# Principles of Computer Game Design and Implementation

**Lecture 15**

# We already learned

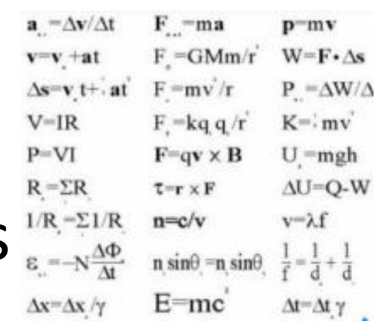
- Collision Detection
  - two approaches (overlap test, intersection test)
  - Low-level, mid-level, and high-level view

# Collision Response

- What happens after a collision is detected?
  1. Prologue
     - Check if collision should be ignored
     - Sound / visual effects
  2. Collision
     - *Resolve collision*
  3. Epilogue
     - Propagate the effects
       - destroy object(s), play sound…

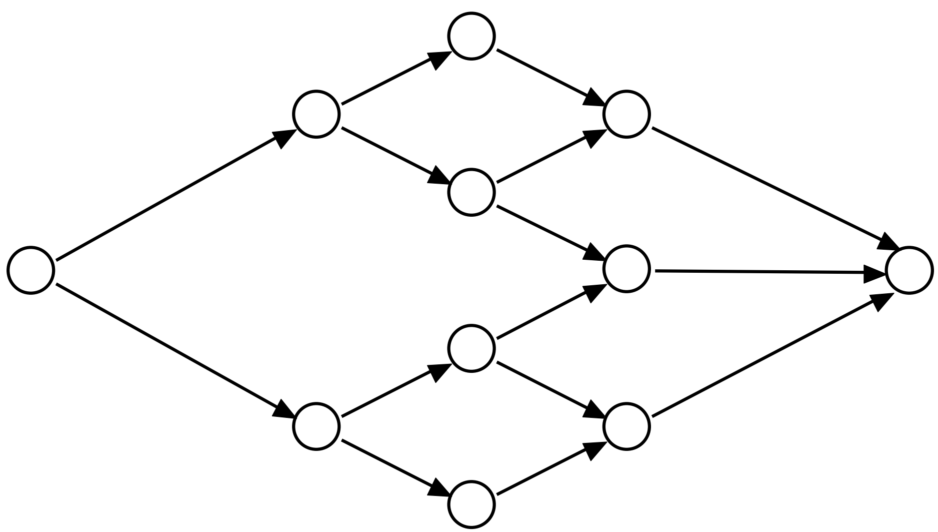
# Collision Resolution

- Animation based
  - An artist models collision
    - A rocket hits a target…
  - Motion-capture
    - Sport games
- Physics based
  - Generated by an algorithm
  - Based on (more or less) realistic models



# Recall: Classic Game Structure

- A convexity
- Starts with a single choice, widens to many choices, returns to a single choice

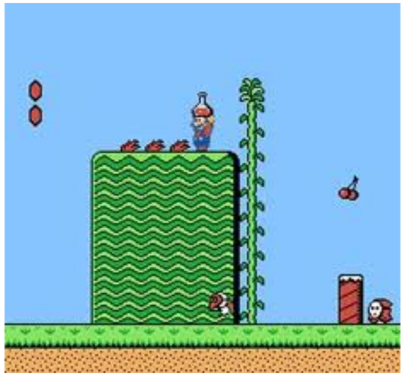


# Why Physics?

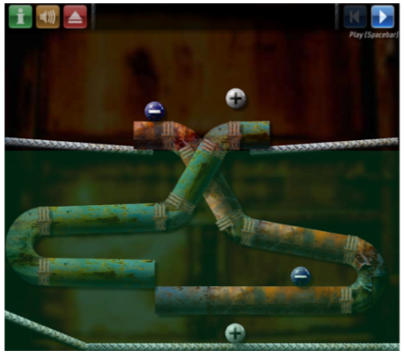
- Responsive behaviour
  - Infinitely many possibilities
- For centauries people were *describing* the world
  - We can use the equations to *model* the world
- Can be hard
  - Knowledge of physics
  - “Real” physics is too expensive computationally

