Use of physics in hyrax.exe

A Bouncing Hyrax

rattatwinko

May 7, 2026

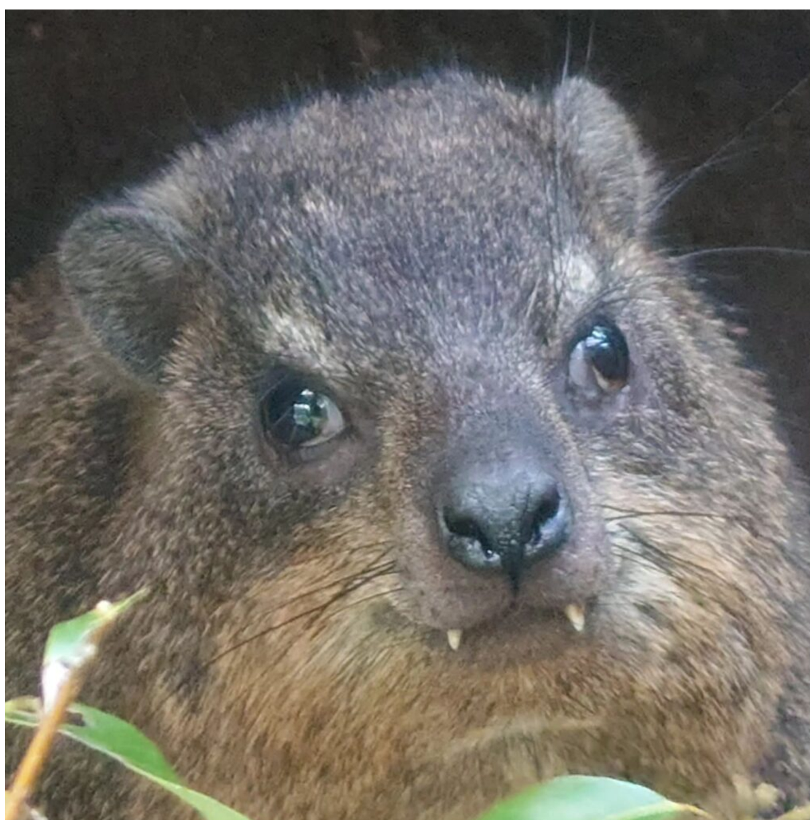

Figure 1: The `hyrax.exe`, the hyrax in the window

Contents

1 Overview **2**

2 Physical Model **2**

2.1 Gravity 2
2.2 Air Drag 3
2.3 Position Update (Semi-Implicit Euler) 3
2.4 SI Euler Integration 3

3 Boundary Handling **4**
3.1 Vertical Walls (Left / Right) 4
3.2 Horizontal Walls (Top / Bottom) 4
3.2.1 Top Edge 4
3.2.2 Bottom Edge (Ground) 4

4 Numerical Considerations **4**
4.1 Time Step and Frame Rate 4
4.2 Collision Order and Tunnelling 4
4.3 Integration Order 5

5 Constants Reference **5**

6 Implementation **5**
6.1 Usage Notes 8
6.2 GDI+ Rendering Integration 8

Abstract

`hyrax.exe` is a small executable Win32 application which works with GDI+, to draw a sprite of a hyrax on the desktop. The hyrax is bouncy, it can be flung, this is because of a small physics engine, written in C++ which implements a semi-implicit Euler integrator, constant acceleration, and a coefficient of restitution (which is $\beta = 0.6$). Ground contact includes friction ($\mu = 0.8$). This document outlines the mathematical model, bound checking, and some minor implementation details. All code which is outlined is written in C++ and written for Windows.

1 Overview

The Program updates the hyrax's position (x, y) and velocity (v_x, v_y) each frame with a time step Δt (seconds). The coordinate system is that of GDI+: origin at the top-left corner, $+x$ right, $+y$ down. All distances are in pixels, velocities in pixels per second, acceleration in px/s^2.

2 Physical Model

2.1 Gravity

A constant downward acceleration ($G = 1200\,\mathrm{s}^{-2}$) is applied at the beginning of each time step:

$$v_y \leftarrow v_y + G \cdot \Delta t.$$

2.2 Air Drag

After gravity, both velocity components are multiplied by the drag factor $\alpha = 0.999$:

$$v_x \leftarrow \alpha\, v_x, \qquad v_y \leftarrow \alpha\, v_y.$$

Drag models a weak linear damping (Stokes law). Because $\alpha < 1$, the velocity decays exponentially in the absence of other forces, preventing indefinite acceleration under gravity.

