Understanding a Physics Solver

Eric Christensen

GTC 2009

What is a Rigid Body?

- A non-deformable (ideal) representation of a solid object
- Mass (linear m, angular I)
- Velocity (linear v, angular w)
- Collision Geometry

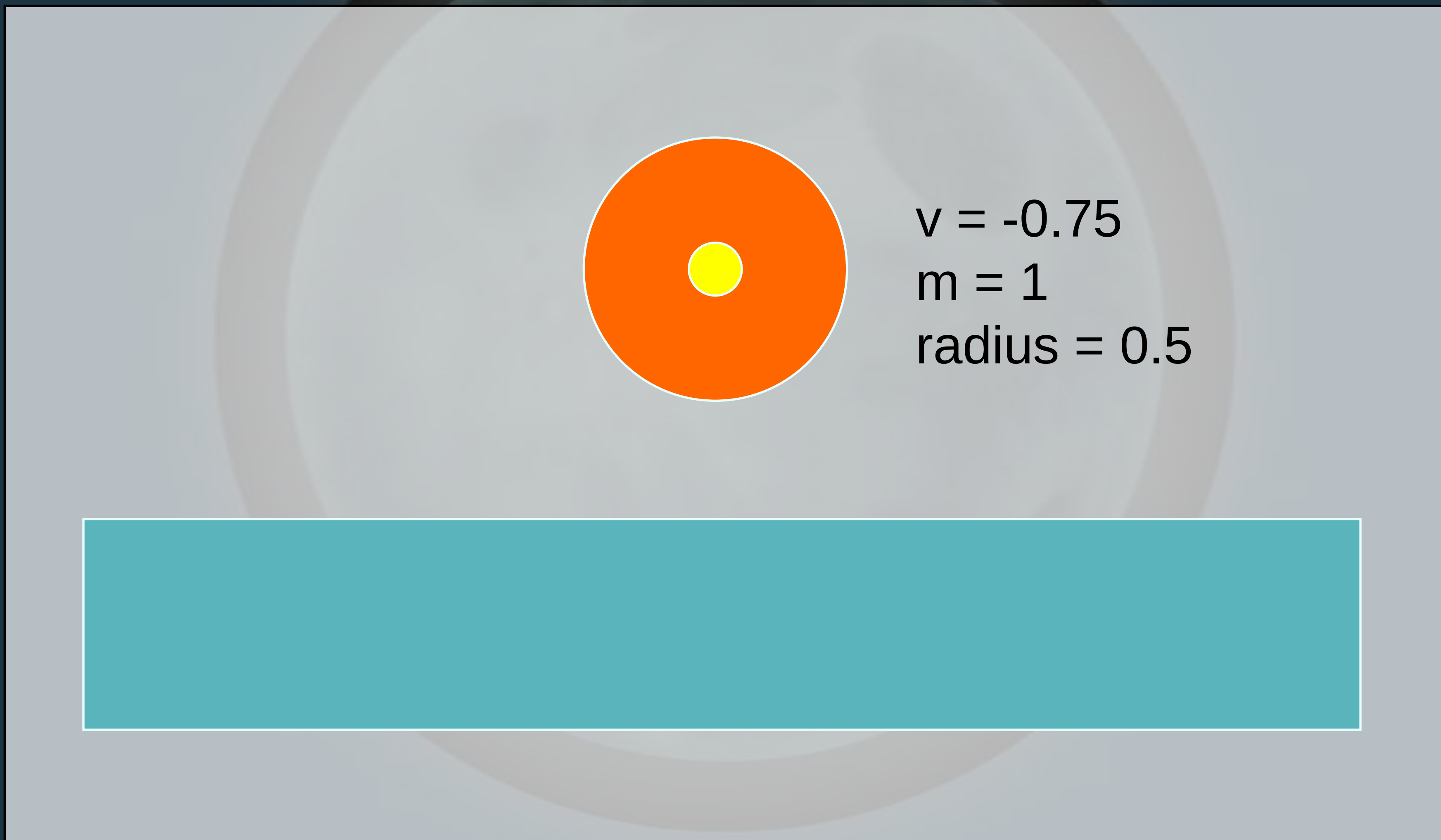
What is a Rigid Body?