# “Motion Science” in Games

- Kinematics
  - Motion of bodies without considering forces, friction, acceleration,…
  - Not realistic



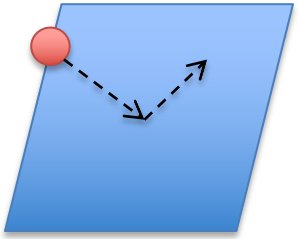
- Dynamics
  - Interaction with forces and torques



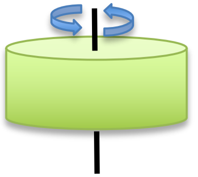
# Keep It Simple

Separate translation and rotation

- Particle physics
  - A sphere with a perfect smooth, frictionless surface. No rotation
  - Interaction with forces and environment
    - Position, Velocity, Acceleration



- Solid body physics
  - Torques, angular velocity, angular momentum



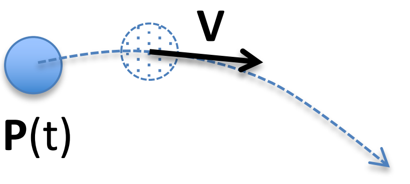
# Continuous Motion

- Particles move in a "smooth way"
  - Position as a *function* of time

    ***P***(t) is the position of P in the moment t
  - The *derivative*

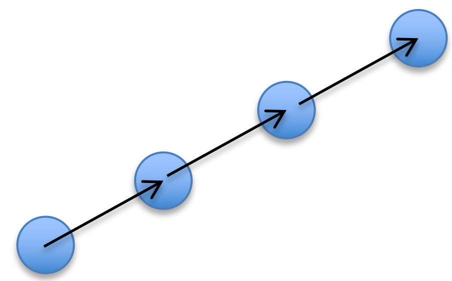
$$\frac{d\,\mathbf{P}(t)}{dt}$$

    describes how ***P***(t) changes over time
    - Velocity (speed)

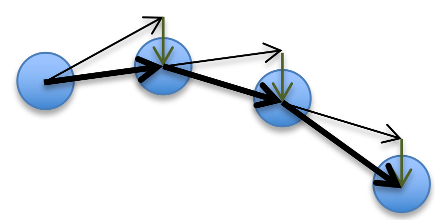


# Discrete Particle Motion

- Uniform motion
  - Nothing affects the motion



- Gravitational pull



# Integrators

- The process of computing the position of a body based on forces and interaction with other bodies in called *integration*

- A program that computes it is an *integrator*

# Newton's Laws

1. Every body remains in a state of rest or uniform motion unless it is acted on by an external force
2. A body of mass *m* subject to force ***F*** accelerates as described by

Vectors

$$\boldsymbol{F} = m\boldsymbol{a}$$

3. Every action has an equal and opposite reaction

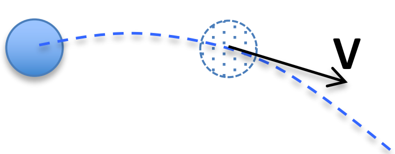
# Position and Velocity

## Continuous physics

- $$\mathbf{V}(t) = \frac{d\,\mathbf{P}(t)}{dt}$$

⇩

- $\mathbf{P}(t) = \ldots$ (maths)



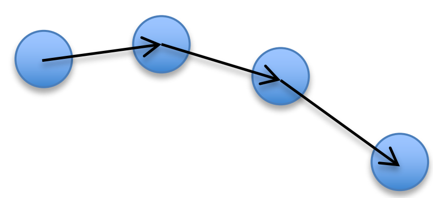

## Discrete physics

- $$\mathbf{V}(t) = \frac{\Delta\,\mathbf{P}(t)}{\Delta t} = \frac{\mathbf{P}_{i+1} - \mathbf{P}_i}{tpf}$$

⇩

- $$\mathbf{P}_{i+1} = \mathbf{P}_i + tpf \cdot \mathbf{V}(t)$$

Main loop iteration ($i+1$) — Time per frame ($tpf$)

