SANDBOX BEGINNER TUTORIAL SERIES

PART 1 :CHARACTER MOVEMENT BASIC VECTORS

by :BLASTIPEDE

Overview:

This PDF walks you through an extended Unity 2D player controller. We start with a simple movement controller and then add: dash, sprite flipping, procedural squash & stretch, and an afterimage trail for dashes. Every important line of code is explained in plain language — like a GeeksForGeeks line-by-line breakdown.

Player sprite (attached):

Part 1 — Simple 2D Rigidbody Movement (Base Controller)

We begin with a minimal controller that reads keyboard input and moves a Rigidbody2D. This is the foundation. Read the code first, then the line-by-line explanations.

```
using UnityEngine;

public class PlayerMovement : MonoBehaviour
{
    [Header("Movement Settings")]
    public float moveSpeed = 5f;

    private Rigidbody2D rb;
    private Vector2 moveInput;

    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        rb.gravityScale = 0;
    }

    void Update()
    {
        moveInput.x = Input.GetAxisRaw("Horizontal");
        moveInput.y = Input.GetAxisRaw("Vertical");

        moveInput.Normalize();
    }

    void FixedUpdate()
```

```
    {
        rb.velocity = moveInput * moveSpeed;
    }
}
```

Line-by-line explanation — Base Controller

```
using UnityEngine;
```

Imports Unity's core library so you can use classes like MonoBehaviour, Rigidbody2D, Vector2, and Input.

```
public class PlayerMovement : MonoBehaviour
```

Declares a new component class. 'MonoBehaviour' lets Unity call special methods (Start, Update, etc.) and lets you attach this script to GameObjects.

```
{
```

Start of the class body — where we declare variables and methods.

```
    [Header("Movement Settings")]
```

Editor-only attribute. This displays a labeled header above the next variable in the Unity Inspector to organize fields visually.

```
    public float moveSpeed = 5f;
```

A public float that sets how fast the player moves. Because it is public, Unity shows it in the Inspector where you can tweak the value at runtime. '5f' is the default value (f means float).

```
    private Rigidbody2D rb;
```

Private variable to store a reference to the Rigidbody2D component (used for physics movement). 'private' means the Inspector won't show it.

```
    private Vector2 moveInput;
```

Holds the player's input direction as an (x,y) vector. Vector2 stores two floats.

```
    void Start()
```

Start() is called once by Unity when the object becomes active (before the first frame). Use it to cache components and do initialization.

```
    {
```

Start of the Start method.

```
        rb = GetComponent<Rigidbody2D>();
```

Finds the Rigidbody2D attached to the same GameObject and stores it in 'rb'. If no Rigidbody2D exists, 'rb' becomes null (which causes errors when used).

```
        rb.gravityScale = 0;
```

Sets the Rigidbody2D's gravity scale to zero so the character won't fall — useful for top-down movement.

```
    }
```

End of Start.

```
    void Update()
```

Update() runs every frame (frame-rate dependent). It's a good place to read player input because input checks should happen each frame.

```
    {
```

Start of Update.

```
        moveInput.x = Input.GetAxisRaw("Horizontal");
```

Reads horizontal input: A/D or Left/Right arrows (returns -1, 0, or 1 for keyboard). 'GetAxisRaw' is unsmoothed (instant changes).

```
        moveInput.y = Input.GetAxisRaw("Vertical");
```

Reads vertical input: W/S or Up/Down arrows.

```
        moveInput.Normalize();
```

Normalizes the vector to length 1 if it's non-zero. This prevents diagonal movement being faster (diagonal = sqrt(2) if unnormalized).

```
    }
```

End of Update.

```
    void FixedUpdate()
```

FixedUpdate() runs on a fixed timestep (default 0.02s). Use it for physics updates like changing Rigidbody velocity or applying forces.

```
    {
```

Start of FixedUpdate.

```
        rb.velocity = moveInput * moveSpeed;
```

Sets the Rigidbody's velocity directly. 'moveInput' is the direction, 'moveSpeed' scales the magnitude (units per second).

```
    }
```

End of FixedUpdate.

```
}
```

End of the class.