2.3 Position Update (Semi-Implicit Euler)

The position is advanced using the *new* velocities:

$$x \leftarrow x + v_x \cdot \Delta t, \qquad y \leftarrow y + v_y \cdot \Delta t.$$

This scheme (velocity then position) is SI (Symplectic Integrator) and more stable than the explicit Euler method for oscillatory systems. It ensures energy conservation in the undamped, elastic case.

2.4 SI Euler Integration

The simulation uses the semi-implicit (symplectic) Euler method to integrate Newton equations of motion. Let acceleration be $a = (a_x, a_y)$, velocity $v = (v_x, v_y)$, and position $p = (x, y)$. For a time step Δt, the update is:

$$v_{n+1} = v_n + a_n \Delta t,$$

followed by

$$p_{n+1} = p_n + v_{n+1} \Delta t.$$

Unlike the explicit Euler method, which updates position using the *old* velocity v_n, the semi implicit form uses the newly computed velocity v_{n+1}. This ordering improves numerical stability, particularly for heavy oscillation models or heavy collision models.

With the hyrax, acceleration consists primarily of gravity:

$$a_y = G,$$

while drag is applied as a multiplicative damping factor after the velocity update:

$$v \leftarrow \alpha v.$$

The complete integration sequence for one frame is, therefore,:

$$\text{gravity} \rightarrow \text{drag} \rightarrow \text{position} \rightarrow \text{collision response}.$$

Although the method is only first-order accurate in time, it generally exhibits a better long term energy behavior than the standard explicit Euler integrator.

3 Boundary Handling

The window has width W and height H. The hyrax sprite occupies a bounding box of width w and height h. The center of the sprite cannot cross the inner bounds:

$$R = W - w \quad \text{(right bound)}, \qquad B = H - h \quad \text{(bottom bound)}.$$

Collisions are resolved in x first, then y, to prevent double-penetration issues. Each collision applies a coefficient of restitution $\beta = 0.6$ and corrects the position exactly to the boundary.

3.1 Vertical Walls (Left / Right)

- **Left:** if $x < 0$, set $x = 0$ and $v_x = -\beta\, v_x$.
- **Right:** if $x > R$, set $x = R$ and $v_x = -\beta\, v_x$.

Vertical velocity v_y is unchanged. No friction is applied on vertical walls.

3.2 Horizontal Walls (Top / Bottom)

3.2.1 Top Edge

If $y < 0$, clamp $y = 0$ and invert the vertical speed: $v_y = -\beta\, v_y$.

3.2.2 Bottom Edge (Ground)

Ground contact is more elaborate:
1. Clamp position: $y = B$.
2. Apply bounce: $v_y = -\beta\, v_y$.
3. Apply ground friction: $v_x = \mu\, v_x$, with $\mu = 0.8$.
4. If $v_y < \epsilon$ (where $\epsilon = 5\,\mathrm{s}^{-1}$), set $v_y = 0$ to stop micro-bouncing.

The friction coefficient μ reduces horizontal speed only while the sprite is on the ground. The rest threshold eliminates the “infinite bounce” phenomenon when v_y becomes negligible.

4 Numerical Considerations

4.1 Time Step and Frame Rate

The physics are updated every frame with the actual elapsed time Δt (typically 1/60s). The integrator is first-order accurate, so large Δt (e.g., under heavy load) can cause energy drift or missed collisions. For stable simulation, Δt should not exceed about 0.02s.

4.2 Collision Order and Tunnelling

Because the order of checks is x then y, a fast moving sprite could theoretically tunnel through a corner if it crosses both boundaries in one frame. In practice, with Δt small and speeds below $\sim 2000\,\mathrm{s}^{-1}$, tunnelling is rare. A more robust approach would use swept collision detection, but it is omitted for simplicity.

4.3 Integration Order

The sequence **gravity** → **drag** → **position** → **collisions** is deliberate:

- Applying gravity before drag allows the drag to damp the newly gained speed (physically plausible).
- Updating position with the post-drag velocity gives a semi-implicit Euler step.
- Collisions are resolved after the position update to prevent objects from getting stuck inside boundaries.

Changing this order would alter the effective dynamics.