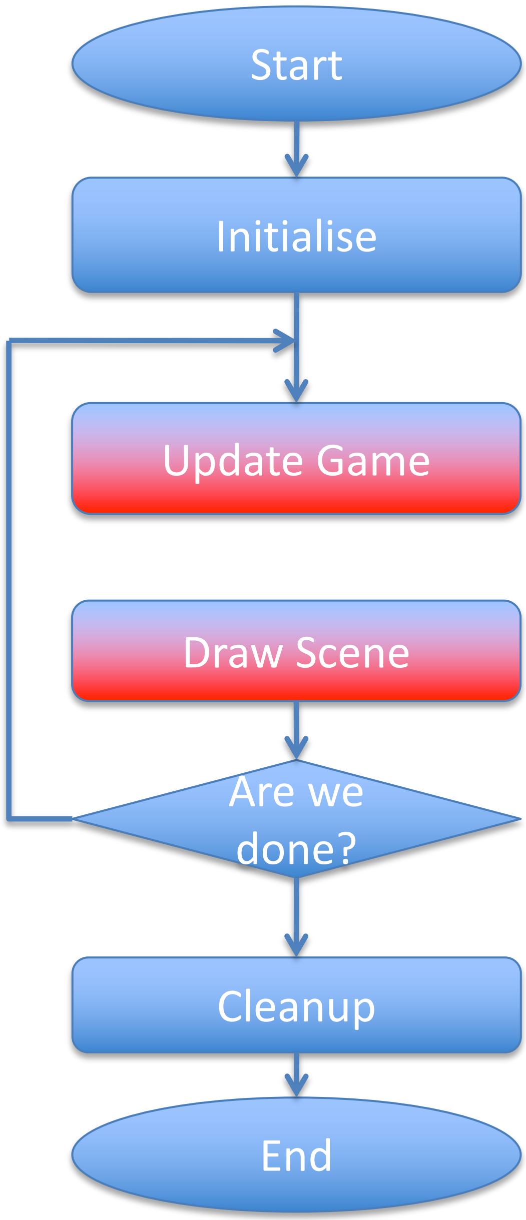


# Recall: Arbitrary Translation

- Every iteration *update* the position

$$P = P + speed \cdot tpf \cdot \mathbf{U}(t)$$

- **U**(t) - the direction of movement
  - Depends on time!!
- *speed* is speed
- *tpf* is time per frame



# Velocity and Acceleration

## Continuous physics

- $$\mathbf{a}(t) = \frac{d\,\mathbf{V}(t)}{dt}$$

⇓

- $\mathbf{V}(t) = \ldots$(maths)

## Discrete physics

- $$\mathbf{a}(t) = \frac{\Delta\,\mathbf{V}(t)}{\Delta t} = \frac{\mathbf{V}_{i+1} - \mathbf{V}_i}{tpf}$$

⇓

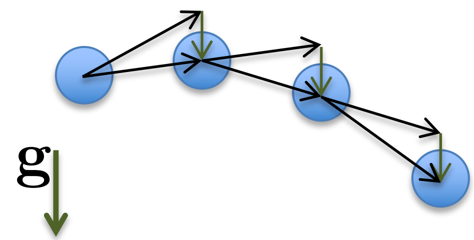
- $$\mathbf{V}_{i+1} = \mathbf{V}_i + tpf \cdot \mathbf{a}(t)$$

Main loop iteration

Time per frame

# Example: Gravitational Pull

- $\mathbf{a}(t) = \mathbf{g}$ = 9.8 N/kg
- $\mathbf{V}_{i+1} = \mathbf{V}_i + tpf \cdot \mathbf{g}$
- $\mathbf{P}_{i+1} = \mathbf{P}_i + tpf \cdot \mathbf{V}_{i+1}$




```
Vector3f velocity = new Vector3f(10,10,0);
Vector3f gravity = new Vector3f(0, -9.8f, 0);
…
public void simpleUpdate() {
   velocity = velocity.add(gravity(tpf));
   ag.move(velocity.mult(tpf));
}
```

