Chapter 4.3
Real-time Game Physics

Outline

- Introduction
  - Motivation for including physics in games
  - Practical development team decisions
- Particle Physics
  - Particle Kinematics
  - Closed-form Equations of Motion
- Numerical Simulation
  - Finite Difference Methods
  - Explicit Euler Integration
  - Verlet Integration
- Brief Overview of Generalized Rigid Bodies
- Brief Overview of Collision Response
- Final Comments

Real-time Game Physics

Introduction

Why Physics?

- The Human Experience
  - Real-world motions are physically-based
  - Physics can make simulated game worlds appear more natural
  - Makes sense to strive for physically-realistic motion for some types of games
- Emergent Behavior
  - Physics simulation can enable a richer gaming experience

Why Physics?

- Developer/Publisher Cost Savings
  - Classic approaches to creating realistic motion:
    - Artist-created keyframe animations
    - Motion capture
    - Both are labor intensive and expensive
  - Physics simulation:
    - Motion generated by algorithm
    - Theoretically requires only minimal artist input
    - Potential to substantially reduce content development cost

High-level Decisions

- Physics in Digital Content Creation Software:
  - Many DCC modeling tools provide physics
  - Export physics-engine-generated animation as keyframe data
  - Enables incorporation of physics into game engines that do not support real-time physics
  - Straightforward update of existing asset creation pipelines
  - Does not provide player with the same emergent-behavior-rich game experience
  - Does not provide full cost savings to developer/publisher

High-level Decisions

- Real-time Physics in Game at Runtime:
  - Enables the emergent behavior that provides player a richer game experience
  - Potential to provide full cost savings to developer/publisher
  - May require significant upgrade of game engine
  - May require significant update of asset creation pipelines
  - May require special training for modelers, animators, and level designers
  - Licensing an existing engine may significantly increase third party middleware costs

High-level Decisions

- License vs. Build Physics Engine:
  - License middleware physics engine
    - Complete solution from day 1
    - Proven, robust code base (in theory)
    - Most offer some integration with DCC tools
    - Features are always a tradeoff

High-level Decisions

- License vs. Build Physics Engine:
  - Build physics engine in-house
    - Choose only the features you need
    - Opportunity for more game-specific optimizations
    - Greater opportunity to innovate
    - Cost can be easily be much greater
    - No asset pipeline at start of development

Real-time Game Physics

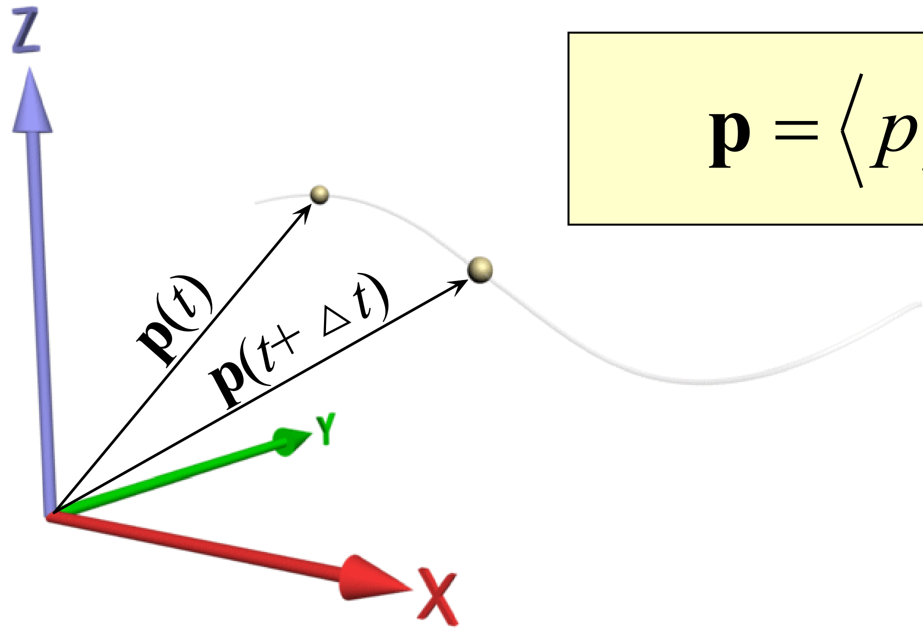
The Beginning: Particle Physics

The Beginning: Particle Physics

- What is a Particle?
  - A sphere of finite radius with a perfectly smooth, frictionless surface
  - Experiences no rotational motion
- Particle Kinematics
  - Defines the basic properties of particle motion
  - Position, Velocity, Acceleration

Particle Kinematics - Position

- Location of Particle in World Space
  - SI Units: meters (m)



$$\mathbf{p} = \langle p_x, p_y, p_z \rangle$$

  - Changes over time when object moves

Particle Kinematics - Velocity and Acceleration

- Velocity (SI units: m/s)
  - First time derivative of position:

$$\mathbf{V}(t) = \lim_{\Delta t \to 0} \frac{\mathbf{p}(t+\Delta t) - \mathbf{p}(t)}{\Delta t} = \frac{d}{dt}\mathbf{p}(t)$$

- Acceleration (SI units: m/s^2)
  - First time derivative of velocity
  - Second time derivative of position

$$\mathbf{a}(t) = \frac{d}{dt}\mathbf{V}(t) = \frac{d^2}{dt^2}\mathbf{p}(t)$$

Newton's 2nd Law of Motion

- Paraphrased – "An object's change in velocity is proportional to an applied force"
- The Classic Equation:

$$\mathbf{F}(t) = m\mathbf{a}(t)$$

  - m = mass (SI units: kilograms, kg)
  - $\mathbf{F}(t)$ = force (SI units: Newtons)

What is Physics Simulation?

- The Cycle of Motion:
  - Force, $\mathbf{F}(t)$, causes acceleration
  - Acceleration, $\mathbf{a}(t)$, causes a change in velocity
  - Velocity, $\mathbf{V}(t)$ causes a change in position
- Physics Simulation:
  - Solving variations of the above equations over time to emulate the cycle of motion

Example: 3D Projectile Motion

- Constant Force
  - Weight of the projectile, $\mathbf{W} = m\mathbf{g}$
  - $\mathbf{g}$ is constant acceleration due to gravity
- Closed-form Projectile Equations of Motion:

$$\mathbf{V}(t) = \mathbf{V}_{init} + \mathbf{g}(t - t_{init})$$

$$\mathbf{p}(t) = \mathbf{p}_{init} + \mathbf{V}_{init}(t - t_{init}) + \frac{1}{2}\mathbf{g}(t - t_{init})^2$$

