# Principles of Computer Game Design and Implementation

**Lecture 16**

# We already learned

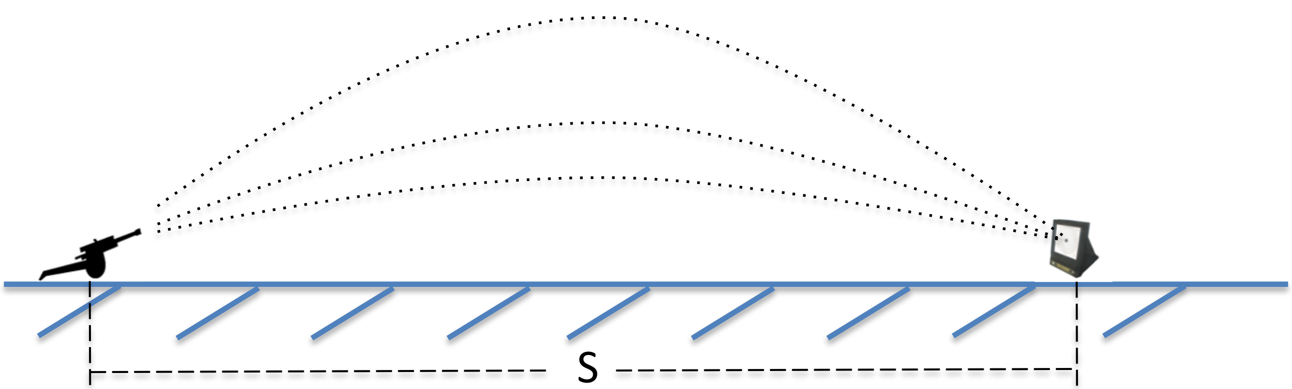
- Collision detection
  - two approaches (overlap test, intersection test)
  - Low-level, mid-level, and high-level view
- Collision response
  - Newtonian mechanics

# Outline for today

- An application of Newtonian dynamics in targeting
- Collision recipe
  - Bouncing problem

# Physics: Prediction

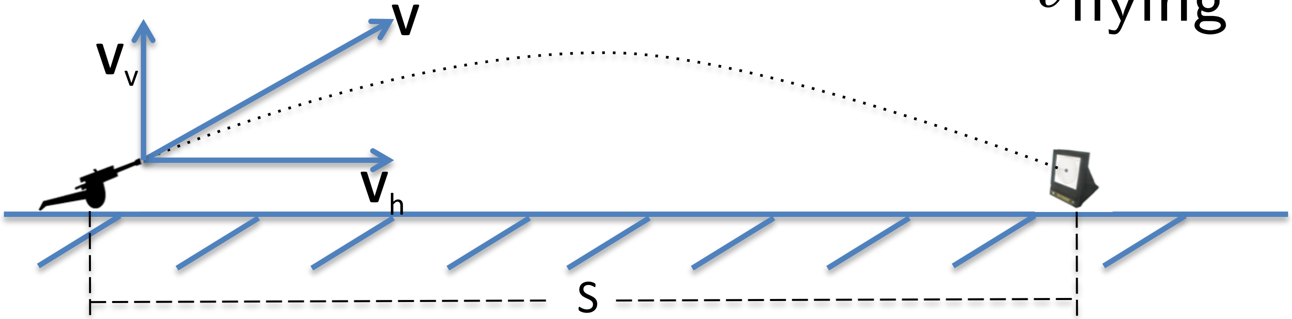
- Consider the targeting problem: a gun takes aim at a target
  - Given: *S* – distance to the target
  - Compute the bullet velocity vector
    - Incomplete information



