**🔍 Analysis: Ball Jumping Between Blocks - Root Causes**

Based on my code analysis, here are the most likely causes of the ball jumping issue:

**1. 🎯 PRIMARY SUSPECT: Ground Contact Detection Logic**

The

BallTouchingTheGroundThresholdAsDotProductResult = 0.866f

 (30°) is used in **two different ways**:

**In PhysicObject.cs (Line 137):**

csharp

CopyInsert

if ((-fDot) > Definitions.BallTouchingTheGroundThresholdAsDotProductResult)

this.OnGroundContact(pCollision.gameObject, true);

**In Block.cs (Line 201):**

csharp

CopyInsert

if (s > Definitions.BallTouchingTheGroundThresholdAsDotProductResult)

this.SetContact(true);

**⚠️ ISSUE**: The PhysicObject uses (-fDot) while Block uses

s

 directly. This inconsistency could cause:

* Different contact detection between player and blocks
* Intermittent ground contact detection
* Ball "losing contact" momentarily between blocks

**2. 🔧 PHYSICS ENGINE ISSUES**

**A. Continuous Collision Detection Missing:**

* No evidence of

CollisionDetectionMode.ContinuousDynamic

* Fast-moving ball might tunnel through collision detection gaps
* Default discrete collision detection can miss contacts at block edges

**B. Rigidbody Sleep Threshold:**

csharp

CopyInsert

this.GetComponent<Rigidbody>().sleepThreshold = 0.0f;

* Sleep disabled completely - ball never "settles"
* Could cause micro-vibrations that trigger false edge detections

**3. 🎲 POTENTIAL BLOCK ALIGNMENT ISSUES**

**A. No Scale Modifications Found:**

* Blocks use default Unity cube scale (1,1,1)
* BUT: No explicit verification that prefabs are exactly 1x1x1 units
* Tiny gaps (0.001 units) would be invisible but cause physics issues

**B. Floating Point Precision:**

* Block positions from XML might have floating-point rounding errors
* Adjacent blocks at positions like (1.0000001, 0, 0) and (2.0000001, 0, 0) create micro-gaps

**4. 🎪 GRAVITY SYSTEM COMPLEXITY**

csharp

CopyInsert

this.GetComponent<Rigidbody>().AddForce(Gravity); // Every FixedUpdate

* Custom gravity applied via

AddForce

 instead of Unity's built-in gravity

* Could interact poorly with Unity's physics solver
* Might cause inconsistent force application at block boundaries

**🔍 Recommended Investigation Steps:**

**Immediate Checks:**

1. **Verify block prefab scales** - Check if Unity prefabs are exactly 1x1x1 units
2. **Add collision debugging** - Log collision normals and dot products when jumping occurs
3. **Check block positioning precision** - Verify adjacent blocks have exact integer coordinates
4. **Test with Unity's built-in gravity** - Temporarily disable custom gravity system

**Likely Quick Fixes:**

1. **Standardize dot product calculation** between PhysicObject and Block
2. **Enable continuous collision detection** on the player Rigidbody
3. **Add small overlap** between adjacent blocks (0.01 units) to eliminate micro-gaps
4. **Increase ground contact threshold** from 30° to 45° (0.707) for more forgiving detection

The **dot product inconsistency** is the most suspicious - it could cause the ball to lose ground contact detection right at block edges, causing the physics engine to apply different forces and create the jumping effect.

**Implemented that!! (Git commit: “slightly improved physics”)**

**📋 Detailed Action Plan for Physics Optimization**

**Phase 1: Immediate Improvements (2-3 hours)**

**1.1 Custom Gravity Timing Optimization**

**Problem**:

AddForce()

 in

FixedUpdate()

 may cause force application timing issues **Solutions to Test**:

csharp

CopyInsert

// Option A: Smoother force application

protected virtual void FixedUpdate()

{

var rigidbody = GetComponent<Rigidbody>();

// Instead of AddForce, directly set acceleration

rigidbody.velocity += Gravity \* Time.fixedDeltaTime;

}

// Option B: Impulse-based gravity

protected virtual void FixedUpdate()

{

var rigidbody = GetComponent<Rigidbody>();

// Apply gravity as impulse for more stable integration

rigidbody.AddForce(Gravity, ForceMode.Acceleration);

}

**1.2 Contact Point Stabilization**

**Problem**: Ball loses contact at block edges **Solution**: Add contact prediction/smoothing

csharp

CopyInsert

// Add to Player.cs - predict contact loss and maintain stability

void FixedUpdate()

{

// Raycast ahead in movement direction to predict contact loss

RaycastHit hit;

Vector3 rayDirection = GetComponent<Rigidbody>().velocity.normalized;

if (Physics.Raycast(transform.position, rayDirection, out hit, 0.6f))

{

// Approaching block edge - apply stabilizing force

if (hit.distance < 0.3f)