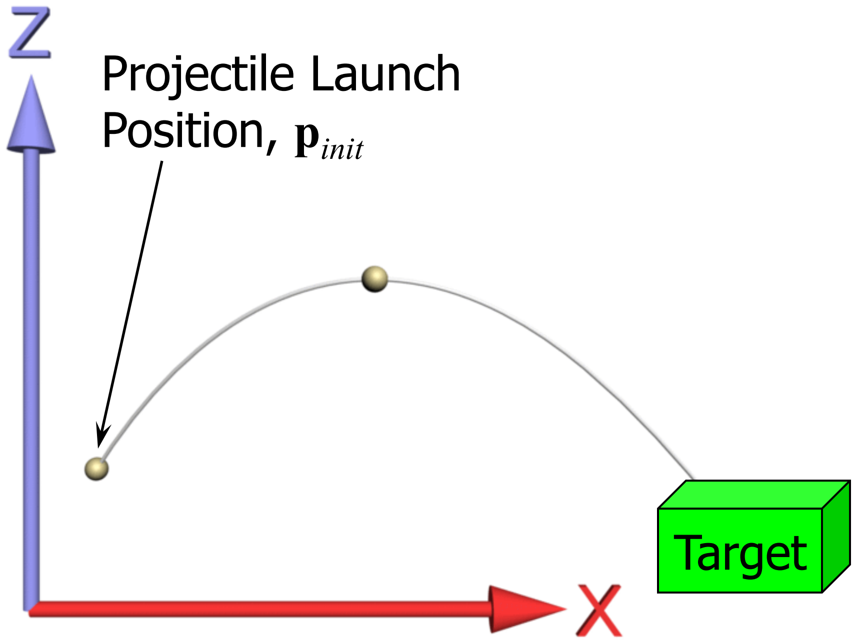
  - These closed-form equations are valid, *and exact**, for any time, t, in seconds, greater than or equal to t_{init}

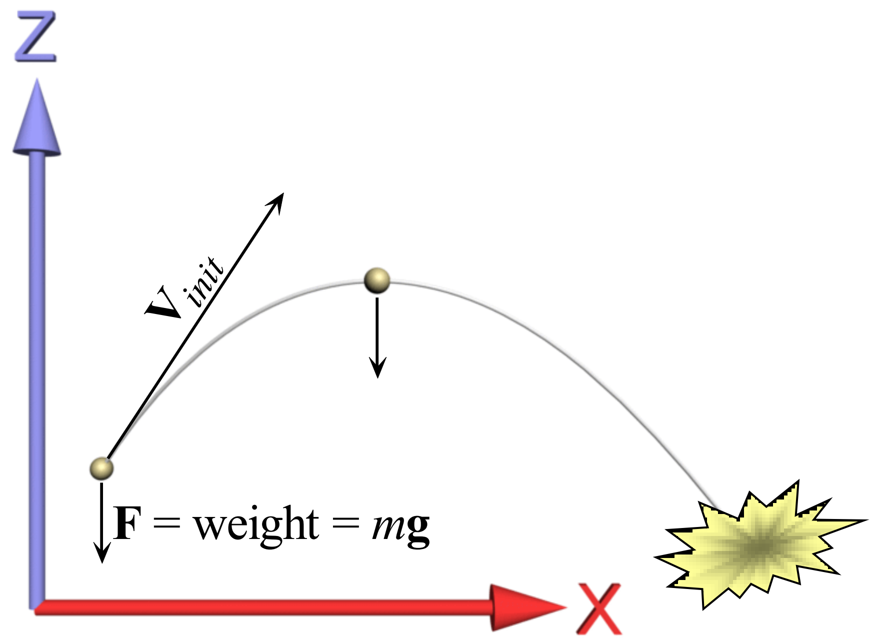
Example: 3D Projectile Motion

- Initial Value Problem
  - Simulation begins at time t_{init}
  - The initial velocity, $\mathbf{v}_{init}$ and position, $\mathbf{p}_{init}$, at time t_{init}, are known
  - Solve for later values at any future time, t, based on these initial values
- On Earth:
  - If we choose positive Z to be straight up (away from center of Earth), g_{Earth} = 9.81 m/s^2:

$$\mathbf{g}_{Earth} = -g_{Earth}\hat{k} = \langle 0.0, 0.0, -9.81 \rangle \ \mathrm{m/s}^2$$

Concrete Example: Target Practice





Concrete Example: Target Practice

- Choose $\mathbf{V}_{init}$ to Hit a Stationary Target
  - $\mathbf{p}_{target}$ is the stationary target location
  - We would like to choose the initial velocity, $\mathbf{V}_{init}$, required to hit the target at some future time, t_{hit}.
  - Here is our equation of motion at time t_{hit}:

$$\mathbf{p}_{target} = \mathbf{p}_{init} + \mathbf{V}_{init}\left(t_{hit} - t_{init}\right) + \frac{1}{2}\mathbf{g}\left(t_{hit} - t_{init}\right)^2$$

  - Solution in general is a bit tedious to derive…
  - Infinite number of solutions!
  - Hint: Specify the magnitude of $\mathbf{V}_{init}$, solve for its direction

Concrete Example: Target Practice

- Choose Scalar launch speed, V_{init}, and Let:

$$\mathbf{V}_{init} = \langle V_{init} \cos\theta \cos\phi, V_{init} \sin\theta \cos\phi, V_{init} \sin\phi \rangle$$

- Where:

$$\cos\theta = \frac{p_{target,x} - p_{init,x}}{\sqrt{\left(p_{target,x} - p_{init,x}\right)^2 - \left(p_{target,y} - p_{init,y}\right)^2}} \;;\; \sin\theta = \frac{p_{target,y} - p_{init,y}}{\sqrt{\left(p_{target,x} - p_{init,x}\right)^2 - \left(p_{target,y} - p_{init,y}\right)^2}}$$

$$\tan\phi = \frac{A \pm \sqrt{A^2 - 2g\left(\frac{A}{V_{init}}\right)^2 \left(\frac{1}{2} g \left(\frac{A}{V_{init}}\right)^2 + p_{target,z} - p_{init,z}\right)}}{g} \left(\frac{V_{init}}{A}\right)^2$$

$$A = \frac{\left(p_{target,y} + p_{target,x}\right) - \left(p_{init,y} + p_{init,x}\right)}{\left(\cos\theta + \sin\theta\right)}$$