Part 2 — Adding Dash (step-by-step)

We now extend the base controller to add a dash: a short burst of high speed with a cooldown. We'll show only the additions and then explain each new line.

```
// DASH: variables and state
public float dashSpeed = 15f;
public float dashDuration = 0.2f;
public float dashCooldown = 1f;
private bool isDashing = false;
private float dashTimer;
private float dashCooldownTimer;
```

```
public float dashSpeed = 15f;
```

How fast the player moves while dashing. Usually much larger than moveSpeed to feel like a burst.

```
public float dashDuration = 0.2f;
```

How long the dash lasts in seconds. 0.2s is a short quick dash.

```
public float dashCooldown = 1f;
```

Time after a dash before the player can dash again. Prevents spamming.

```
private bool isDashing = false;
```

Flag tracking whether the player is currently dashing. We use it to change behavior while dashing.

```
private float dashTimer;
```

Counts down the remaining time of the current dash.

```
private float dashCooldownTimer;
```

Tracks cooldown time before the next dash can be used.

```
// inside Update()
if (Input.GetKeyDown(KeyCode.Space) && !isDashing && dashCooldownTimer <= 0f && moveInput.magnitude > 0.1f)
{
    StartDash();
}

if (isDashing)
{
    dashTimer -= Time.deltaTime;
    if (dashTimer <= 0f) EndDash();
}
else
{
    dashCooldownTimer -= Time.deltaTime;
    if (dashCooldownTimer < 0f) dashCooldownTimer = 0f;
}
```

```
Input.GetKeyDown(KeyCode.Space)
```

Checks if the Space key was just pressed this frame (ideal for single-press actions).

```
&& !isDashing
```

Make sure we aren't already dashing — we don't want to restart a dash mid-dash.

```
&& dashCooldownTimer <= 0f
```

Ensure cooldown finished before starting another dash.

```
&& moveInput.magnitude > 0.1f
```

Optional: require that the player is pressing a direction to dash (prevents dashing in place).

```
StartDash();
```

Sets dash state and timers (we define this method next).

```
dashTimer -= Time.deltaTime;
```

Counts down dash duration every frame while dashing. Time.deltaTime is the seconds since last frame.

```
if (dashTimer <= 0f) EndDash();
```

When the dash time runs out, stop dashing and reset necessary values.

```
dashCooldownTimer -= Time.deltaTime;
```

When not dashing, count down the cooldown timer each frame until it reaches zero.

```
// inside FixedUpdate()
if (isDashing)
{
    rb.velocity = moveInput * dashSpeed;
}
else
{
    rb.velocity = moveInput * moveSpeed;
}
```

```
rb.velocity = moveInput * dashSpeed;
```

While dashing, set velocity based on dashSpeed so the player moves faster.

```
rb.velocity = moveInput * moveSpeed;
```

When not dashing, use regular move speed.

```
void StartDash()
{
    isDashing = true;
    dashTimer = dashDuration;
    dashCooldownTimer = dashCooldown;
}

void EndDash()
{
    isDashing = false;
}
```

Explanation:

StartDash(): sets isDashing true and initializes the dashTimer to the configured dashDuration. It also starts the cooldown timer so the next dash can't happen immediately. EndDash(): stops the dash by clearing the flag. You could also reset velocities or play effects here.

Part 3 — Sprite Flipping and Procedural Squash & Stretch

To make the chicken feel alive, we'll flip the sprite when changing horizontal direction and apply a simple squash & stretch by changing the Transform's scale while moving or dashing.

```
private SpriteRenderer sr;
private Vector3 originalScale;

public float moveSquashAmount = 0.9f;
public float moveStretchAmount = 1.1f;
public float dashSquash = 1.2f;
public float dashStretch = 0.8f;
```

```
private SpriteRenderer sr;
```

A reference to the SpriteRenderer so we can flip the sprite horizontally (sr.flipX).

```
private Vector3 originalScale;
```

Stores the original local scale so we can return to it when idle.

```
moveSquashAmount / moveStretchAmount
```

Multipliers used when the player is walking/running to slightly squash horizontally and stretch vertically for a bouncy feel.

```
dashSquash / dashStretch
```

