# Tutorial: Creating a "Juicy" 2D Jump Mechanic in Unity

This tutorial follows the concepts from Blackthornprod's video to create a highly responsive and satisfying 2D jump mechanic. We'll start with a basic jump and progressively add features like variable jump height, coyote time, and jump buffering to make it feel "juicy."

## Prerequisites

- A Unity 2D project.
- A "Player" GameObject with a `Rigidbody2D` (set Gravity Scale to 2-3) and a `CapsuleCollider2D` (or similar).
- A "Ground" GameObject with a `BoxCollider2D` (or `TilemapCollider2D`).
- Set up a "Ground" layer in your project (Edit -> Project Settings -> Tags and Layers) and assign your "Ground" GameObject to this layer.
- **Note on 2.5D:** This tutorial uses 2D physics (`Rigidbody2D`, `Physics2D`). Your visuals can still be 3D models; just ensure their movement is constrained to the 2D plane. This is often called a "2.5D" setup.

## Step 1: The Basic Jump & Ground Check

First, we need to make the player jump, but *only* if they are on the ground.

1. **Ground Check Setup:** In Unity, create an empty GameObject as a child of your "Player" and name it `GroundCheck`. Position it at the player's feet.
2. **Player Script:** Create a new C# script named `PlayerJump` (or similar) and add it to your Player.

Open the script and add the following code. This will check for the "Jump" button input (Spacebar by default) and apply an upward force *only* if our `GroundCheck` object is overlapping with the "Ground" layer.

using UnityEngine;

public class PlayerJump : MonoBehaviour

{

```
    private Rigidbody2D rb;

    public float jumpForce = 10f;



    // Ground Check variables

    public Transform groundCheck;

    public LayerMask groundLayer;

    private bool isGrounded;

    public float groundCheckRadius = 0.2f;



    void Start()

    {

        rb = GetComponent<Rigidbody2D>();

    }



    void Update()

    {

        // Check for ground

        isGrounded = Physics2D.OverlapCircle(groundCheck.position, groundCheckRadius,
groundLayer);
```

```
        // --- Basic Jump Input ---
        if (Input.GetButtonDown("Jump") && isGrounded)
        {
            rb.velocity = new Vector2(rb.velocity.x, jumpForce);
        }
    }

    // Helper function to visualize the ground check radius in the Scene view
    private void OnDrawGizmosSelected()
    {
        if (groundCheck == null) return;
        Gizmos.color = Color.yellow;
        Gizmos.DrawWireSphere(groundCheck.position, groundCheckRadius);
    }
}
```

**In Unity:**

- Drag your `Rigidbody2D` component into the `rb` field (or let `GetComponent` handle it as written).
- Drag the `GroundCheck` object into the `groundCheck` field.
- Set the `Ground Layer` dropdown to your "Ground" layer.
- Adjust `jumpForce` (e.g., 15) and `groundCheckRadius` (e.g., 0.2) until it feels right.

### Step 2: Variable Jump Height

This allows the player to perform a short hop by tapping the jump button or a full jump by holding it. We do this by cutting the jump short if the player releases the button while still moving upwards.

Add this code to your `Update()` method:

```
  void Update()

  {

      isGrounded = Physics2D.OverlapCircle(groundCheck.position, groundCheckRadius,
groundLayer);




      if (Input.GetButtonDown("Jump") && isGrounded)

      {

          rb.velocity = new Vector2(rb.velocity.x, jumpForce);

      }




      // --- Variable Jump Height ---

      if (Input.GetButtonUp("Jump") && rb.velocity.y > 0f)

      {
```

```
        // If the button is released while jumping, cut the upward velocity

        rb.velocity = new Vector2(rb.velocity.x, rb.velocity.y * 0.5f);

    }

}
```

### Step 3: Better Falling (Fall Multiplier)

The default Unity gravity can feel "floaty." We can make the jump feel snappier by increasing gravity when the player is falling. This creates a "fast fall."