5 Constants Reference

Constant	Value	Description
GRAVITY	1200.0	Downward acceleration (pxs^2)
RESTITUTION	0.6	Energy retention after bounce
AIR_DRAG	0.999	Velocity multiplier per frame
GROUND_FRICTION	0.8	Horizontal speed reduction on ground
REST_THRESHOLD	5.0	Minimum $|v_y|$ to consider at rest

Table 1: Physics constants defined as `static constexpr double` in the header.

6 Implementation

The C++ header declares the `physics` class and its members. The update method implements exactly the steps described above.

```cpp
#pragma once

#define scd static constexpr double

class physics
{
public:
  scd GRAVITY = 1200.0;
  scd RESTITUTION = 0.6;
  scd AIR_DRAG = 0.999;
  scd GROUND_FRICTION = 0.8;
  scd REST_THRESHOLD = 5.0;

  physics();
  physics(double x, double y, double vx = 0.0, double vy = 0.0);
  void update(double dt, int screenW, int screenH, int objW, int
   objH);

  // getters / setters
  double x() const { return m_x; }
  double y() const { return m_y; }
  double vx() const { return m_vx; }
  double vy() const { return m_vy;  }
  void setPos(double x, double y);
  void setVelocity(double vx, double vy);
  void resetVelocity();

private:
  double m_x, m_y;
  double m_vx, m_vy;
};
```

```cpp
#include "physics.h"
#include <cmath>

physics::physics() : physics(0.0, 0.0) {}

physics::physics(double x, double y, double vx, double vy)
    : m_x(x), m_y(y), m_vx(vx), m_vy(vy) {
}

void physics::update(double dt, int screenWidth, int screenHeight,
   int objWidth, int objHeight) {
    m_vy += GRAVITY * dt;
    m_vx *= AIR_DRAG;
    m_vy *= AIR_DRAG;

    m_x += m_vx * dt;
    m_y += m_vy * dt;

    const double rightBound = static_cast<double>(screenWidth -
   objWidth);
    const double bottomBound = static_cast<double>(screenHeight -
   objHeight);

    if (m_x < 0.0) {
        m_x = 0.0;
        m_vx = -m_vx * RESTITUTION;
    }
    else if (m_x > rightBound) {
        m_x = rightBound;
        m_vx = -m_vx * RESTITUTION;
    }

    if (m_y < 0.0) {
        m_y = 0.0;
        m_vy = -m_vy * RESTITUTION;
    }
    else if (m_y > bottomBound) {
        m_y = bottomBound;
        m_vy = -m_vy * RESTITUTION;
        m_vx *= GROUND_FRICTION;
        if (std::fabs(m_vy) < REST_THRESHOLD)
            m_vy = 0.0;
    }
}

void physics::setPos(double x, double y) {
    m_x = x;
    m_y = y;
}

void physics::setVelocity(double vx, double vy) {
    m_vx = vx;
    m_vy = vy;
}

void physics::resetVelocity() {
    m_vx = m_vy = 0.0;
}
```

6.1 Usage Notes

A `physics` object stores the complete dynamic state of the hyrax, including position and velocity. The object may be constructed with an initial position and optional starting velocity, allowing the sprite to begin either at rest or already in motion.

During execution, the physics should be updated once per rendered frame by calling:

```
update(dt, screenW, screenH, objW, objH)
```

The parameter `dt` represents the elapsed time since the previous frame, measured in seconds. Using the real elapsed time rather than a fixed frame delay ensures that the simulation remains approximately frame-rate independent. For example, at 60 fps

$$dt \approx \frac{1}{60}\ \text{s}.$$

The remaining parameters specify the dimensions of both the screen and the sprite, allowing the physics engine to compute collision boundaries correctly.

After each update step, the current position may be retrieved using `x()` and `y()`, while the instantaneous velocity components are accessible through `vx()` and `vy()`. These values are typically passed directly to the rendering layer.

External impulses may be applied with:

```
setVelocity(vx, vy)
```

This is used, for example, when the user clicks or drags the hyrax, giving it an instantaneous change in momentum. Calling `resetVelocity()` sets both velocity components to zero, stopping all motion.

6.2 GDI+ Rendering Integration

Because GDI+ uses the same top-left origin and downward y axis, the physics coordinates map directly to drawing coordinates. For a sprite of size $w \times h$, draw it at:

$$\text{left} = x - \frac{w}{2}, \quad \text{top} = y - \frac{h}{2}.$$

Conclusion

The `hyrax.exe` physics engine provides a simple, fun way to have a bouncy window on your desktop! And using C++ makes it fast.