# Targeting Problem (1)

- Consider *horizontal* and *vertical* components of the velocity vector **V**
- **Assume** that
  - – the horizontal component is given and
  - – it does not change (no wind / drag)
- Flying time is

$$t_{\text{flying}} = \frac{S}{V_h}$$

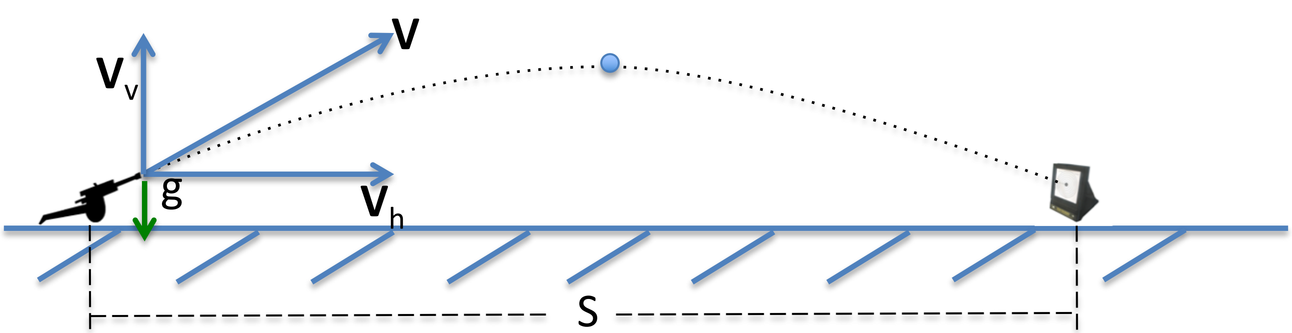


# Targeting Problem (2)

- Vertically, the motion is *up* and *down*

$$V_v(t) = V_v - gt$$

- **Assume** that
  - the gun and target are levelled
- At the highest point $V_v(t) = 0$
  - time to the highest point is half the flying time



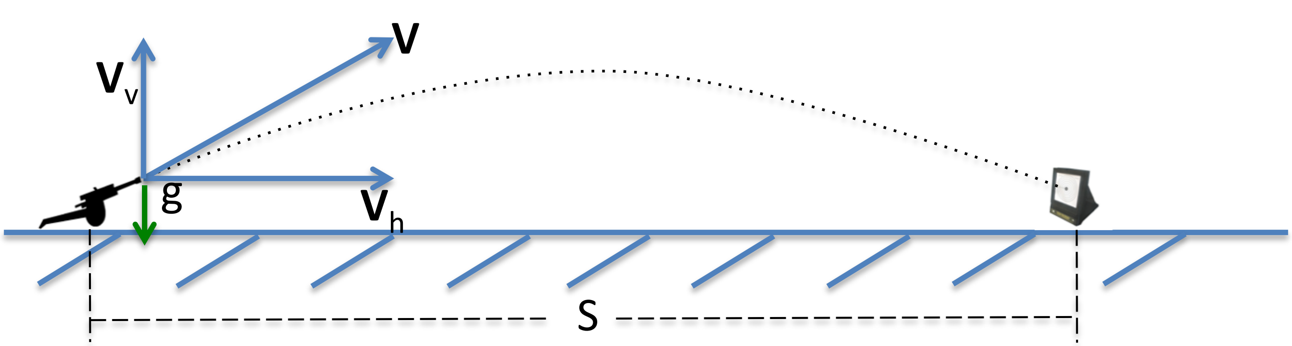
# Targeting Problem (3)

- Thus,

$$0 = V_v - g(t_{\text{flying}})/2$$

$$t_{\text{flying}} = \frac{S}{V_h}$$

$$V_v = \frac{gS}{2V_h}$$



# HelloAiming

```
float distance = 100f;
bullet.setLocalTranslation(0, 0, 0);
target.setLocalTranslation(distance, 0, 0);
...
float vx = 20f;
float vy = (g*distance) / (2*vx);
velocity = new Vector3f(vx,vy,0);
...
pubic void simpleUpdate() {
  if(bullet.getLocalTranslation().getY() >= 0 ) {
   velocity = velocity.add(gravity.mult(tpf));
   bullet.move(velocity.mult(tpf));
  }
}
```

X-component of velocity vector. “Horizontal” speed.

Run it with different vx!!