- Starts stationary and remains that way unless influenced by a force
- Will remain in constant motion unless influenced by another force
- Almost always under the influence of (accelerated by) gravity (in a game)

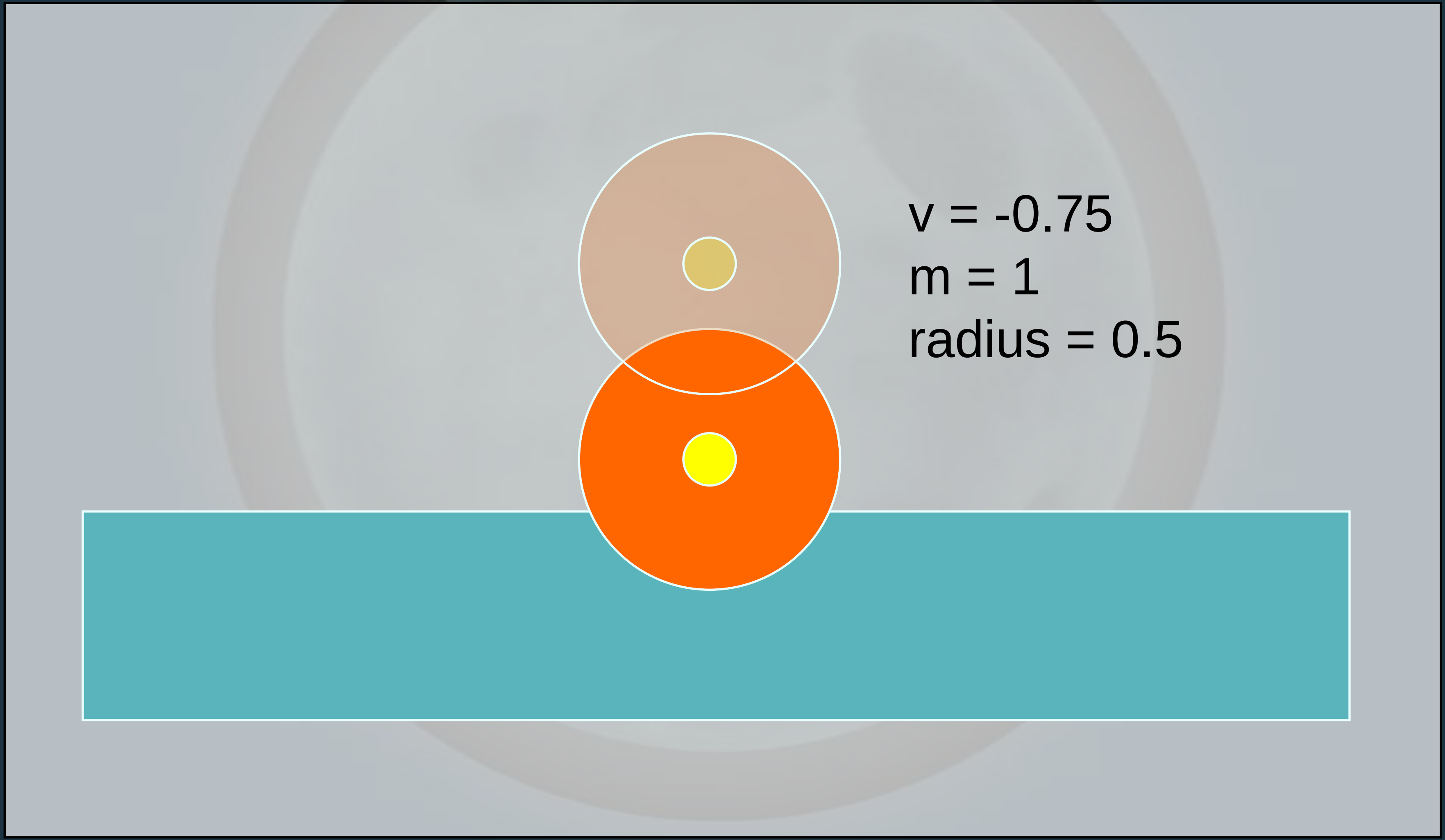
Collision Response

- Contact constraint generated by intersection of Rigid Body and Geometry
- Constraint consists of intersection normal, position, and depth
- Move rigid body to positive side of plane
- Ideally [v’ = v * normal + depth * normal]