{

Vector3 stabilizingForce = -hit.normal \* 2.0f;

GetComponent<Rigidbody>().AddForce(stabilizingForce);

}

}

}

**1.3 Physics Material Optimization**

**Action**: Create optimized physics materials

* **Ball Material**: Zero bounce, controlled friction
* **Block Material**: Consistent surface properties
* **Combined Material**: Predictable interaction

**Phase 2: Advanced Physics Tuning (3-4 hours)**

**2.1 Rigidbody Configuration Optimization**

**Current Issues**:

* sleepThreshold = 0.0f

 prevents settling

* Angular drag might cause instability
* Mass/inertia distribution

**Optimizations**:

csharp

CopyInsert

// In Player.Awake()

var rb = GetComponent<Rigidbody>();

rb.sleepThreshold = 0.005f; // Allow micro-settling

rb.angularDrag = 1.0f; // Reduce from 2.5 for smoother rolling

rb.drag = 0.1f; // Slight linear drag for stability

rb.interpolation = RigidbodyInterpolation.Interpolate; // Smoother visual movement

**2.2 Contact Detection Refinement**

**Problem**: 45° threshold might still be too restrictive **Solutions**:

csharp

CopyInsert

// Dynamic threshold based on movement speed

float dynamicThreshold = Mathf.Lerp(0.5f, 0.707f,

GetComponent<Rigidbody>().velocity.magnitude / 10.0f);

if ((-fDot) > dynamicThreshold)

{

this.OnGroundContact(pCollision.gameObject, true);

}

**2.3 Block Boundary Smoothing**

**Concept**: Add tiny overlaps between blocks to eliminate micro-gaps

csharp

CopyInsert

// In Level.cs block creation

// Scale blocks slightly larger (1.001f) to create tiny overlaps

pObject.transform.localScale = Vector3.one \* 1.001f;

**Phase 3: Advanced Solutions (4-6 hours)**

**3.1 Hybrid Physics System**

**Concept**: Combine Unity's stability with custom gravity direction

csharp

CopyInsert

// Use Unity gravity for base physics, custom system only for direction changes

if (gravityDirectionChanged)

{

// Temporarily disable Unity gravity

rigidbody.useGravity = false;

// Apply custom gravity for smooth transition

rigidbody.AddForce(newGravityDirection \* gravityStrength);

// Re-enable Unity gravity after transition

StartCoroutine(ReEnableUnityGravity(1.0f));

}

**3.2 Predictive Contact System**

**Concept**: Maintain virtual contact even when physical contact is lost

csharp

CopyInsert

// Maintain "virtual ground" for short periods during transitions

private bool virtualGroundContact = false;

private float virtualGroundTimer = 0.0f;

void Update()

{

if (!physicalGroundContact && virtualGroundTimer > 0)

{

virtualGroundTimer -= Time.deltaTime;

// Apply stabilizing forces during virtual contact

}

}

**3.3 Custom Collision Response**

**Concept**: Override Unity's collision response at block boundaries

csharp

CopyInsert

// Custom collision handling for block edges

void OnCollisionStay(Collision collision)

{

if (IsBlockBoundary(collision))

{

// Apply custom forces to maintain smooth rolling

ApplyBoundaryStabilization(collision);

}

}

**Phase 4: Alternative Approaches (If needed)**

**4.1 Kinematic Movement System**

* Convert ball to kinematic Rigidbody
* Implement custom physics simulation
* Full control over movement and collision response

**4.2 Mesh-Based Rolling**

* Use sphere-cast instead of rigidbody physics
* Calculate rolling motion mathematically
* Apply rotation and position updates manually

**4.3 Contact Point Interpolation**

* Track contact history
* Interpolate between contact points
* Smooth out micro-gaps in collision detection

**🎯 Recommended Implementation Order**

**Week 1: Quick Wins**

1. **Force Mode Change**: Switch to

ForceMode.Acceleration

1. **Physics Materials**: Create and assign optimized materials
2. **Rigidbody Tuning**: Adjust drag, sleep threshold, interpolation

**Week 2: Contact Improvements**

1. **Dynamic Threshold**: Implement speed-based contact detection
2. **Contact Prediction**: Add raycast-based stability
3. **Block Overlaps**: Test tiny block scaling for gap elimination

**Week 3: Advanced Systems**

1. **Hybrid Gravity**: Implement Unity+Custom gravity system
2. **Boundary Detection**: Add specialized block edge handling
3. **Performance Testing**: Measure impact of optimizations

**📊 Success Metrics**

* **Target**: 95%+ smooth rolling (currently ~70%)
* **Performance**: No frame rate impact
* **Compatibility**: Preserve all gravity-switching mechanics

**🚨 Risk Assessment**

* **Low Risk**: Physics materials, rigidbody tuning
* **Medium Risk**: Force mode changes, contact prediction
* **High Risk**: Hybrid systems, custom collision response