Concrete Example: Target Practice

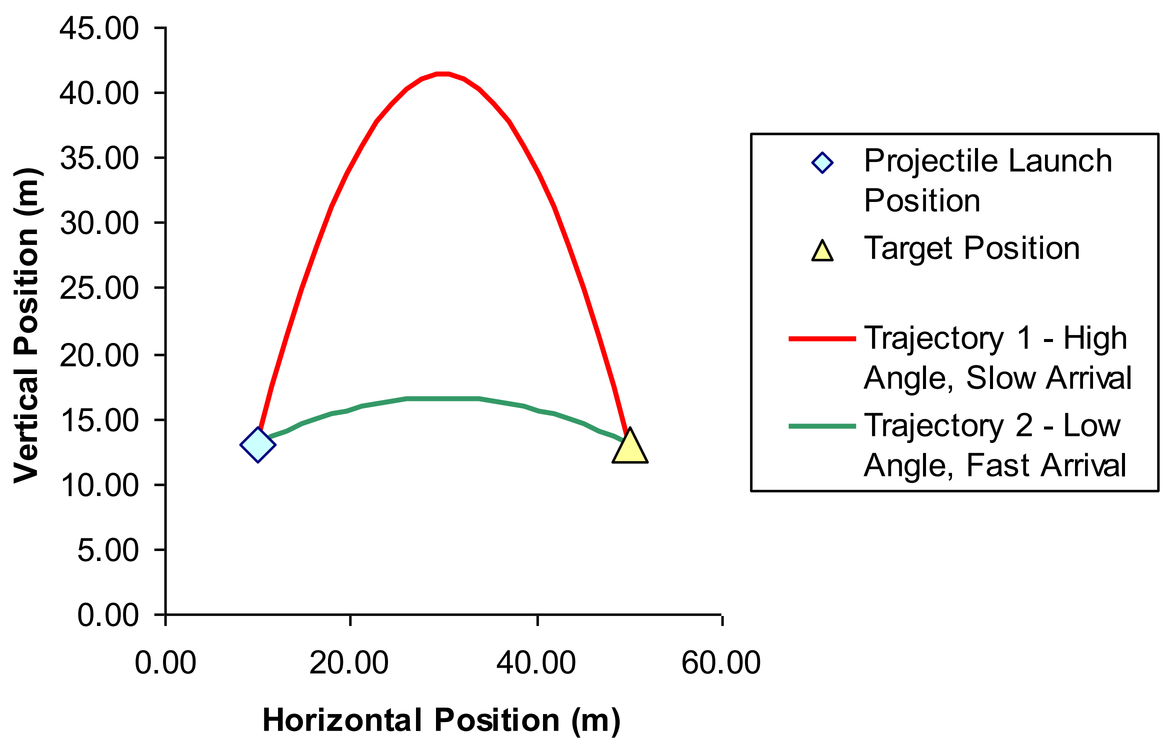
- If Radicand in $\tan\phi$ Equation is Negative:
  - No solution. V_{init} is too small to hit the target

$$\text{if}\left(A^2 - 2g\left(\frac{A}{V_{init}}\right)^2\left(\frac{1}{2}g\left(\frac{A}{V_{init}}\right)^2 + p_{target,z} - p_{init,z}\right)\right) < 0, \text{ then no solution!}$$

- Otherwise:
  - One solution if radicand == 0
  - If radicand > 0, TWO possible launch angles, ϕ
    - Smallest ϕ yields earlier time of arrival, t_{hit}
    - Largest ϕ yields later time of arrival, t_{hit}

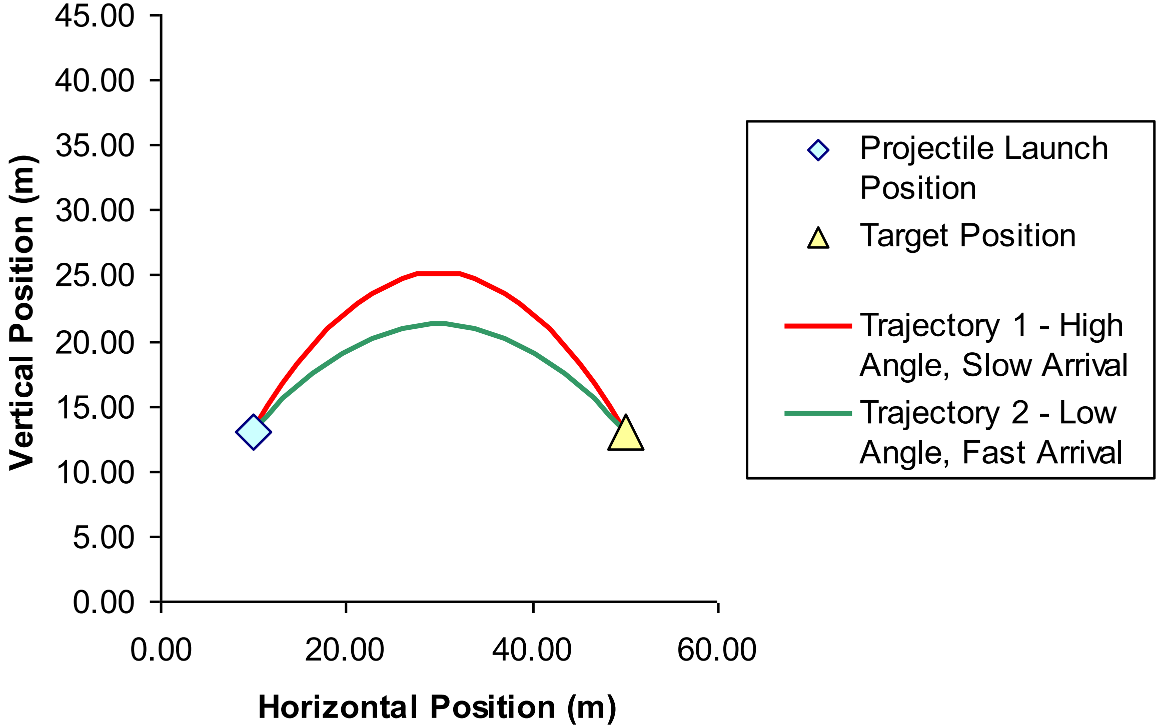
Target Practice – A Few Examples

V_{init} = 25 m/s
Value of Radicand of tanϕ equation: **969.31**
Launch angle ϕ: 19.4 deg or 70.6 deg