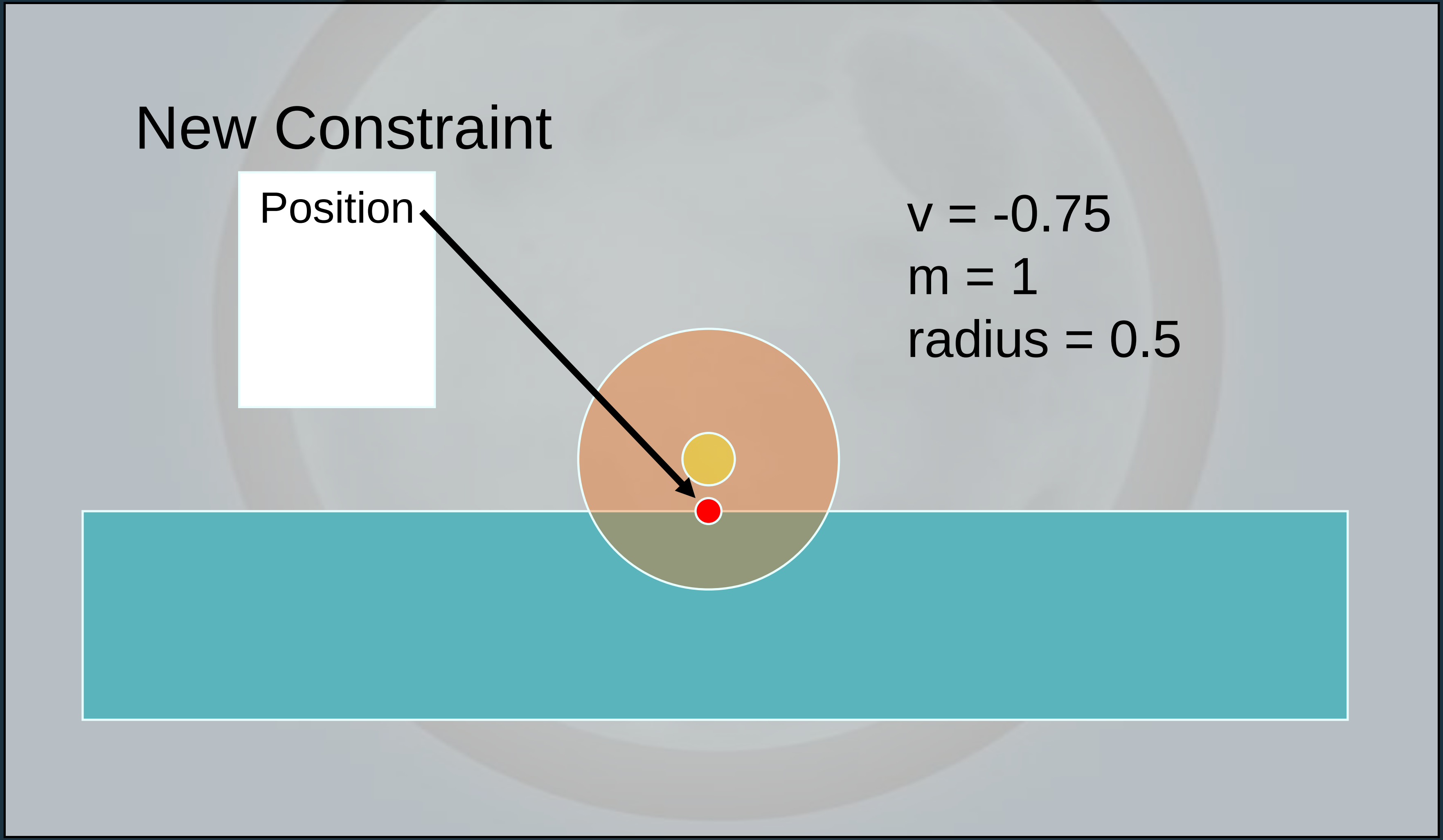
Collision Response



Collision Response

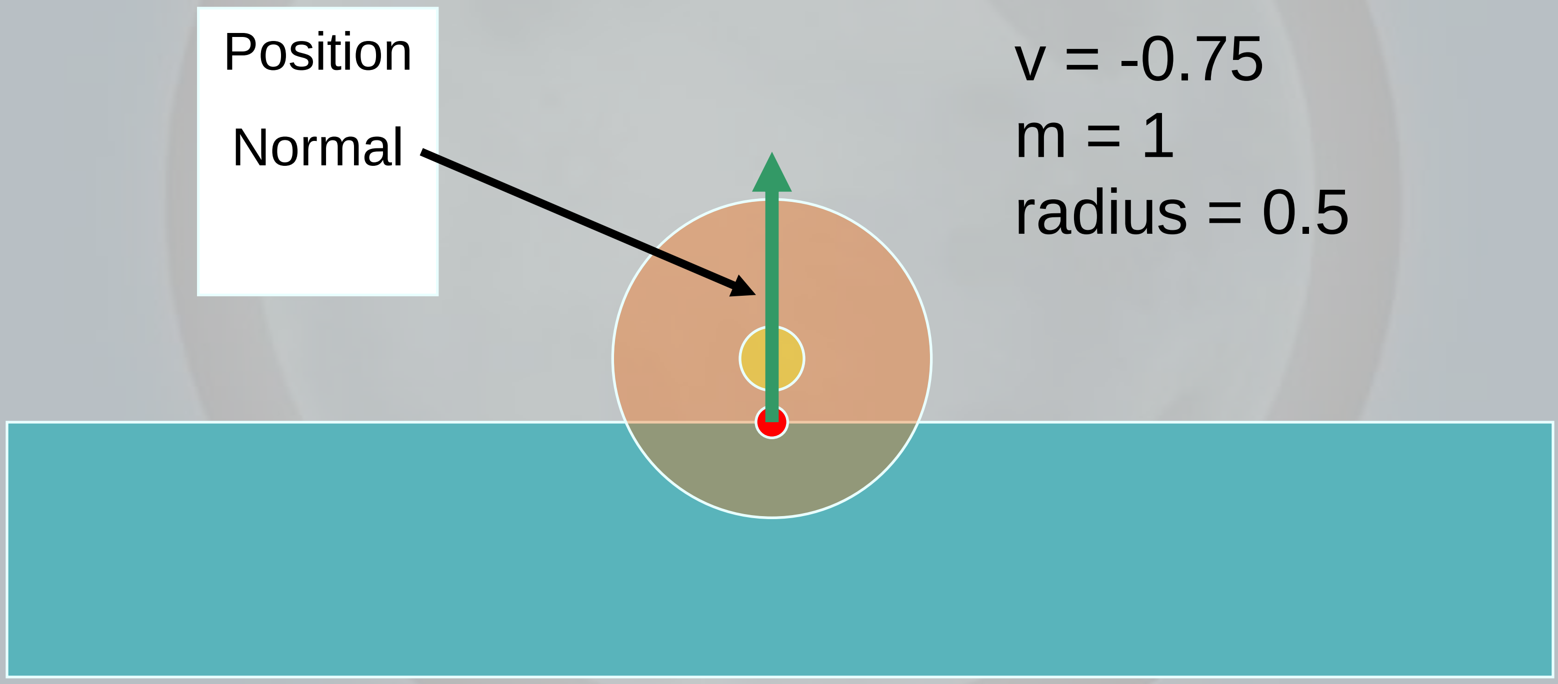


Collision Response