Add new public variables to your script:

```
public float fallMultiplier = 2.5f;

public float lowJumpMultiplier = 2f;
```

1.
2. We will now use `FixedUpdate` for all physics modifications. Move the variable jump height logic here and add the fall multiplier.

```
// Update is now only for checking input

void Update()

{

    isGrounded = Physics2D.OverlapCircle(groundCheck.position, groundCheckRadius, groundLayer);



    if (Input.GetButtonDown("Jump") && isGrounded)

    {
```

```
            rb.velocity = new Vector2(rb.velocity.x, jumpForce);

        }

    }



    // FixedUpdate is for physics

    void FixedUpdate()

    {

        // --- Better Falling Logic ---

        if (rb.velocity.y < 0)

        {

            // We are falling - apply the fallMultiplier

            rb.velocity += Vector2.up * Physics2D.gravity.y * (fallMultiplier - 1) *
Time.fixedDeltaTime;

        }

        else if (rb.velocity.y > 0 && !Input.GetButton("Jump"))

        {

            // We are rising, but not holding Jump - apply the lowJumpMultiplier

            // This replaces the GetButtonUp logic from Step 2 for a smoother feel

            rb.velocity += Vector2.up * Physics2D.gravity.y * (lowJumpMultiplier - 1) *
Time.fixedDeltaTime;
```

```
    }

}
```

*Note:* We apply gravity manually (`* (multiplier - 1)`) because Unity is already applying gravity once (`* 1`). This adds the *extra* gravity we want. We also moved the variable jump height logic here for a more consistent, physics-based feel.

### Step 4: Coyote Time

"Coyote Time" is a game-feel trick. It allows the player to jump for a very short time *after* walking off a ledge, preventing "unfair" missed jumps.

Add new variables:
```
private float coyoteTime = 0.15f;


private float coyoteTimeCounter;
```

1.
2. Modify `Update()`:
    - When we are on the ground, reset the `coyoteTimeCounter`.
    - When we leave the ground, the counter starts ticking down.
    - We can only jump if the `coyoteTimeCounter` is greater than zero.

```
void Update()

{

    isGrounded = Physics2D.OverlapCircle(groundCheck.position, groundCheckRadius, groundLayer);



    // --- Coyote Time Logic ---

    if (isGrounded)

    {
```

```
            coyoteTimeCounter = coyoteTime;

        }

        else

        {

            coyoteTimeCounter -= Time.deltaTime;

        }



        // --- Modify Jump Input Check ---

        if (Input.GetButtonDown("Jump") && coyoteTimeCounter > 0f) // Check counter, not
isGrounded

        {

            rb.velocity = new Vector2(rb.velocity.x, jumpForce);

            coyoteTimeCounter = 0f; // Use up the coyote time jump

        }

    }



    // FixedUpdate remains the same as Step 3

    void FixedUpdate()

    {
```

```
        if (rb.velocity.y < 0)

        {

            rb.velocity += Vector2.up * Physics2D.gravity.y * (fallMultiplier - 1) *
Time.fixedDeltaTime;

        }

        else if (rb.velocity.y > 0 && !Input.GetButton("Jump"))

        {

            rb.velocity += Vector2.up * Physics2D.gravity.y * (lowJumpMultiplier - 1) *
Time.fixedDeltaTime;

        }

    }
```

## Step 5: Jump Buffering

"Jump Buffering" is the opposite of Coyote Time. It "remembers" the jump input for a short time *before* the player hits the ground. This prevents "unfair" missed jumps where the player presses jump just a few frames too early.

Add new variables:

```
private float jumpBufferTime = 0.15f;

private float jumpBufferCounter;
```

1.
2. Modify `Update()` to "buffer" the input:

```
    void Update()

    {
```

```
        isGrounded = Physics2D.OverlapCircle(groundCheck.position, groundCheckRadius,
groundLayer);



        // --- Coyote Time Logic ---

        if (isGrounded)

        {

            coyoteTimeCounter = coyoteTime;

        }

        else

        {

            coyoteTimeCounter -= Time.deltaTime;

        }



        // --- Jump Buffering Logic ---

        if (Input.GetButtonDown("Jump"))

        {

            jumpBufferCounter = jumpBufferTime;

        }

        else
```

```
        {
            jumpBufferCounter -= Time.deltaTime;
        }

        // --- COMBINED Jump Input Check ---
        if (jumpBufferCounter > 0f && coyoteTimeCounter > 0f)
        {
            rb.velocity = new Vector2(rb.velocity.x, jumpForce);

            // Reset counters
            jumpBufferCounter = 0f;
            coyoteTimeCounter = 0f;
        }
    }

    // FixedUpdate remains the same as Step 3
    // ...
```

Now, the jump will only execute if **both** conditions are met:

1. The player has pressed "Jump" recently (buffer).
2. The player is on the ground *or* has *just* left it (coyote time).