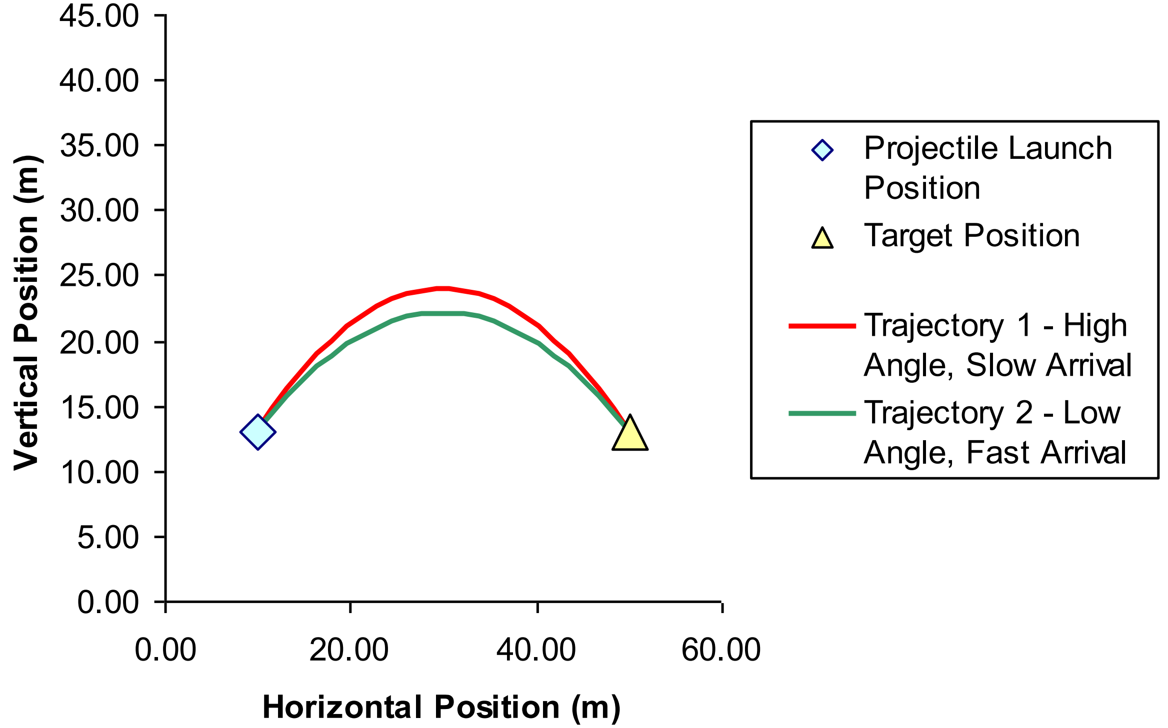
Target Practice – A Few Examples

V_{init} = 20 m/s
Value of Radicand of tanϕ equation: **60.2**
Launch angle ϕ: 39.4 deg or 50.6 deg



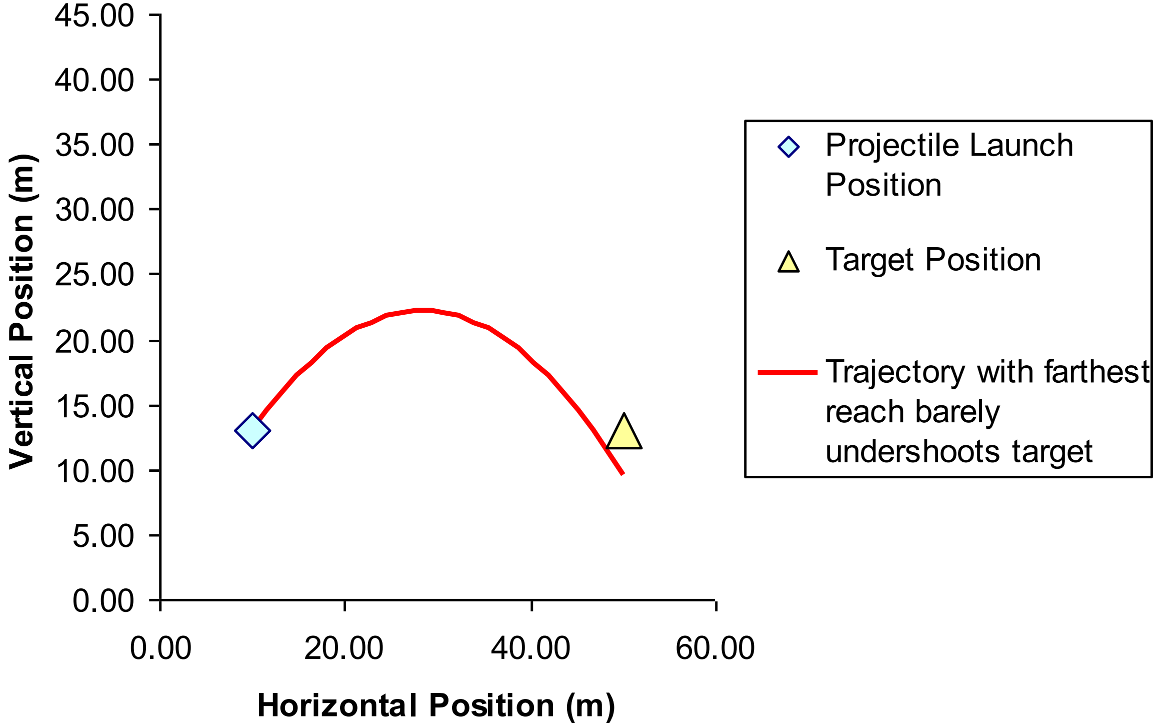
Target Practice – A Few Examples

V_{init} = 19.85 m/s
Value of Radicand of tanϕ equation: **13.2**
Launch angle ϕ: 42.4 deg or 47.6 deg (note convergence)



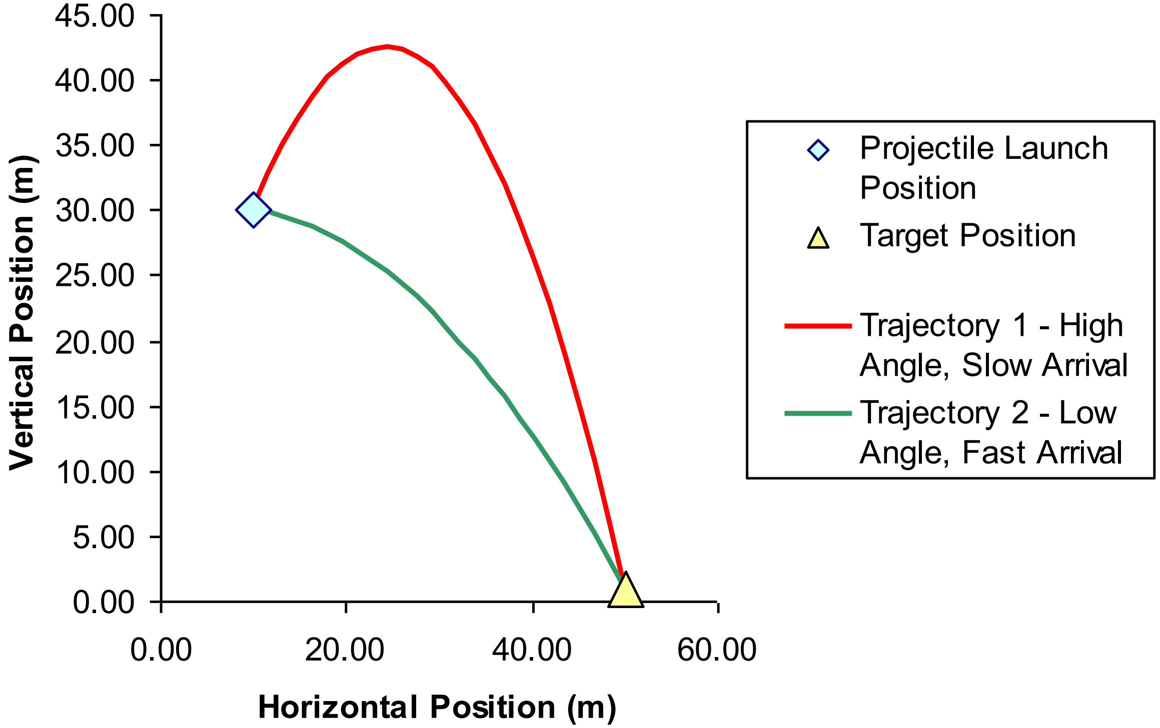
Target Practice – A Few Examples

V_{init} = 19 m/s
Value of Radicand of tanϕ equation: **-290.4**
Launch angle ϕ: No solution! V_{init} too small to reach target!