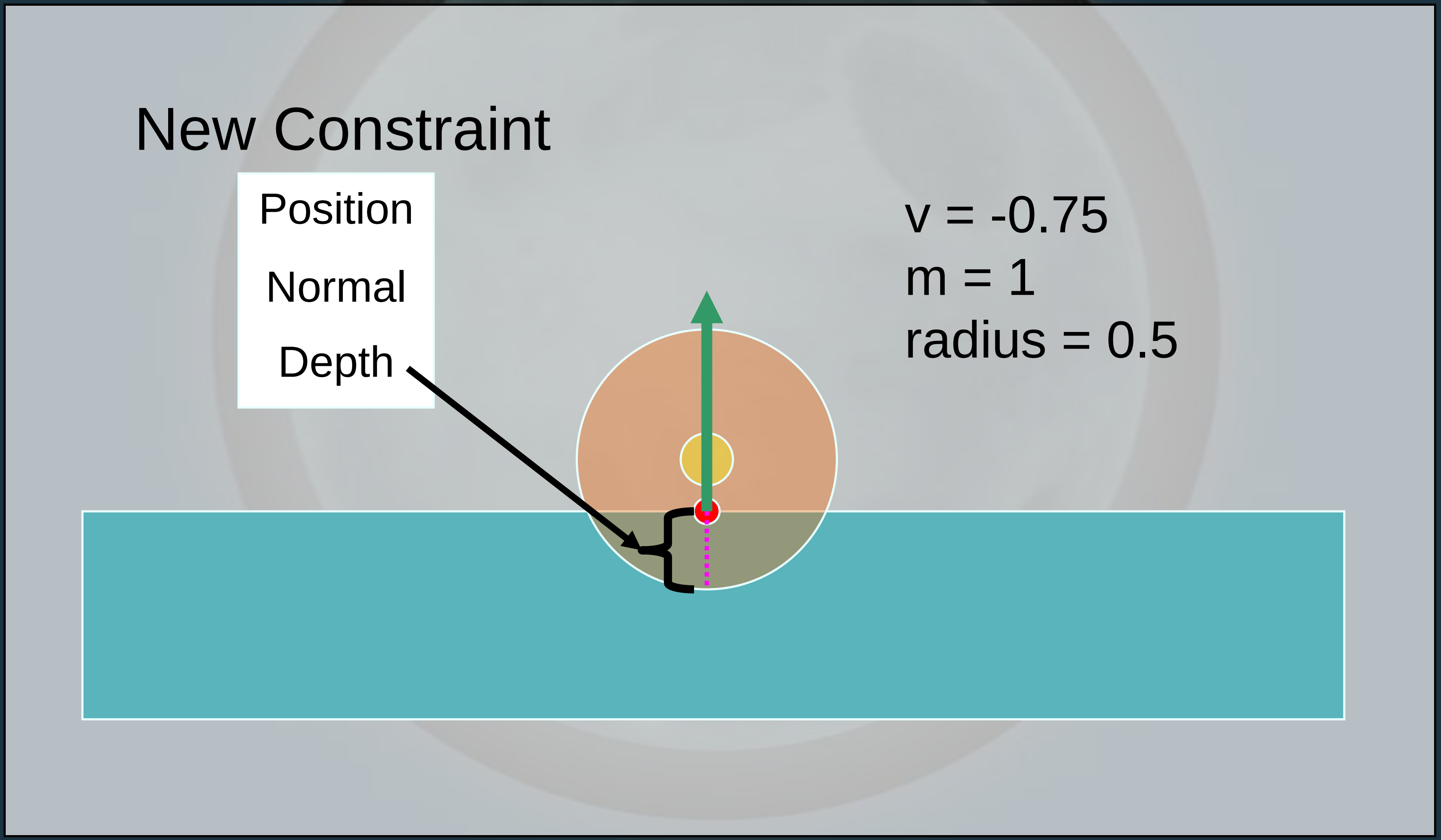
Collision Response

New Constraint



v = -0.75
m = 1
radius = 0.5

Collision Response

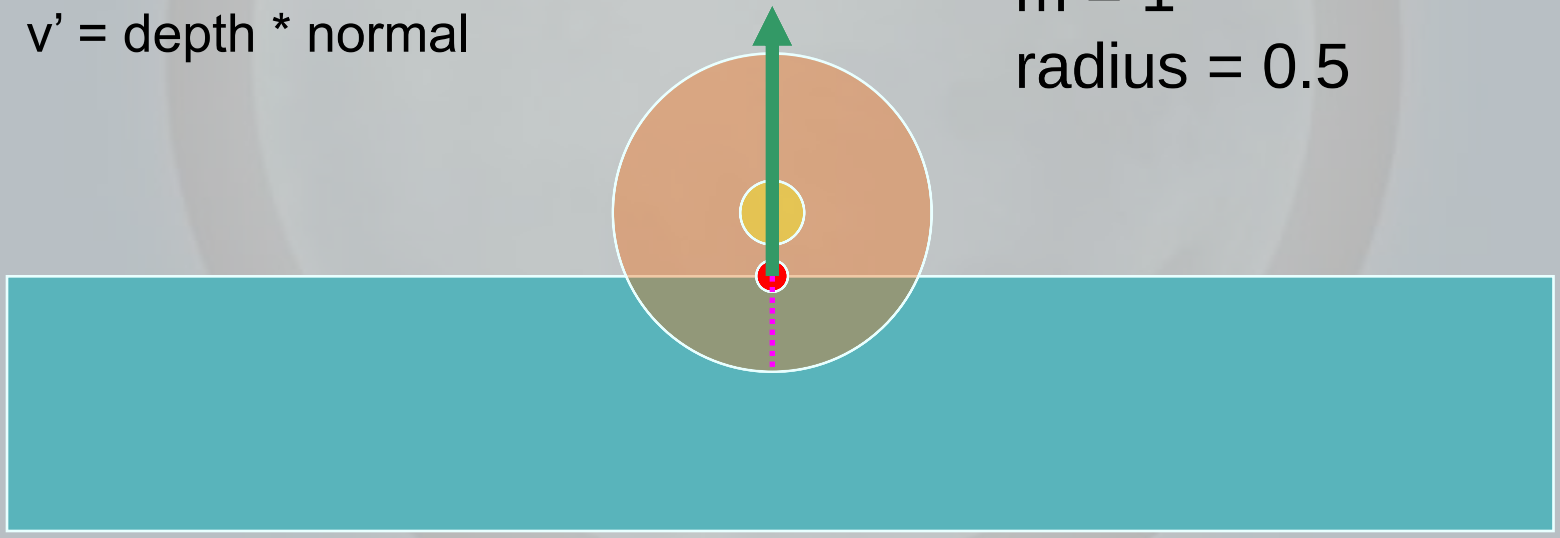


