## A. Overall Update Loop & State Machine

Every frame, the skater runs through roughly this sequence (in CSkaterCorePhysicsComponent::UpdatePhysics in **SkaterCorePhysicsComponent.cpp**):

cpp

CopyEdit

// 1. Read controller & map to desired inputs

m\_pControlComponent->ReadInput(dt);

// 2. Advance state machine (GROUND, AIR, RAIL, LIP, etc.)

Switch (m\_state) {

case GROUND: ApplyGroundPhysics(dt); break;

case AIR: ApplyAirPhysics(dt); break;

case RAIL: ApplyRailPhysics(dt); break;

case LIP: ApplyLipLogic(dt); break;

}

// 3. Integrate velocities

m\_position += m\_velocity \* dt;

m\_orientation = IntegrateAngularVelocity(m\_orientation, m\_angVel, dt);

// 4. Resolve collisions & snap via AdjustPhysicsComponent

m\_pAdjustComponent->FixupPenetrations(dt);

// 5. Update trick, balance, score, sound components

m\_pTrickComponent->Update(dt);

m\_pBalanceComponent->Update(dt);

m\_pScoreComponent ->Update(dt);

– This core loop lives in **SkaterCorePhysicsComponent.cpp** and drives every frame of play .

## B. Input Mapping & Stat‑Driven Parameters

All player inputs (analog stick X/Y, buttons) are read in CSkaterPhysicsControlComponent::ReadInput (in **SkaterPhysicsControlComponent.h/cpp**). Key mappings:

* **Forward/Back** → throttle or brake
* **Left/Right** → turn on ground or spin in air
* **Jump** → queue a lip pop or manual/tuck in air
* **Grind** → hold to stay on rails or to start a manual

Those raw inputs get stored as floats (m\_stickForward, m\_stickLateral, m\_doJump, m\_doGrind, etc.), then later used by the physics component to compute forces. Every force parameter is looked up via

cpp

CopyEdit

float accelStat = m\_pSkater->GetScriptedStat("ACCEL", dt);

float topSpeed = m\_pSkater->GetScriptedStat("TOP\_SPEED", dt);

float groundFric = m\_pSkater->GetScriptedStat("GROUND\_FRICTION", dt);

…so you can tweak everything in data tables without touching code .

## C. Ground Physics Equations

When in **GROUND** state, THUG does:

1. **Compute Desired Acceleration**

cpp

CopyEdit

Vector3 forwardDir = m\_orientation.Transform(Vector3(0,0,1));

Vector3 accelForce = forwardDir \* (m\_stickForward \* accelStat);

m\_velocity += accelForce \* dt;

1. **Clamp to Top Speed**

cpp

CopyEdit

float speed = m\_velocity.Length();

if (speed > topSpeed)

m\_velocity = m\_velocity \* (topSpeed / speed);

1. **Apply Rolling Friction**

cpp

CopyEdit

// exponential decay: vel \*= pow(friction, dt)

m\_velocity \*= powf(groundFric, dt);

1. **Slope Slide Component**  
   Each ground feeler returns a contact normal **n**; the code projects out the perpendicular:

cpp

CopyEdit

Vector3 gravityProj = Dot(m\_velocity, n) \* n;

m\_velocity -= gravityProj \* (1.0f - slopeFriction);

Slope friction is another stat (SLOPE\_FRICTION) .

## D. Air Physics & Rotation

In **AIR** state:

1. **Gravity & Drag**

cpp

CopyEdit

m\_velocity += Vector3(0,-gravity,0) \* dt;

m\_velocity \*= powf(airFriction, dt);

1. **Angular Drag**

cpp

CopyEdit

m\_angVel \*= powf(airAngularFriction, dt);

1. **Player‑Controlled Rotation**
   * **Spin** (around board’s Y): lateral stick → spin torque
   * **Flip** (around local X): button combo → flip torque

cpp

CopyEdit

m\_angVel += stickX \* spinSpeedStat \* dt \* upAxis;

m\_angVel += buttonFlip \* flipSpeedStat \* dt \* rightAxis;

1. **Orientation Integration**  
   The board’s orientation quaternion q is updated by

cpp

CopyEdit

q = Normalize(q + 0.5f \* Quaternion(0, m\_angVel) \* q \* dt);

so you can mix spins + flips seamlessly .

## E. Ramp “Lip” Pop Logic

When a toe/heel feeler hits a lip collider flagged in the map:

1. **Detect Lip & Buffer**  
   A short timer (m\_lipGraceTimer) prevents double‑pops.
2. **On Jump Button**

cpp

CopyEdit

Vector3 popImpulse = lipNormal \* popStrengthStat

+ forwardDir \* popForwardStat;

m\_velocity += popImpulse;

1. **Switch State → AIR**  
   The Core physics drops you into AIR, carrying your popped velocity .

## F. Rail Grinding Math

Entering **RAIL** state:

1. **Find Closest Rail Segment** via side‑feeler raycasts in **SkaterAdjustPhysicsComponent**.
2. **Project Velocity onto Rail**

cpp

CopyEdit

Vector3 railTangent = (p2 - p1).Normalized();

m\_velocity = railTangent \* Dot(m\_velocity, railTangent);

1. **Apply Grind Friction**

cpp

CopyEdit

m\_velocity \*= powf(railFrictionStat, dt);

1. **Rail Snap & Alignment**  
   Each frame the skater’s board mesh is repositioned onto the rail with

cpp

CopyEdit

pos = ClosestPointOnSegment(p1, p2, wheelPos);

orient.AlignZ(railNormal);

Grinds break on low speed or grind‑button release .

## G. Balance Meter Dynamics

Manuals & rail grinds use the same meter:

1. **Stateful Balance Value** b ∈ [–1,1], starts at 0.
2. **Wobble + Player Correction**

cpp

CopyEdit

float wobble = sin(time \* wobbleFreq) \* wobbleAmp;

float desired = (isManual ? stickY : 0.0f);

b += (desired + wobble - b) \* manualControlStat \* dt;

1. **Auto‑Center & Bail**
   * If |b| > 1 → TriggerBail().
   * If |b| < 0.05 → drift toward zero by abs(b)\*manualRecoveryStat\*dt.

Stats “MANUAL\_BALANCE” (wobbleAmp scale) and “MANUAL\_CONTROL” (correction speed) tune the feel .

## H. Special Meter & Buffs

Every successful trick adds points to the special bar. Once full:

1. **Activate** → sets a flag in ScoreComponent.
2. GetScriptedStat(name) sees this flag and returns

cpp

CopyEdit

baseValue + (specialActive ? specialBonus : 0);

so **top speed**, **pop**, **spin**, even **grind friction** get boosted by + 2–4 points each .

### ✦ ****Next Steps for Your 1:1 Remake****

* **Port the exact state enum** and transition table from **SkaterCorePhysicsComponent.h/.cpp**.
* **Recreate the feelers** (6 ground, 2 side for rails, 2 for lips) with equivalent raycasts.
* **Load all stats** from a human‑editable table (like the game’s .q files).
* **Integrate** the balance‑meter equation verbatim (wobble + catch).
* **Copy** the quaternion integration routine for perfect aerial spin/flips.
* **Wire up** a special‐state flag that gates stat boosts inside your stat‑lookup function.