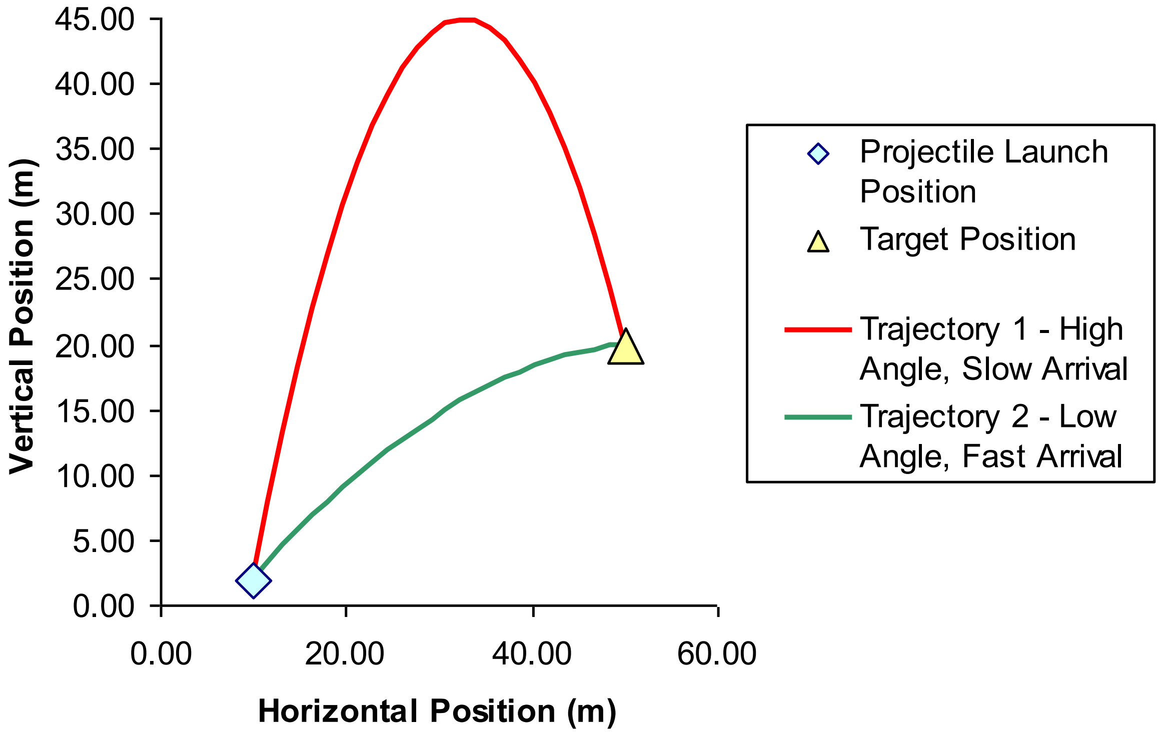
Target Practice – A Few Examples

V_{init} = 18 m/s
Value of Radicand of tanϕ equation: **2063**
Launch angle ϕ: -6.38 deg or 60.4 deg



Target Practice – A Few Examples

V_{init} = 30 m/s
Value of Radicand of tanϕ equation: **668**
Launch angle ϕ: 39.1 deg or 75.2 deg



- Stop Here

Real-time Game Physics

Practical Implementation:
Numerical Simulation

What is Numerical Simulation?

- Equations Presented Above
  - They are “closed-form”
  - Valid and exact for constant applied force
  - Do not require time-stepping
    - Just determine current game time, t, using system timer
      - *e.g.*, t = QueryPerformanceCounter / QueryPerformanceFrequency or equivalent on Microsoft® Windows® platforms
    - Plug t and t_{init} into the equations
    - Equations produce identical, repeatable, stable results, for any time, t, regardless of CPU speed and frame rate

What is Numerical Simulation?

- The above sounds perfect
- Why not use those equations always?
  - Constant forces aren't very interesting
    - Simple projectiles only
  - Closed-form solutions rarely exist for interesting (non-constant) forces
- We need a way to deal when there is no closed-form solution…

Numerical Simulation represents a series of techniques for incrementally solving the equations of motion when forces applied to an object are not constant, or when otherwise there is no closed-form solution

Finite Difference Methods

- What are They?
  - The most common family of numerical techniques for rigid-body dynamics simulation
  - Incremental "solution" to equations of motion
  - Derived using truncated Taylor Series expansions
  - See text for a more detailed introduction
- "Numerical Integrator"
  - This is what we generically call a finite difference equation that generates a "solution" over time

Finite Difference Methods

- The ***Explicit Euler*** Integrator:

$$\underbrace{\mathbf{S}(t+\Delta t)}_{\text{new state}} = \underbrace{\mathbf{S}(t)}_{\text{prior state}} + \Delta t \underbrace{\frac{d}{dt}\mathbf{S}(t)}_{\text{state derivative}}$$

- Properties of object are stored in a state vector, $\mathbf{S}$
- Use the above integrator equation to incrementally update $\mathbf{S}$ over time as game progresses
- Must keep track of prior value of $\mathbf{S}$ in order to compute the new
- For Explicit Euler, one choice of state and state derivative for particle:

$$\mathbf{S} = \langle m\mathbf{V}, \mathbf{p} \rangle \qquad d\mathbf{S}/dt = \langle \mathbf{F}, \mathbf{V} \rangle$$

Explicit Euler Integration

V_{init} = 30 m/s
Launch angle, ϕ: 75.2 deg (slow arrival)
Launch angle, θ: 0 deg (motion in world xz plane)
Mass of projectile, m: 2.5 kg
Target at <50, 0, 20> meters

Time	Position (m)			Linear Momentum (kg-m/s)			Force (N)			Velocity (m/s)		
	p_x	p_y	p_z	mV_x	mV_y	mV_z	F_x	F_y	F_z	V_x	V_y	V_z
5.00	10.00	0.00	2.00	19.20	0.00	72.50	0.00	0.00	-24.53	7.68	0.00	29.00

t_{init} $\quad$ $\mathbf{p}_{init}$ $\quad$ $m\mathbf{V}_{init}$ $\quad$ $\mathbf{F}$=Weight = $m\mathbf{g}$ $\quad$ $\mathbf{V}_{init}$

$$\mathbf{S} = \langle m\mathbf{V}_{init}, \mathbf{p}_{init} \rangle$$

$$d\mathbf{S}/dt = \langle m\mathbf{g}, \mathrm{V}_{init} \rangle$$

Explicit Euler Integration

$$\mathbf{S}(t+\Delta t) = \mathbf{S}(t) + \Delta t \frac{d}{dt}\mathbf{S}(t) = \begin{bmatrix} 19.2 \\ 0.0 \\ 72.5 \\ 10.0 \\ 0.0 \\ 2.0 \end{bmatrix} + \Delta t \begin{bmatrix} 0.0 \\ 0.0 \\ -24.53 \\ 7.68 \\ 0.0 \\ 29.0 \end{bmatrix}$$