Collision Response

Simple Problem

Depth = 0.375
v’ = v * normal = 0
v’ = depth * normal

v = -0.75
m = 1
radius = 0.5

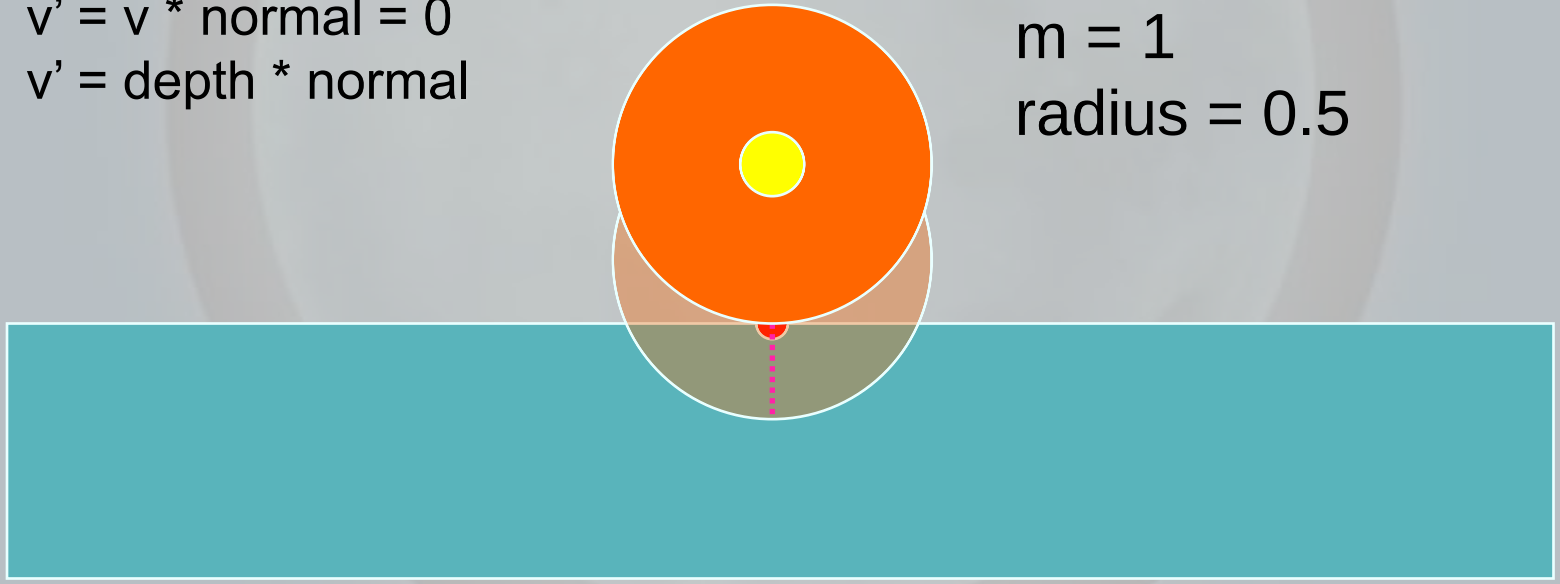


Collision Response

Simple Problem

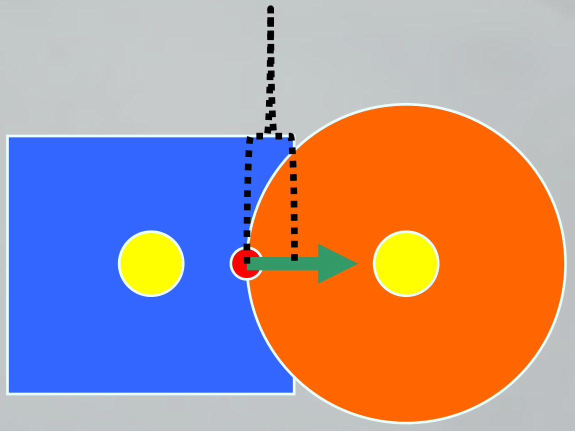
Depth = 0.375
v' = v * normal = 0
v' = depth * normal