## Step 6: Adding a Double Jump

Now let's add the ability to jump a second time while in the air. We'll use the flexibility of our jump buffering to make this feel responsive too.

Add new variables to your script. `extraJumps` is public so you can decide if you want 1, 2, or more air jumps in the Inspector.
public int extraJumps = 1;

private int extraJumpsValue;

1.

We need to give the player their extra jumps back when they start. Add this in your `Start()` method:
void Start()

{

rb = GetComponent<Rigidbody2D>();

extraJumpsValue = extraJumps; // Set initial jumps

}

2.

Modify your `Update()` method. We will reset the `extraJumpsValue` when the player is grounded (just like coyote time) and change the jump logic to check for either a ground jump or an air jump.
*This replaces the `Update()` method from Step 5.*
void Update()

{

isGrounded = Physics2D.OverlapCircle(groundCheck.position, groundCheckRadius, groundLayer);

```
// --- Coyote Time & Double Jump Reset ---
if (isGrounded)
{
    coyoteTimeCounter = coyoteTime;
    extraJumpsValue = extraJumps; // Reset double jumps
}
else
{
    coyoteTimeCounter -= Time.deltaTime;
}

// --- Jump Buffering Logic ---
if (Input.GetButtonDown("Jump"))
{
    jumpBufferCounter = jumpBufferTime;
}
else
```

```
    {
        jumpBufferCounter -= Time.deltaTime;
    }

    // --- COMBINED Jump Input Check ---
    if (jumpBufferCounter > 0f)
    {
        if (coyoteTimeCounter > 0f) // Priority 1: Ground Jump (uses coyote time)
        {
            rb.velocity = new Vector2(rb.velocity.x, jumpForce);
            coyoteTimeCounter = 0f; // Consume coyote time
            jumpBufferCounter = 0f; // Consume buffer
        }
        else if (extraJumpsValue > 0) // Priority 2: Air Jump
        {
            rb.velocity = new Vector2(rb.velocity.x, jumpForce); // You could use a different
jumpForce here
            extraJumpsValue--; // Consume an air jump
            jumpBufferCounter = 0f; // Consume buffer
```

```
        }

    }

}
```

3. 
4. **No changes needed for `FixedUpdate`!** The "Better Falling" logic from Step 3 will automatically apply to the double jump as well.

Your player can now double jump, and it benefits from all the "juicy" features like jump buffering and variable height (if you release the button).