$\Delta t = .2\ s$	$\Delta t = .1\ s$	$\Delta t = .01\ s$
$= \begin{bmatrix} 19.2025 \\ 0.0 \\ 67.5951 \\ 11.5362 \\ 0.0 \end{bmatrix}$	$= \begin{bmatrix} 19.2025 \\ 0.0 \\ 72.0476 \\ 10.7681 \\ 0.0 \end{bmatrix}$	$= \begin{bmatrix} 19.2025 \\ 0.0 \\ 72.2549 \\ 10.0768 \\ 0.0 \end{bmatrix}$

Exact, Closed-form Solution

$= \begin{bmatrix} 19.2 \\ 0.0 \\ 67.5951 \\ 11.5362 \\ 0.0 \\ 7.6038 \end{bmatrix}$	$= \begin{bmatrix} 19.2 \\ 0.0 \\ 72.0476 \\ 10.1536 \\ 0.0 \\ 4.8510 \end{bmatrix}$	$= \begin{bmatrix} 19.2 \\ 0.0 \\ 72.2549 \\ 10.0768 \\ 0.0 \\ 2.2895 \end{bmatrix}$

A Tangent: Truncation Error

- The previous slide highlights values in the numerical solution that are different from the exact, closed-form solution
- This difference between the exact solution and the numerical solution is primarily ***truncation error***
- Truncation error is equal and opposite to the value of terms that were removed from the Taylor Series expansion to produce the finite difference equation
- **Truncation error, left unchecked, can accumulate to cause simulation to become unstable**
  - This ultimately produces floating point overflow
  - Unstable simulations behave unpredictably

A Tangent: Truncation Error

- Controlling Truncation Error
  - Under certain circumstances, truncation error can become zero, *e.g.*, the finite difference equation produces the exact, correct result
    - For example, when zero force is applied
  - More often in practice, truncation error is nonzero
  - Approaches to control truncation error:
    - Reduce time step, Δt
    - Select a different numerical integrator
  - See text for more background information and references

Explicit Euler Integration – Truncation Error

Lets Look at Truncation Error (position only)

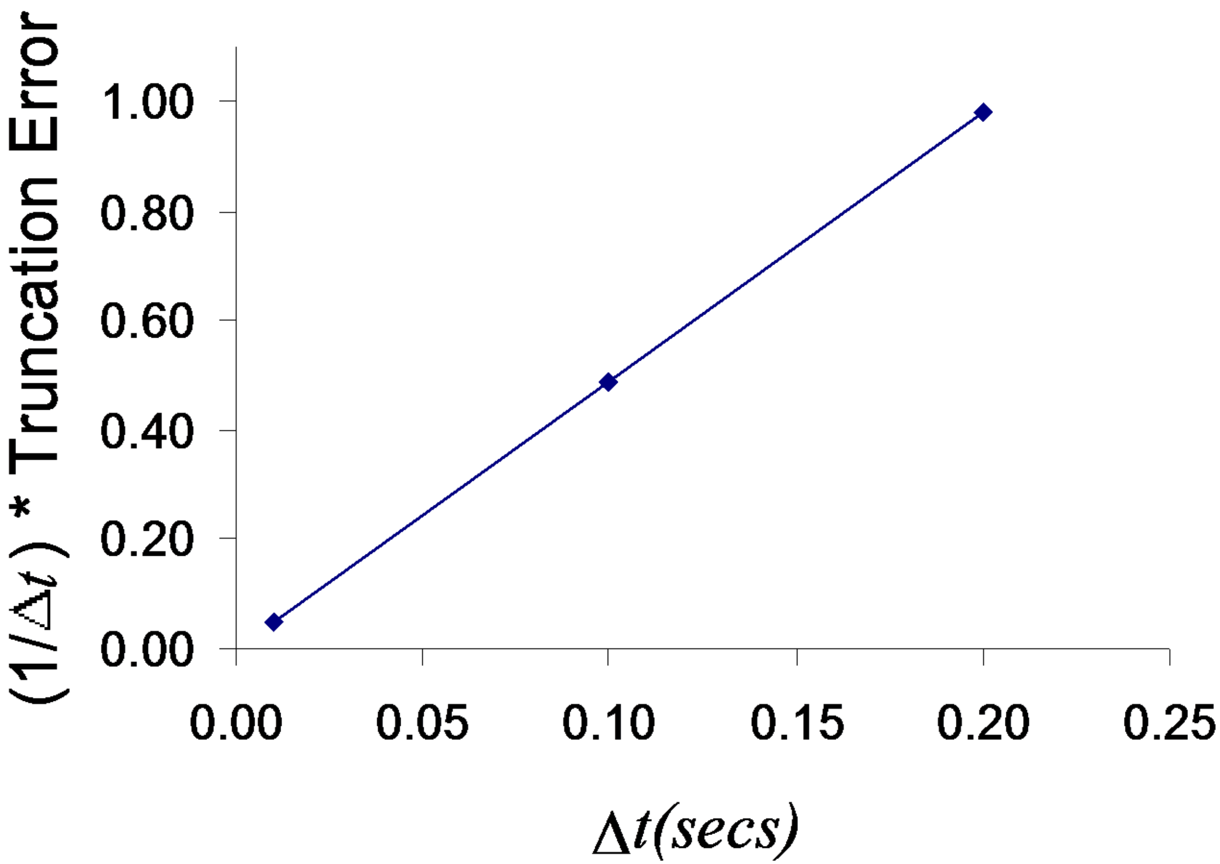
Truncation Error

$$\text{Truncation Error}(\Delta t = 0.2\text{s}) = \begin{bmatrix} 11.5362 \\ 0.0 \\ 7.800 \end{bmatrix}_{\text{numerical}} - \begin{bmatrix} 11.5362 \\ 0.0 \\ 7.6038 \end{bmatrix}_{\text{exact}} = \begin{bmatrix} 0.0 \\ 0.0 \\ 0.1962 \end{bmatrix}$$

$$\text{Truncation Error}(\Delta t = 0.1\text{s}) = \begin{bmatrix} 10.1536 \\ 0.0 \\ 4.9000 \end{bmatrix}_{\text{numerical}} - \begin{bmatrix} 10.1536 \\ 0.0 \\ 4.8510 \end{bmatrix}_{\text{exact}} = \begin{bmatrix} 0.0 \\ 0.0 \\ 0.049 \end{bmatrix}$$

$$\text{Truncation Error}(\Delta t = 0.01\text{s}) = \begin{bmatrix} 10.0768 \\ 0.0 \\ 2.2900 \end{bmatrix}_{\text{numerical}} - \begin{bmatrix} 10.0768 \\ 0.0 \\ 2.2895 \end{bmatrix}_{\text{exact}} = \begin{bmatrix} 0.0 \\ 0.0 \\ 0.0005 \end{bmatrix}$$