v = 0.375
m = 1
radius = 0.5



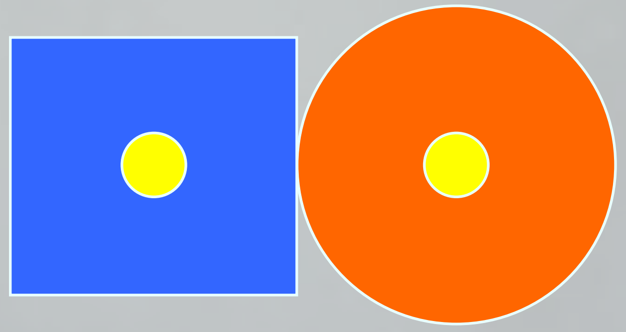
Collision Response

Same with two Rigid Bodies



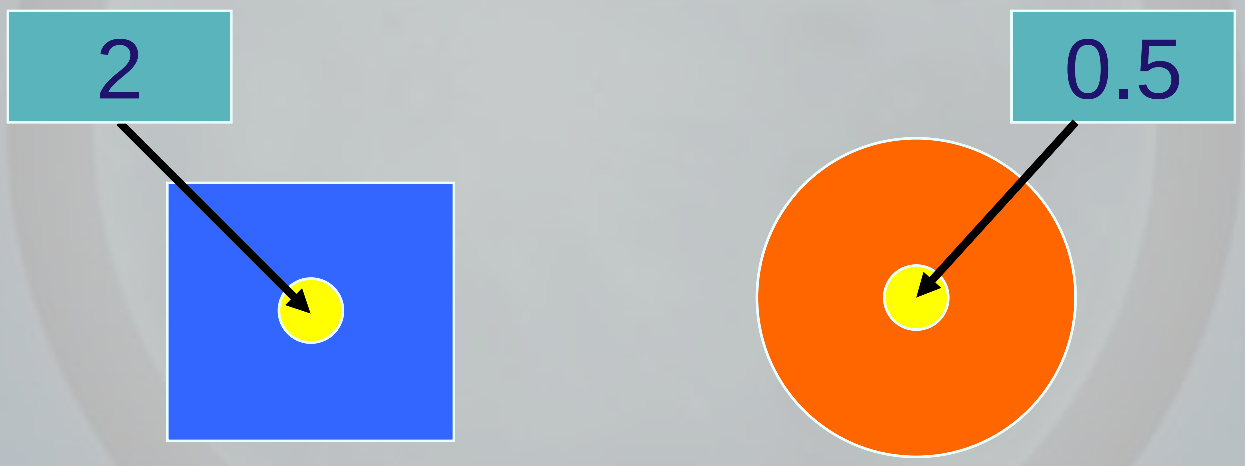
Collision Response

Same with two Rigid Bodies



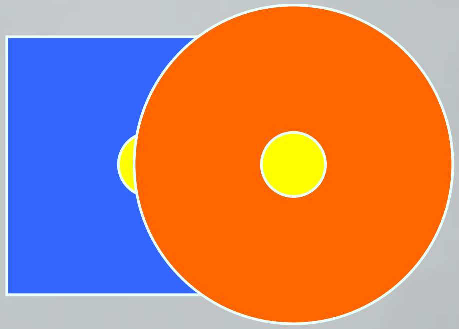
Collision Response

What if they have different mass?



Collision Response

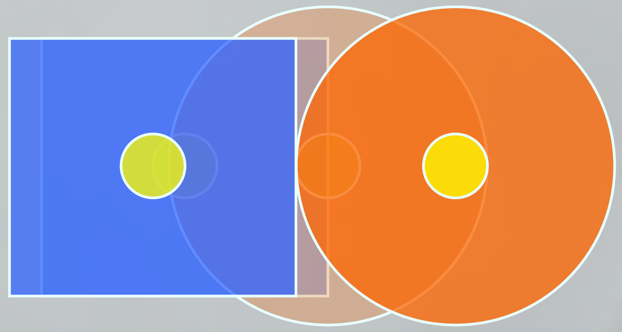
square will have 4 times the
“resistance to acceleration”
as the circle



if Constraint Force = F = 2
Circle Acceleration = ca = F / 0.5 = 4
Square Acceleration = sa = F / 2 = 1

Collision Response

square will have 4 times the
"resistance to acceleration"
as the circle



if Constraint Force = F = 2
Circle Acceleration = ca = F / 0.5 = 4
Square Acceleration = sa = F / 2 = 1

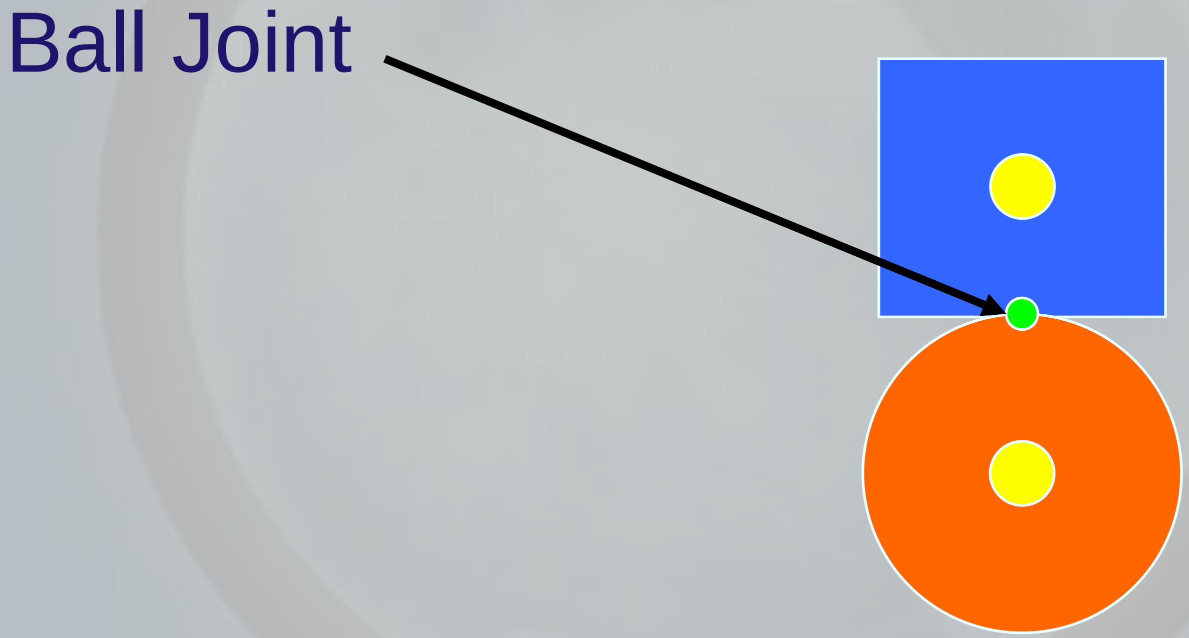
Collision Response
One or Two Rigid Bodies