# Acceleration and Force

Newton's second law: a body of mass $m$ subject to force $\boldsymbol{F}$ accelerates as described by

$$\boldsymbol{F}(\mathrm{t}) = m\boldsymbol{a}(\mathrm{t})$$

⇩

$$\boldsymbol{a}(\mathrm{t}) = \boldsymbol{F}(\mathrm{t})/m$$

Example:

Engine thrust $\boldsymbol{F}_{\mathrm{engine}} = k\boldsymbol{U}_{\boldsymbol{V}}$

Linear drag $\boldsymbol{F}_{\mathrm{D}}(\mathrm{t}) = -b\boldsymbol{V}(\mathrm{t})$

Use more often for simplicity

Quadratic drag $\boldsymbol{F}_{\mathrm{QD}}(\mathrm{t}) = -\mathrm{c}|\boldsymbol{V}(\mathrm{t})|^2\boldsymbol{V}(\mathrm{t})$

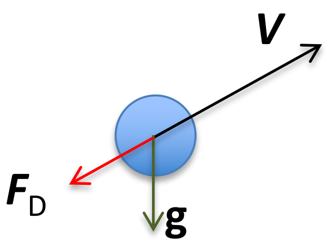
# Example: Pull + Drag

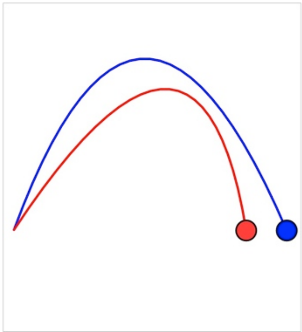
$$\mathbf{F}_{i+1} = -b\mathbf{V}_i$$

$$\mathbf{a}_{i+1} = \mathbf{g} + \mathbf{F}_{i+1}/m$$

$$\mathbf{V}_{i+1} = \mathbf{V}_i + tpf \cdot \mathbf{a}_{i+1}$$

$$\mathbf{P}_{i+1} = \mathbf{P}_i + tpf \cdot \mathbf{V}_{i+1}$$



```
Vector3f force = velocityB.mult(-b);
accelerationB = gravity.add(force.divide(m));
velocityB =
       velocityB.add(accelerationB.mult(tpf));
bg.move(velocityB.mult(tpf));
```

# Example: Pull + Drag + Thrust

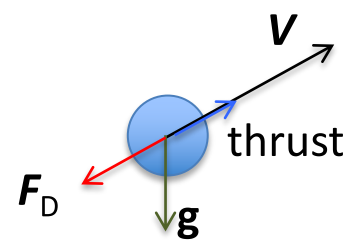
$$\mathbf{F}_{i+1} = -b\mathbf{V}_i + k\mathbf{U}_{\mathbf{V}}$$

Unit vector in the direction of ***V***

$$\mathbf{a}_{i+1} = \mathbf{g} + \mathbf{F}_{i+1}/m$$

$$\mathbf{V}_{i+1} = \mathbf{V}_i + tpf \cdot \mathbf{a}_{i+1}$$

$$\mathbf{P}_{i+1} = \mathbf{P}_i + tpf \cdot \mathbf{V}_{i+1}$$




```
Vector3f directionC = velocityC.normalize();
Vector3f forceC = velocityC.mult(-b).
                              add(directionC.mult(thrust));
accelerationC = gravity.add(forceC.divide(m));
velocityC = velocityC.add(accelerationC.mult(tpf));
cg.move(velocityC.mult(tpf));
```

# Simulation Recipe

- Add up all the forces acting on the object
  - Gravity, drag, thrust, spring pull,…
- Represent the motion as discrete steps

$$\mathbf{a}_{i+1} = \mathbf{g} + \mathbf{F}_{i+1}/m$$
$$\mathbf{V}_{i+1} = \mathbf{V}_i + tpf \cdot \mathbf{a}_{i+1}$$
$$\mathbf{P}_{i+1} = \mathbf{P}_i + tpf \cdot \mathbf{V}_{i+1}$$