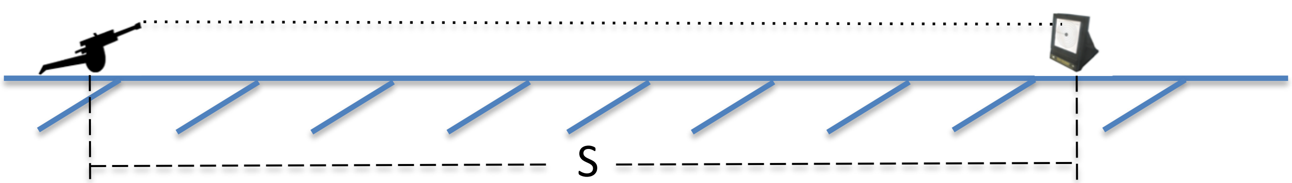
# Euler Steps: Advantages and Disadvantages

- Work well when motion is slow (small simulation steps) and forces are well-defined
  - **F**, **a** and ***V*** remain same in the time interval
- Does not work well when
  - Simulation steps are large
  - Approximation errors accumulate
  - **F**, **a** and ***V*** change rapidly over time

Inaccurate for serious applications (e.g. flying a real rocket)
Widely used in computer games for its simplicity

# If Accuracy Matters

- Use other integration methods
  - Typically, much more computationally demanding
- Cheat
  - E.g. in our aiming example, if the bullet speed is high, consider it travel along a straight line
  - Adjust its position if necessary



# Computer Science Approach: Iterations

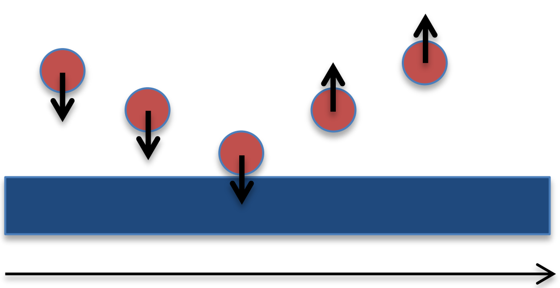
- Shoot at will
- See where it land
- If undershot, increase power
- If overshot, decrease power

But what will the user think?

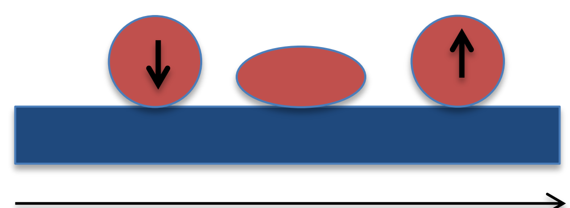
# Collision Resolution

Colliding objects change the trajectory

- Two main approaches
  - Impact
    - Instantaneous change of velocity as a result of collision
  - Contact
    - Gradual change of velocity and position



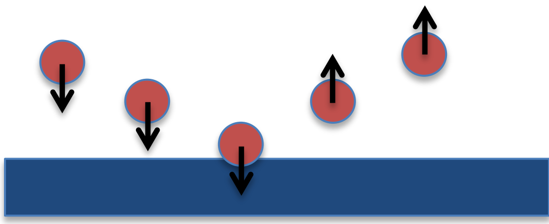
Time line

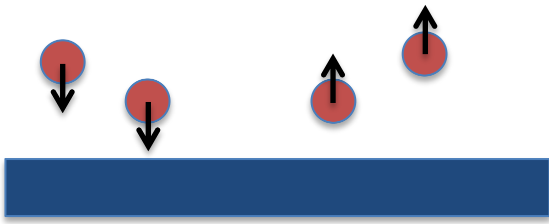


Time line

# Penetration

- Both Impact and Contact may lead to penetration of one entity into another
  - Calculate the exact time of collision
    - Complex computations
    - Collision may never be seen
  - Treat penetration as part of collision

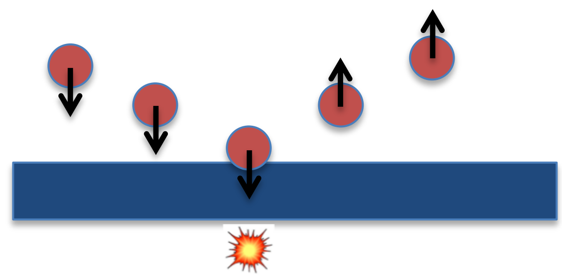




# Collision Detection

```
CollisionResults results =
          new CollisionResults();
boxes.collideWith(ball.
          getWorldBound(), results);
if (results.size() > 0) {
  …
}
```