Explicit Euler Integration – Truncation Error



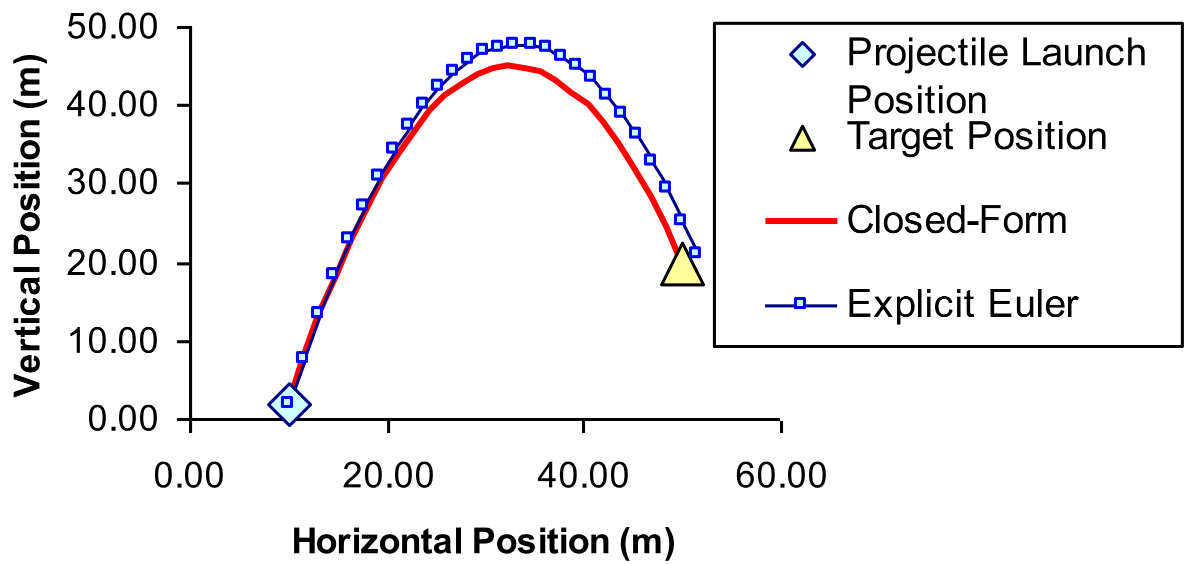
is a linear (first-
plicit Euler
rder-Accurate in

by "O(Δt)"

Explicit Euler Integration - Computing Solution Over Time

- The solution proceeds step-by-step, each time integrating from the prior state

Time	Position (m)			Linear Momentum (kg-m/s)			Force (N)			Velocity (m/s)		
	p_x	p_y	p_z	mV_x	mV_y	mV_z	F_x	F_y	F_z	V_x	V_y	V_z
5.00	10.00	0.00	2.00	19.20	0.00	72.50	0.00	0.00	-24.53	7.68	0.00	29.00
5.20	11.54	0.00	7.80	19.20	0.00	67.60	0.00	0.00	-24.53	7.68	0.00	27.04
5.40	13.07	0.00	13.21	19.20	0.00	62.69	0.00	0.00	-24.53	7.68	0.00	25.08
5.60	14.61	0.00	18.22	19.20	0.00	57.79	0.00	0.00	-24.53	7.68	0.00	23.11
⋮		⋮			⋮			⋮			⋮	
10.40	51.48	0.00	20.87	19.20	0.00	-59.93	0.00	0.00	-24.53	7.68	0.00	-23.97



Finite Difference Methods

- The ***Verlet*** Integrator:

$$\underbrace{\mathbf{S}(t+\Delta t)}_{\text{new state}} = 2\ \underbrace{\mathbf{S}(t)}_{\text{prior state 1}} - \underbrace{\mathbf{S}(t-\Delta t)}_{\text{prior state 2}} + (\Delta t)^2 \underbrace{\left(\frac{d^2}{dt^2}\mathbf{S}(t)\right)}_{\text{state derivative}}$$

- Must store state at two prior time steps, $\mathbf{S}(t)$ **and** $\mathbf{S}(t\text{-}\Delta t)$
- Uses second derivative of state instead of the first
- Valid for constant time step only (as shown above)
- For Verlet, choice of state and state derivative for a particle:

$$\mathbf{S} = \langle \mathbf{p} \rangle \qquad d^2\mathbf{S}/dt^2 = \langle \mathbf{F}/m \rangle = \langle \mathbf{a} \rangle$$

Verlet Integration

- Since Verlet requires two prior values of state, $\mathbf{S}(t)$ and $\mathbf{S}(t-\Delta t)$, you must use some method other than Verlet to produce the first numerical state after start of simulation, $\mathbf{S}(t_{init}+\Delta t)$
- Solution: Use explicit Euler integration to produce $\mathbf{S}(t_{init}+\Delta t)$, then Verlet for all subsequent time steps

Time	Position (m)			Acceleration (m/s^2)		
	p_x	p_y	p_z	a_x	a_y	a_z
5.00	10.00	0.00	2.00	0.00	0.00	-9.81
5.20	11.54	0.00	7.80	0.00	0.00	-9.81

$$\mathbf{S}(t_{init}),\quad \frac{d^2}{dt^2}\mathbf{S}(t_{init})$$

$$\mathbf{S}(t_{init}+\Delta t),\quad \frac{d^2}{dt^2}\mathbf{S}(t_{init}+\Delta t)$$

$\mathbf{p}$ $\mathbf{a}=\langle 0,0,-g\rangle$

$\mathbf{S} = \langle \mathbf{p} \rangle$ $d^2\mathbf{S}/dt^2 = \langle \mathbf{a} \rangle$

Verlet Integration

- The solution proceeds step-by-step, each time integrating from the prior two states

Time	Position (m) p_x	p_y	p_z	Acceleration (m/s^2) a_x	a_y	a_z
5.00	10.00	0.00	2.00	0.00	0.00	-9.81
5.20	11.54	0.00	7.80	0.00	0.00	-9.81
5.40	13.07	0.00	13.21	0.00	0.00	-9.81
5.60	14.61	0.00	18.22	0.00	0.00	-9.81
5.80	16.14	0.00	22.85	0.00	0.00	-9.81
6.00	17.68	0.00	27.08	0.00	0.00	-9.81
⋮	⋮			⋮		
10.40	51.48	0.00	20.87	0.00	0.00	-9.81