Euler steps

# Rotation

- Rotation of a uniform (again simplification) solid body can be described mathematically
  - Speed vs angular speed
  - Force vs torque

- Represent as discrete motion
- Use Euler steps to compute the rotation matrix
- Combine with translation

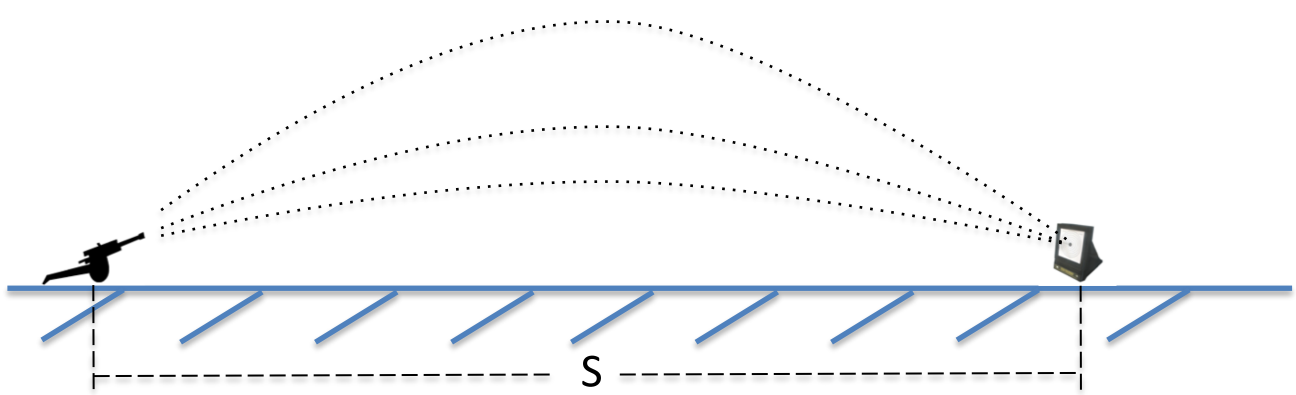
Hard but doable

# Accuracy of Simulation

- How accurate this simulation is?
- Does it matter?
  - It’s all about illusion, if the behaviour looks right, we do not care.
- But…

# Physics: Prediction

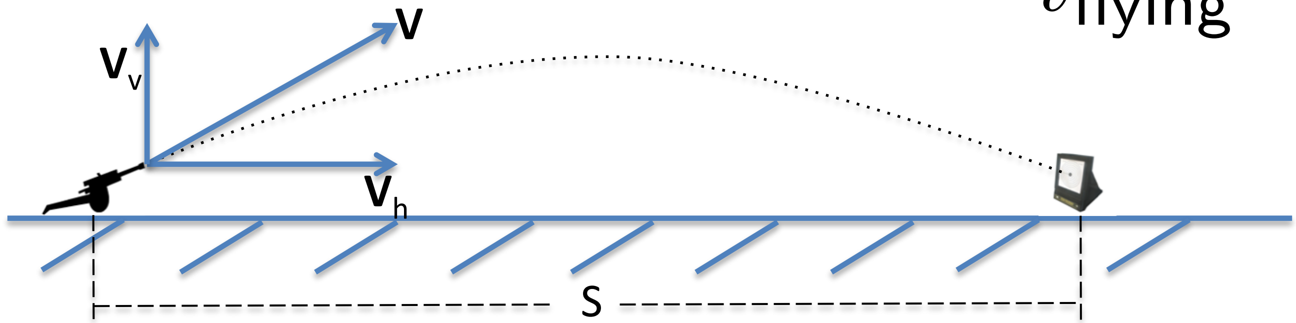
- Consider the targeting problem: a gun takes aim at a target
  - Given: *S* – distance to the target
  - Compute the bullet velocity vector
    - Incomplete information