## Step 7: The Complete Script

This is the final, complete `PlayerJump.cs` script with all features (Coyote Time, Jump Buffering, and Double Jump) included. You can use this to check your work or as a starting point.

```
using UnityEngine;



public class PlayerJump : MonoBehaviour

{

    // Jump & Physics

    private Rigidbody2D rb;

    public float jumpForce = 10f;

    public float fallMultiplier = 2.5f;

    public float lowJumpMultiplier = 2f;
```

```
// Ground Check

public Transform groundCheck;

public LayerMask groundLayer;

public float groundCheckRadius = 0.2f;

private bool isGrounded;



// Coyote Time

private float coyoteTime = 0.15f;

private float coyoteTimeCounter;



// Jump Buffering

private float jumpBufferTime = 0.15f;

private float jumpBufferCounter;



// Double Jump

public int extraJumps = 1;

private int extraJumpsValue;
```

```
    void Start()

    {

        rb = GetComponent<Rigidbody2D>();

        extraJumpsValue = extraJumps; // Set initial jumps

    }



    void Update()

    {

        // --- Ground Check ---

        isGrounded = Physics2D.OverlapCircle(groundCheck.position, groundCheckRadius,
groundLayer);



        // --- Coyote Time & Double Jump Reset ---

        if (isGrounded)

        {

            coyoteTimeCounter = coyoteTime;

            extraJumpsValue = extraJumps; // Reset double jumps

        }

        else
```

```
{
    coyoteTimeCounter -= Time.deltaTime;
}

// --- Jump Buffering ---
if (Input.GetButtonDown("Jump"))
{
    jumpBufferCounter = jumpBufferTime;
}
else
{
    jumpBufferCounter -= Time.deltaTime;
}

// --- COMBINED Jump Input Check ---
if (jumpBufferCounter > 0f)
{
    if (coyoteTimeCounter > 0f) // Priority 1: Ground Jump (uses coyote time)
```

```
        {
            rb.velocity = new Vector2(rb.velocity.x, jumpForce);
            coyoteTimeCounter = 0f; // Consume coyote time
            jumpBufferCounter = 0f; // Consume buffer
        }
        else if (extraJumpsValue > 0) // Priority 2: Air Jump
        {
            rb.velocity = new Vector2(rb.velocity.x, jumpForce); // You could use a different jumpForce here
            extraJumpsValue--; // Consume an air jump
            jumpBufferCounter = 0f; // Consume buffer
        }
    }
}

void FixedUpdate()
{
    // --- Better Falling Logic ---
    if (rb.velocity.y < 0)
```

```
        {
            // We are falling - apply the fallMultiplier
            rb.velocity += Vector2.up * Physics2D.gravity.y * (fallMultiplier - 1) *
Time.fixedDeltaTime;
        }
        else if (rb.velocity.y > 0 && !Input.GetButton("Jump"))
        {
            // We are rising, but not holding Jump - apply the lowJumpMultiplier
            rb.velocity += Vector2.up * Physics2D.gravity.y * (lowJumpMultiplier - 1) *
Time.fixedDeltaTime;
        }
    }

    // Helper function to visualize the ground check radius in the Scene view
    private void OnDrawGizmosSelected()
    {
        if (groundCheck == null) return;
        Gizmos.color = Color.yellow;
        Gizmos.DrawWireSphere(groundCheck.position, groundCheckRadius);
    }
```

}

# FAQ: "Juicy" 2D Jump Tutorial

Here are some frequently asked questions about the jump tutorial, our 2.5D setup, and the concepts from the Blackthornprod video.

## General Troubleshooting

**Q: My player isn't jumping at all. What's wrong?**

**A:** This is the most common issue! Go through this checklist:

1. **Script & Components:** Is your `PlayerJump` script attached to your Player GameObject? Does your Player *also* have a `Rigidbody 2D` and a `Capsule Collider 2D`?
2. **Inspector Links:** In the Inspector for your `PlayerJump` script, did you drag your `GroundCheck` object into the `Ground Check` field?
3. **Layers (The #1 Culprit):** Did you create a "Ground" layer? Is your actual Ground object (the platform) set to this "Ground" layer? In the `PlayerJump` script, is the `Ground Layer` field set to "Ground"?
4. **Collider Contact:** Is your `GroundCheck` object's position (the little gizmo) actually touching the ground? Is the `Ground Check Radius` greater than 0?
5. **Jump Force:** Is your `Jump Force` variable set to a value higher than 0 (e.g., 10)?

**Q: My 3D player model tips over and falls flat. How do I fix it?**

**A:** You need to freeze its rotation. Select your Player, find the `Rigidbody 2D` component in the Inspector, open the `Constraints` dropdown, and check the box for **"Freeze Rotation Z"**.

**Q: I can't see the yellow "Ground Check Radius" gizmo in my Scene.**

**A:** There are two likely reasons:

1. **Gizmos are Off:** In the top-right of your Scene view, there is a "Gizmos" button. Make sure it's enabled (it should look pressed in and blue).
2. **Object Not Selected:** The `OnDrawGizmosSelected()` function only runs when the GameObject it's attached to (your Player) is *selected* in the Hierarchy. Click your Player.

**Q: My player can jump infinitely, even before I added the double jump.**

**A:** Your `GroundCheck` is probably colliding with the player's own collider.

- **The Fix:** Create a *new* layer called "Player" and assign your Player GameObject to it.

- Then, go to `Edit -> Project Settings -> Physics 2D`.
- At the bottom, you'll see a Layer Collision Matrix. Find the row/column for "Player" and uncheck the box where it intersects with "Player". This stops the player from colliding with itself.
- Make sure your `Ground Layer` variable on the script is *only* set to "Ground" and not "Player".