# Simple Impact-Based Response



```
protected void simpleUpdate() {
…
  if(results.size() > 0) {
     velocity.setY(-velocity.getY());
  }
…
}
```

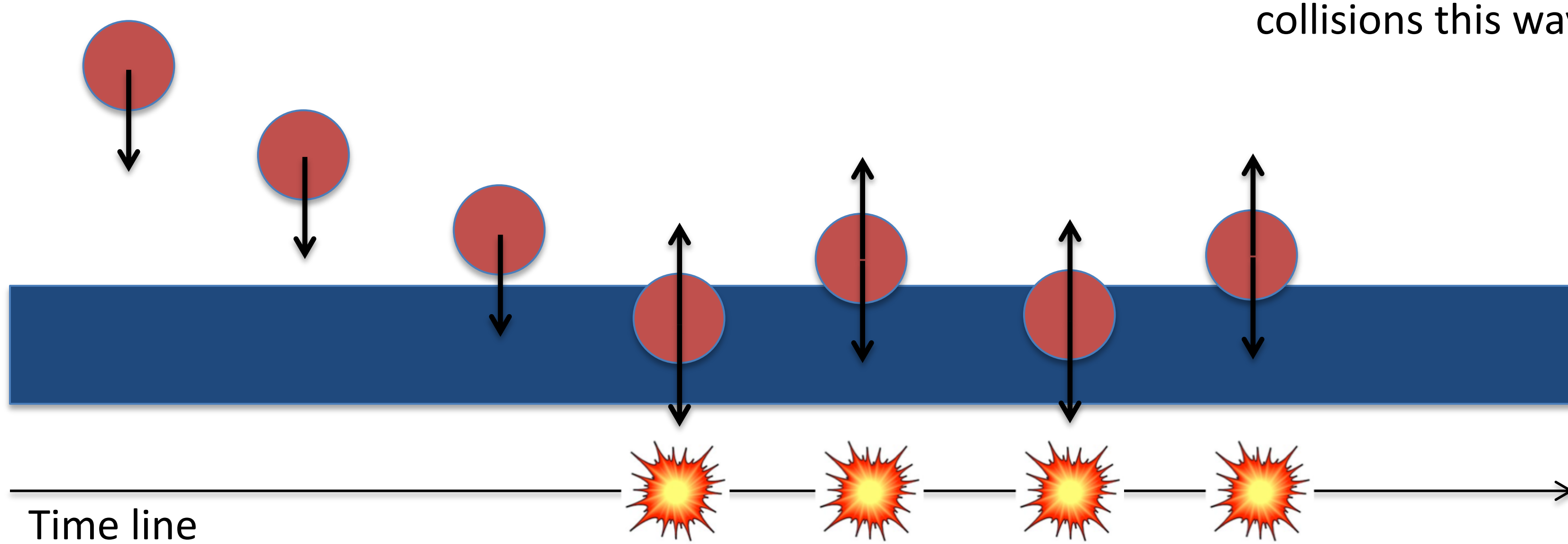
Problems:

1. Assumes floor is horizontal
2. Penetration is not fully taken into account

# Penetration Can Cause Glitches

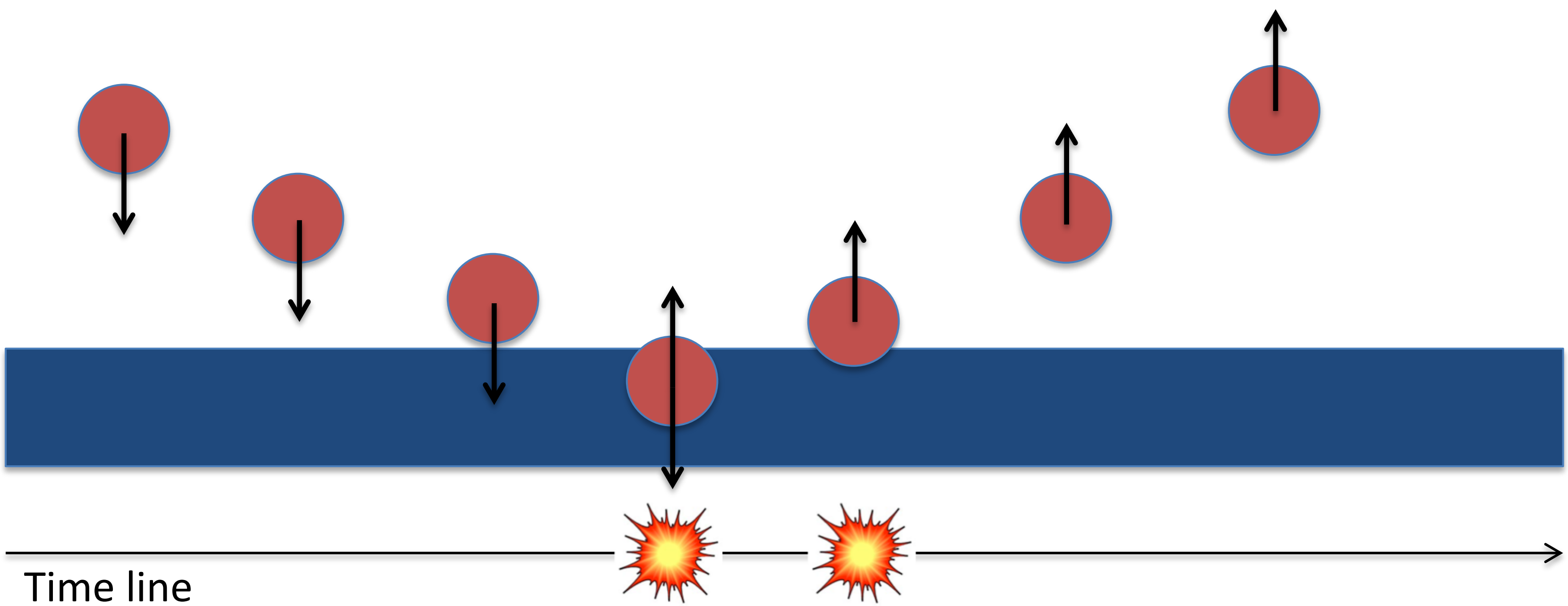
```
if(results.size() > 0) {
    velocity.setY(-velocity.getY());
 }
```