$\mathbf{S}(t-\Delta t)$ → 5.60 row

$\mathbf{S}(t)$ → 5.80 row

$\mathbf{S}(t+\Delta t)$ → 6.00 row

$\frac{d^2}{dt^2}\mathbf{S}(t)$ → 5.80 acceleration row

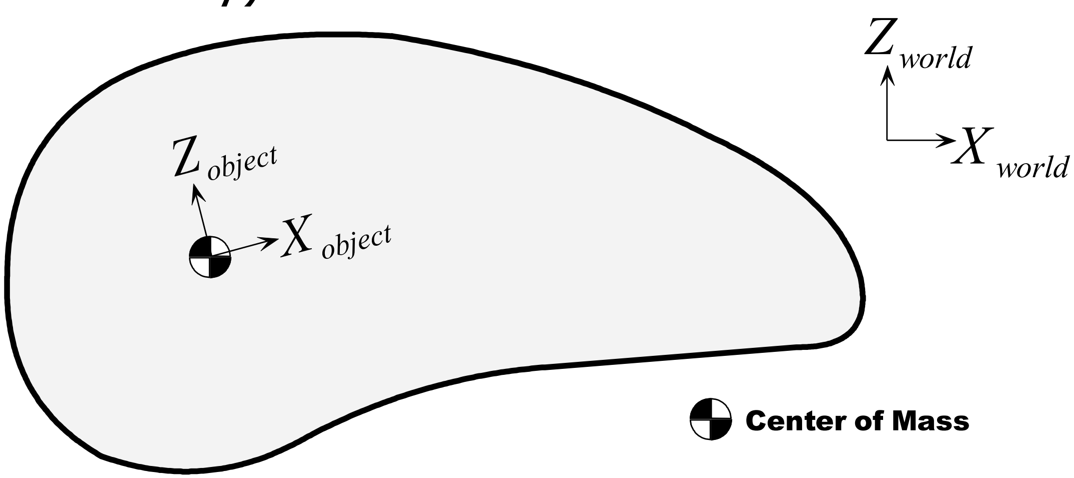
- For constant acceleration, Verlet integration produces results *identical* to those of explicit Euler
- But, results are different when non-constant forces are applied
- Verlet Integration tends to be more stable than explicit Euler for generalized forces

Real-time Game Physics

Generalized Rigid Bodies

Generalized Rigid Bodies

- Key Differences from Particles
  - Not necessarily spherical in shape
  - Position, $\mathbf{p}$, represents object's center-of-mass location
  - Surface may not be perfectly smooth
    - Friction forces may be present
  - Experience rotational motion in addition to translational (position only) motion

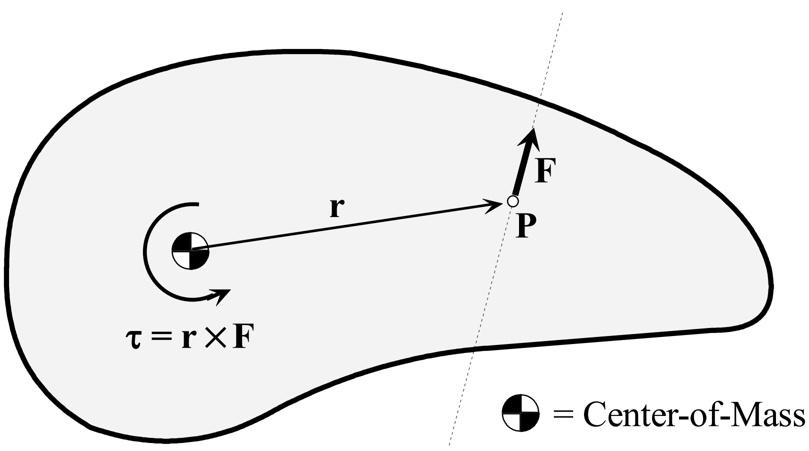


Generalized Rigid Bodies – Simulation

- Angular Kinematics
  - Orientation, 3x3 matrix $\mathbf{R}$ or quaternion, q
  - Angular velocity, $\boldsymbol{\omega}$
  - As with translational/particle kinematics, all properties are measured in world coordinates
- Additional Object Properties
  - Inertia tensor, $\mathbf{J}$
  - Center-of-mass
- Additional State Properties for Simulation
  - Orientation
  - Angular momentum, $\mathbf{L}=\mathbf{J}\boldsymbol{\omega}$
  - Corresponding state derivatives

Generalized Rigid Bodies - Simulation

- Torque
  - Analogous to a force
  - Causes rotational acceleration
    - Cause a change in angular momentum
  - Torque is the result of a force (friction, collision response, spring, damper, etc.)



Generalized Rigid Bodies – Numerical Simulation

- Using Finite Difference Integrators
  - Translational components of state <$m\mathbf{V}$, $\mathbf{p}$> are the same
  - $\mathbf{S}$ and $d\mathbf{S}/dt$ are expanded to include angular momentum and orientation, and their derivatives
  - Be careful about coordinate system representation for $\mathbf{J}$, $\mathbf{R}$, etc.
  - Otherwise, integration step is identical to the translation only case
- Additional Post-integration Steps
  - Adjust orientation for consistency
    - Adjust updated $\mathbf{R}$ to ensure it is orthogonal
    - Normalize q
  - Update angular velocity, $\boldsymbol{\omega}$
  - See text for more details

Collision Response

- Why?
  - Performed to keep objects from interpenetrating
  - To ensure behavior similar to real-world objects
- Two Basic Approaches
  - Approach 1: Instantaneous change of velocity at time of collision
    - Benefits:
      - Visually the objects never interpenetrate
      - Result is generated via closed-form equations, and is perfectly stable
    - Difficulties:
      - Precise detection of time and location of collision can be prohibitively expensive (frame rate killer)
      - Logic to manage state is complex

Collision Response

- Two Basic Approaches (continued)
  - Approach 2: Gradual change of velocity and position over time, following collision
    - Benefits
      - Does not require precise detection of time and location of collision
      - State management is easy
      - Potential to be more realistic, if meshes are adjusted to deform according to predicted interpenetration
    - Difficulties
      - Object interpenetration is likely, and parameters must be tweaked to manage this
      - Simulation can be subject to numerical instabilities, often requiring the use of implicit finite difference methods

Final Comments

- Instantaneous Collision Response
  - Classical approach: Impulse-momentum equations
    - See text for full details
- Gradual Collision Response
  - Classical approach: Penalty force methods
    - Resolve interpenetration over the course of a few integration steps
    - Penalty forces can wreak havoc on numerical integration
      - Instabilities galore
    - Implicit finite difference equations can handle it
      - But more difficult to code
  - Geometric approach: Ignore physical response equations
    - Enforce purely geometric constraints once interpenetration has occurred

Fixed Time Step Simulation

- Numerical simulation works best if the simulator uses a fixed time step
  - *e.g.*, choose Δt = 0.02 seconds for physics updates of 1/50 second
  - Do not change Δt to correspond to frame rate
  - Instead, write an inner loop that allows physics simulation to catch up with frame rate, or wait for frames to catch up with physics before continuing
  - This is easy to do
- Read the text for more details and references!

Final Comments

- Simple Games
  - Closed-form particle equations may be all you need
  - Numerical particle simulation adds flexibility without much coding effort
  - Collision detection is probably the most difficult part of this
- Generalized Rigid Body Simulation
  - Includes rotational effects and interesting (non-constant) forces
  - See text for details on how to get started

Final Comments

- Full-Up Simulation
  - The text and this presentation just barely touch the surface
  - Additional considerations
    - Multiple simultaneous collision points
    - Articulating rigid body chains, with joints
    - Friction, rolling friction, friction during collision
    - Mechanically applied forces (motors, etc.)
    - Resting contact/stacking
    - Breakable objects
    - Soft bodies
    - Smoke, clouds, and other gases
    - Water, oil, and other fluids