More pronounced multipliers used during a dash (wide and short).

```
void Start()
{
    rb = GetComponent<Rigidbody2D>();
    rb.gravityScale = 0;
    rb.constraints = RigidbodyConstraints2D.FreezeRotation;

    sr = GetComponent<SpriteRenderer>();
    originalScale = transform.localScale;
}
```

Explanation: we now cache the SpriteRenderer in Start() and save the original scale. Freezing rotation prevents physics from spinning the sprite on collisions.

```
// flip sprite horizontally
if (moveInput.x != 0)
{
    sr.flipX = moveInput.x < 0;
}
```

Explanation: if the player is pressing left or right, set sr.flipX to true when moving left (moveInput.x < 0). This visually mirrors the sprite.

```
// apply squash & stretch based on movement state
if (isDashing)
{
    transform.localScale = new Vector3(originalScale.x * dashSquash, originalScale.y * dashStretch, originalScale.z);
}
else
{
    if (rb.velocity.magnitude > 0.1f)
    {
        transform.localScale = new Vector3(originalScale.x * moveSquashAmount, originalScale.y * moveStretchAmount, o
    }
    else
    {
        transform.localScale = originalScale;
    }
}
```

Explanation: while dashing we apply dashScale multipliers (wide and short). While moving normally, we apply subtler squash/stretch. When idle we restore original scale.

Part 4 — Afterimages for Dash Trail

Afterimages are ghost copies of the sprite that fade out, creating a trail effect during the dash. We'll create a small prefab and spawn it periodically while dashing.

```
public GameObject afterImagePrefab;
public float afterImageSpacing = 0.05f;
private float afterImageTimer;
```

```
public GameObject afterImagePrefab;
```

A prefab reference to the afterimage object (a GameObject with a SpriteRenderer and AfterImage script). Assign it in the Inspector.

```
public float afterImageSpacing = 0.05f;
```

Interval in seconds between spawning afterimages while dashing. Smaller = denser trail.

```
private float afterImageTimer;
```

Internal timer counting down between spawns.

```
void CreateAfterImage()
{
    if (afterImagePrefab == null) return;

    GameObject ai = Instantiate(afterImagePrefab, transform.position, transform.rotation);
    SpriteRenderer aisr = ai.GetComponent<SpriteRenderer>();
    if (aisr != null)
    {
        aisr.sprite = sr.sprite;
        aisr.flipX = sr.flipX;
    }
}
```

Explanation: We instantiate the afterImagePrefab at the player's position, copy the current sprite and flip state so it matches, then let the prefab fade itself away.

```
// in FixedUpdate when isDashing
afterImageTimer -= Time.fixedDeltaTime;
if (afterImageTimer <= 0f)
{
    CreateAfterImage();
    afterImageTimer = afterImageSpacing;
}
```

Explanation: during dashing we decrement the afterImageTimer at the physics timestep and spawn an afterimage whenever it hits zero, then reset the timer.

AfterImage.cs — Fades and destroys the ghost sprite

```
using UnityEngine;

public class AfterImage : MonoBehaviour
{
    public float fadeSpeed = 5f;
    private SpriteRenderer sr;
    private Color color;

    void Start()
    {
        sr = GetComponent<SpriteRenderer>();
        if (sr == null)
        {
            Debug.LogWarning("AfterImage requires a SpriteRenderer.");
            Destroy(gameObject);
            return;
        }
        color = sr.color;
    }

    void Update()
    {
        color.a -= fadeSpeed * Time.deltaTime;
        sr.color = color;

        if (color.a <= 0f)
        {
            Destroy(gameObject);
        }
    }
}
```

Line-by-line explanation: The script reduces the alpha channel (color.a) every frame until it reaches zero, then destroys the GameObject. fadeSpeed controls how fast it fades.