One of the jME2 examples handles collisions this way…. ☺



# Better Solution

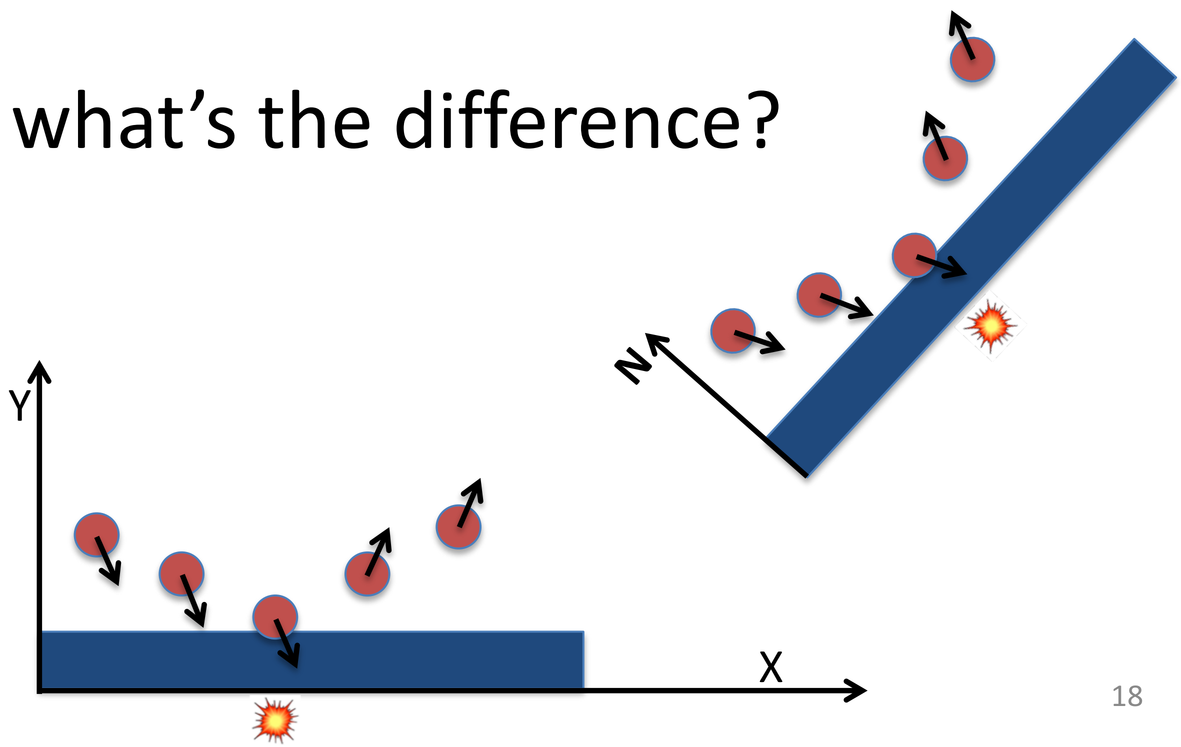
```
if(results.size() > 0) {
  velocity.setY(FastMath.abs(velocity.getY()));
 }
```



# Ball-Plain Collision

```
if(results.size() > 0) {
   velocity.setY(
            FastMath.abs(velocity.getY()));
}
```

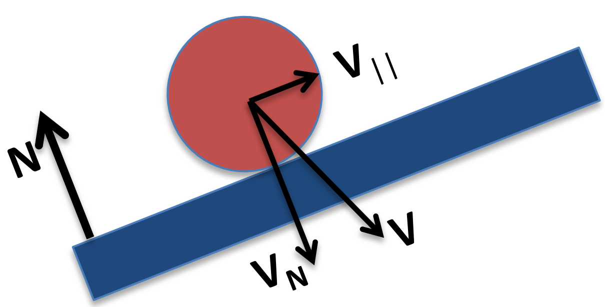
- Still works
  - So, what's the difference?



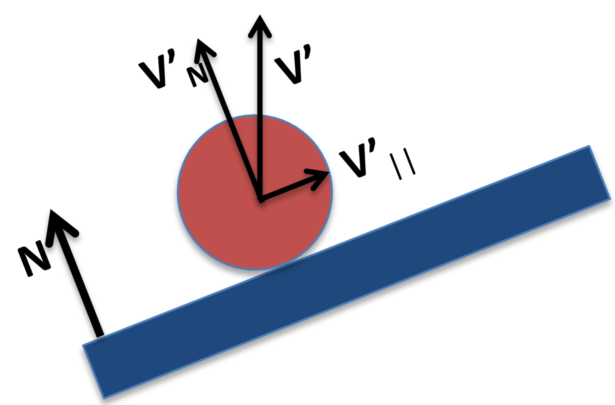
# Ball-Plain Collision Recipe

- Split the ball velocity vector into two components
- $\mathbf{V} = \mathbf{V}_{\mathbf{N}} + \mathbf{V}_{||}$
  - $\mathbf{V}_{\mathbf{N}} = (\mathbf{V}\bullet\mathbf{N})\mathbf{N}$
  - $\mathbf{V}_{||} = \mathbf{V} - \mathbf{V}_{\mathbf{N}}$

$\mathbf{V'} = \mathbf{V'}_{\mathbf{N}} + \mathbf{V'}_{||}$
- $\mathbf{V'}_{\mathbf{N}} = \mathrm{abs}(\mathbf{V}\bullet\mathbf{N})\mathbf{N}$
- $\mathbf{V'}_{||} = \mathbf{V}_{||}$