# 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

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

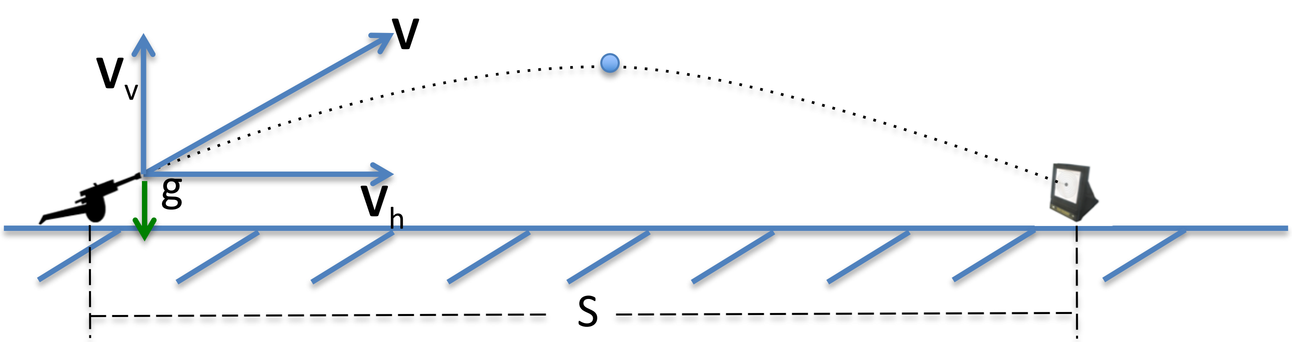


# Targeting Problem (2)

- 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



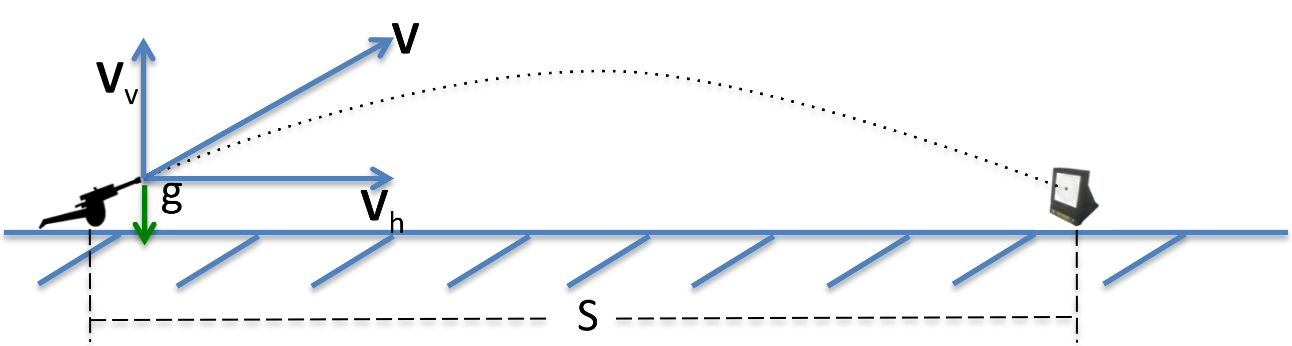
# Targeting Problem (3)

- 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!!

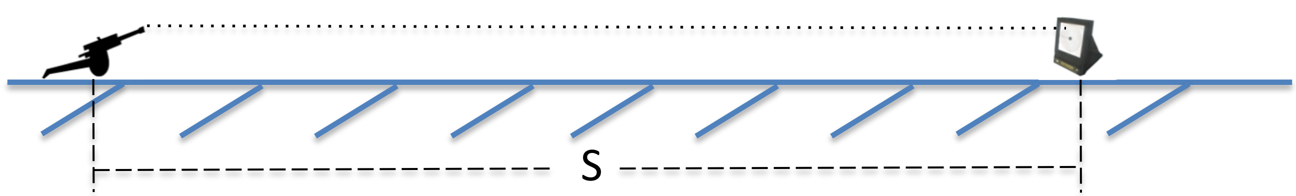
# Euler Steps: Advantages and Disadvantages

- 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?