## Game Feel & Mechanics Explained

**Q: What's the point of `fallMultiplier`? Why not just increase the `Gravity Scale` on the Rigidbody?**

**A:** This is all about "game feel." If you just increase `Gravity Scale`, the player feels heavy on *both* the way up and the way down. This makes the jump feel weak.

By using a normal `Gravity Scale` (e.g., 2-3) and a high `fallMultiplier` (e.g., 2.5), you get the best of both worlds:

- **Jump Up:** The player rises with normal gravity, feeling light and powerful.
- **Fall Down:** The `fallMultiplier` kicks in *only* when falling, yanking the player down quickly. This feels "snappy" and gives the player more control.

**Q: What is "Coyote Time" and why do I need it?**

**A:** "Coyote Time" (named after Wile E. Coyote, who famously runs off cliffs) is a game-feel trick. It gives the player a tiny window of time (e.g., 0.15 seconds) to press the jump button *after* they have already walked off a ledge.

This prevents "unfair" moments where the player *knows* they pressed jump, but the game says they were 1 frame too late and already in the air. It makes the controls feel forgiving and responsive.

**Q: What is "Jump Buffering" and why do I need it?**

**A:** "Jump Buffering" is the opposite of Coyote Time. It "remembers" the player's jump input for a tiny window of time *before* they hit the ground.

This prevents "unfair" moments where the player presses jump a few frames *too early* while landing, and the jump doesn't execute. With buffering, the game "catches" that input and executes the jump on the very first frame the player becomes grounded.

When combined, **Coyote Time** and **Jump Buffering** mean the player's jump will *almost always* work, even if their timing is slightly off, which makes the game feel amazing to control.

## Customization

**Q: How can I make the double jump weaker or stronger than the first jump?**

**A:** This is an easy change!

1. Add a new public variable to your script: `public float doubleJumpForce = 8f;`
2. In your `Update()` method, find the `else if (extraJumpsValue > 0)` block.
3. Change this line: `rb.velocity = new Vector2(rb.velocity.x, jumpForce);` ...to this: `rb.velocity = new Vector2(rb.velocity.x, doubleJumpForce);`

**Q: My jump still feels too "floaty" or "heavy." What numbers should I change?**

**A:** The "perfect jump" is a balance of four values. Tweak them one at a time:

1. **`Gravity Scale`** (on `Rigidbody 2D`): This is your base "weight." Most platformers use a value between 2 and 5. Start here.
2. **`jumpForce`**: This is your "power." Adjust this *after* setting your gravity so the player can jump as high as you want.
3. **`fallMultiplier`**: This is your "snappiness." If the player feels like they hang in the air too long at the peak of the jump, increase this value.
4. **`lowJumpMultiplier`**: This controls your "short hop." If you tap the button and the jump is still too high, increase this value.

## Video Concepts & Further Learning

Here are resources related to the topics in the original Blackthornprod video.

- **Original Video:**
  - **How to make a 2D JUMP in Unity - "JUICY" Tutorial** by Blackthornprod
  - https://www.youtube.com/watch?v=hG9SzQxaCm8
- **Game Feel & "Juice":**
  - **GDC Talk: "Juice it or Lose it"**
    - A famous talk about adding "juice" to make games feel alive.
    - https://www.youtube.com/watch?v=Fy0aCDmgnxg
  - **Book: "Game Feel: A Game Designer's Guide to Virtual Sensation"**
    - By Steve Swink. This is considered the "bible" on the topic of game feel.
- **Coyote Time & Buffering (Advanced):**
  - **GDC Talk: "Physics-based Controls for 'Celeste'"**
    - The game *Celeste* is famous for its incredible-feeling platformer controls, which use these techniques heavily.
    - https://www.youtube.com/watch?v=N8i6-gIe-gE
- **Unity Physics:**
  - **Unity Manual: `Rigidbody 2D`**
    - Official documentation for the component that controls 2D physics.
    - https://docs.unity3d.com/Manual/class-Rigidbody2D.html
  - **Unity Manual: `Physics2D.OverlapCircle`**
    - Official documentation for the function we use to check for the ground.

- https://docs.unity3d.com/ScriptReference/Physics2D.OverlapCircle.html