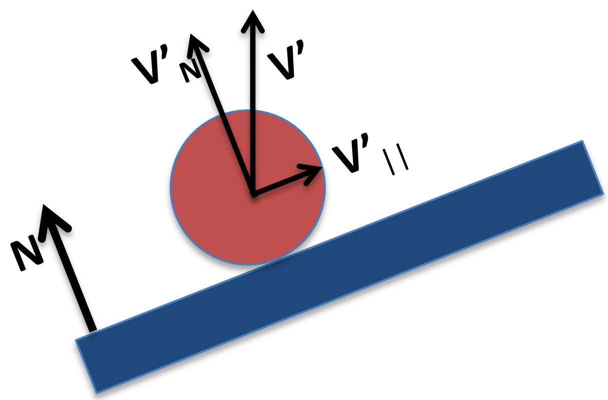

Before collision




After collision

# Energy Loss

- When entities collide some energy is lost
- Simple model:




After collision

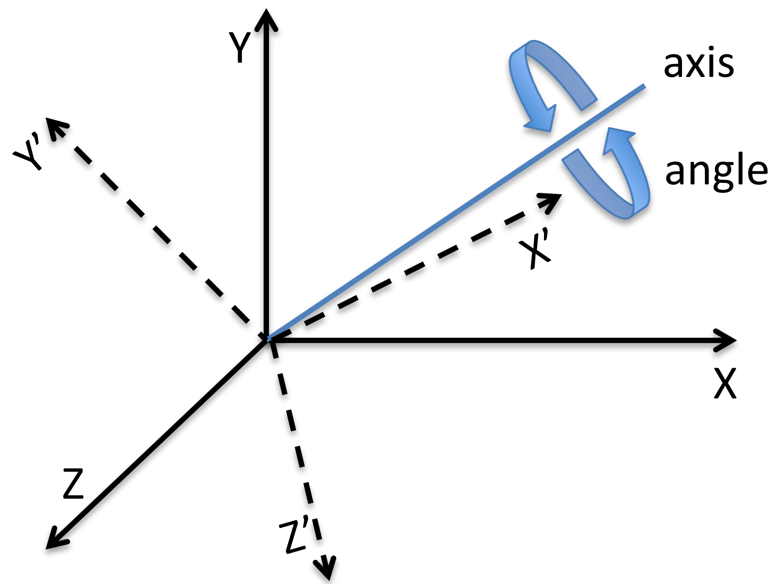
Energy loss

$$\mathbf{V'} = \mathbf{V'_N} + \mathbf{V'_{||}}$$

- $\mathbf{V'_N} = \lambda \, \mathrm{abs}(\mathbf{V} \bullet \mathbf{N})\mathbf{N}$
- $\mathbf{V'_{||}} = \mathbf{V_{||}}$

No friction

# Recall: Quaternion from 3 Vectors




- q.fromAngleAxis(angle, axis) : (**x**,**y**,**z**) -> (**x**1,**y**1,**z**1)
- `q.fromAxes(`**`x`**`1,`**`y`**`1,`**`z`**`1)` –

  "`inverse`"

# HelloBounce (1)

## Just to set up the scenery

```
protected Geometry boxFromNormal(String name,
                                      Vector3f n) {
  Box b = new Box(10f, 1f, 10f);
  Geometry bg = new Geometry(name, b);
  Material mat = new Material…; bg.setMaterial(mat);

  Quaternion q = new Quaternion();
  q.fromAxes(n.cross(Vector3f.UNIT_Z), n,
                                    Vector3f.UNIT_Z);
  bg.setLocalRotation(q);
  return bg;
}
```

Recall: $\mathbf{X} = \mathbf{Y} \times \mathbf{Z}$

# HelloBounce (2)

```
if(…) {
  float projVal = velocity.dot(floor2Normal);
  Vector3f projection = floor2Normal.mult
                                        (projVal);
  Vector3f parall = velocity.subtract
                                    (projection);
  velocity = parall.add(floor2Normal.mult
        (energyLoss*FastMath.abs(projVal)));
}
```