Final Combined Script — PlayerMovement.cs (Full Extended)

```
using UnityEngine;

[RequireComponent(typeof(Rigidbody2D), typeof(SpriteRenderer))]
public class PlayerMovement : MonoBehaviour
{
    [Header("Movement Settings")]
    public float moveSpeed = 5f;

    [Header("Dash Settings")]
    public float dashSpeed = 15f;
    public float dashDuration = 0.2f;
    public float dashCooldown = 1f;
    private bool isDashing = false;
    private float dashTimer;
    private float dashCooldownTimer;

    [Header("Squash & Stretch")]
    public float moveSquashAmount = 0.9f;
    public float moveStretchAmount = 1.1f;
    public float dashSquash = 1.2f;
    public float dashStretch = 0.8f;

    private Rigidbody2D rb;
    private Vector2 moveInput;
    private SpriteRenderer sr;
    private Vector3 originalScale;

    [Header("Afterimage Settings")]
    public GameObject afterImagePrefab;
    public float afterImageSpacing = 0.05f;
    private float afterImageTimer;

    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        rb.gravityScale = 0;
        rb.constraints = RigidbodyConstraints2D.FreezeRotation;

        sr = GetComponent<SpriteRenderer>();
        originalScale = transform.localScale;
    }

    void Update()
    {
        moveInput.x = Input.GetAxisRaw("Horizontal");
        moveInput.y = Input.GetAxisRaw("Vertical");
        moveInput.Normalize();

        if (moveInput.x != 0)
        {
            sr.flipX = moveInput.x < 0;
        }

        if (Input.GetKeyDown(KeyCode.Space) && !isDashing && dashCooldownTimer <= 0f && moveInput.magnitude > 0.1f)
        {
            StartDash();
        }

        if (isDashing)
        {
            dashTimer -= Time.deltaTime;
            if (dashTimer <= 0f) EndDash();
        }
        else
        {
            dashCooldownTimer -= Time.deltaTime;
            if (dashCooldownTimer < 0f) dashCooldownTimer = 0f;
```

```
        }
    }

    void FixedUpdate()
    {
        if (isDashing)
        {
            rb.velocity = moveInput * dashSpeed;

            afterImageTimer -= Time.fixedDeltaTime;
            if (afterImageTimer <= 0f)
            {
                CreateAfterImage();
                afterImageTimer = afterImageSpacing;
            }

            transform.localScale = new Vector3(originalScale.x * dashSquash, originalScale.y * dashStretch, originalS
        }
        else
        {
            rb.velocity = moveInput * moveSpeed;

            if (rb.velocity.magnitude > 0.1f)
            {
                transform.localScale = new Vector3(originalScale.x * moveSquashAmount, originalScale.y * moveStretchA
            }
            else
            {
                transform.localScale = originalScale;
            }
        }
    }

    void StartDash()
    {
        isDashing = true;
        dashTimer = dashDuration;
        dashCooldownTimer = dashCooldown;
        afterImageTimer = 0f;
    }

    void EndDash()
    {
        isDashing = false;
        transform.localScale = originalScale;
    }

    void CreateAfterImage()
    {
        if (afterImagePrefab == null) return;

        GameObject ai = Instantiate(afterImagePrefab, transform.position, transform.rotation);
        SpriteRenderer aisr = ai.GetComponent<SpriteRenderer>();
        if (aisr != null)
        {
            aisr.sprite = sr.sprite;
            aisr.flipX = sr.flipX;
        }
    }
}
```

Setup & Testing (quick steps)

1) Create a Unity 2D project.

2) Import your chicken sprite as a Sprite (Texture Type -> Sprite).

3) Create a Player GameObject, add a SpriteRenderer and set the sprite.

4) Add Rigidbody2D (Gravity Scale 0) and a collider (CircleCollider2D recommended).

5) Add PlayerMovement.cs to the Player GameObject and assign AfterImage prefab.

6) Create AfterImage prefab: a GameObject with SpriteRenderer and AfterImage.cs, set initial alpha ~0.5, and make it a prefab.

7) Tweak parameters (moveSpeed, dashSpeed, dashDuration, afterImageSpacing) to taste.

8) Play and test: WASD/Arrow keys to move, Space to dash.

Troubleshooting quick tips

Player doesn't move: Ensure Rigidbody2D component exists and script is attached. Check Console for NullReferenceException.

Dash doesn't trigger: Make sure you're pressing Space and also pressing a direction (script requires movement input). Check dashCooldown and dashDuration values.

Afterimages cause lag: Increase afterImageSpacing or implement pooling instead of Instantiate/Destroy.

Sprite flips weird: Check that sprite pivot is centered; flipX mirrors; ensure no other code changes scale negatively.

End of tutorial — happy hacking! If you want, I can also generate an object pooling version of the afterimage system (recommended for performance).