vector determined by keyboard input:

```
private Vector2 moveInput;
```

- For the balloon, direction is vector that starts moving right:

```
private Vector3 direction = Vector3.right;
```

Automatic Movement (Balloon)

- In BalloonMovement.cs, we move the balloon automatically using:

```
transform.Translate(direction * speed * Time.deltaTime);
```

- Translate moves the object in the direction vector at a given speed.

- And we can get the ensure that it does not go off screen by using the main camera:

```
Camera cam = Camera.main;
```

Detecting Screen Edges

- To prevent objects from leaving the screen, we calculate the camera edges:

  ```
  Vector3 rightEdge = cam.ViewportToWorldPoint(new
  Vector3(1f, 0.5f, camDistance));
  ```

  ```
  Vector3 leftEdge  = cam.ViewportToWorldPoint(new
  Vector3(0f, 0.5f, camDistance));
  ```

- ViewportToWorldPoint converts camera coordinates (0–1) to world coordinates.
- Then we check the object’s position and flip its direction if it reaches an edge.

Flipping Sprites

- Sprites face one direction by default.
- To make them "turn around," we flip them horizontally using:

```
sr.flipX = true;  // or false
```

- Used in both the balloon and player to visually match movement.

Reading Keyboard Input

- In `PlayerMovement.cs`, we read keys directly:

```
var keyboard = Keyboard.current;
if (keyboard.wKey.isPressed ||
keyboard.upArrowKey.isPressed)
    moveInput.y += 1;
```

- Allows movement with WASD or arrow keys without extra setup.

Normalizing Movement

- When moving diagonally, the combined vector `(x + y)` can be longer than 1, causing a diagonal speed boost.

- Normalize to keep movement consistent in all directions:

  ```
  moveInput = moveInput.normalized;
  ```

Clamping Player to Camera Bounds

- Prevents the player from moving off-screen using `Mathf.Clamp`:

  ```
  newPos.x = Mathf.Clamp(newPos.x,
  leftBound, rightBound);
  newPos.y = Mathf.Clamp(newPos.y,
  bottomBound, topBound);
  ```

- Ensures the player always stays visible.

Summary / Takeaways

- **Automatic movement** uses `Translate` + `direction` vector.
- **Player input movement** uses keyboard + `normalized` vector + `Rigidbody2D`.
- **Sprite flipping** and **clamping** make the objects look correct and stay on-screen.
- **Camera viewport conversion** allows dynamic edge detection.
- **Awake()** vs **Start()** vs **Update()** vs **FixedUpdate()** helps organize initialization, input, and physics logic.

Additional Suggestions

- Modify `speed` or `direction` vectors for the balloon.
- Add vertical bouncing to the balloon using `Translate`.
- Restrict player to a smaller area using `Mathf.Clamp`.
- Experiment with flipping and `Rigidbody2D` constraints.