- Easy to understand
- Correct intersection functions most difficult
- Good starting point
- No need for a solver

Joints

- Opposite collision constraints
- Binds rigid bodies together
- Binds rigid body to the world
- Limits rotation about one or more axis

Joints

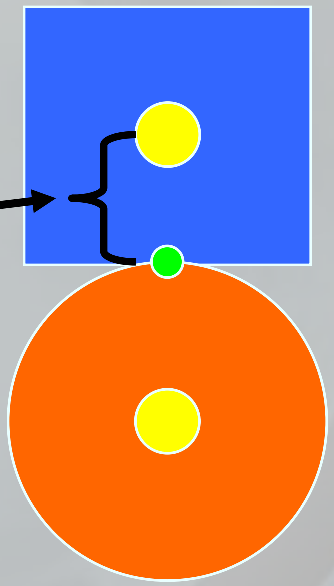


Joints

Ball Joint

Consists of

Local offset to body A



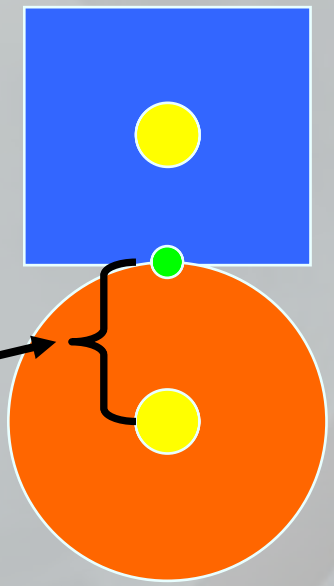
Joints

Ball Joint

Consists of

Local offset to body A

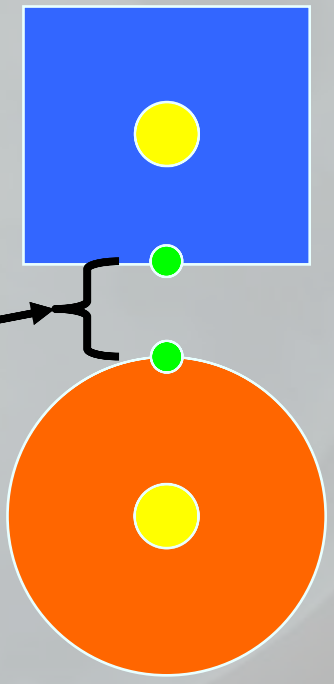
Local offset to body B



Joints

Ball Joint

When separated a force is created that moves the bodies back together

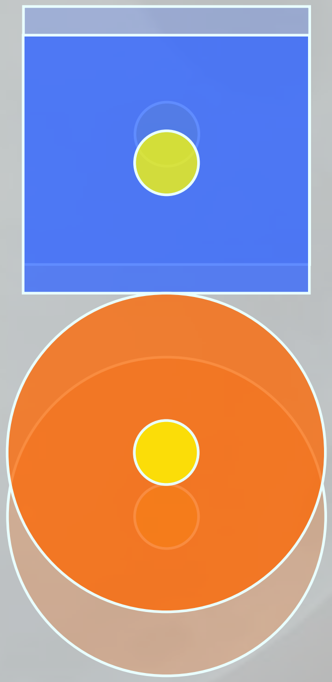


Joints

Ball Joint

Square still has a mass of 2

Circle still has a mass of 0.5



What is a physics constraint?

- Limits movement in a particular direction for one or two rigid bodies
- A condition that a solution to an optimization problem must satisfy

What is a physics constraint?

- Set of linear and angular components that describe a restriction on velocity
- The output is generally referred to as Jacobian data
- Removes one degree of freedom

What is Jacobian Data?

- Linear vector
- Angular vector
- Constraint force
- Upper and Lower boundaries
- Mixing Coefficient

How is the data used?

- Velocities are projected onto the linear and angular vectors
- The magnitudes are limited and force is added in the direction of the vectors
- Corrects error between one or two rigid bodies

Why use a solver?

- When number of interacting rigid bodies is high
- Constrained rigid bodies in a pool are dependent on each other
- Prevent catch-22 symptoms
- Stability

Why use a solver?

- Moving one rigid body requires moving all neighbor rigid bodies
- System requires “simultaneous” solution
- Special quadratic programming problem called Linear Complementarity

Benefits?

- We can use this solver for as many constraints as we want
- The more we iterate, the more accurate the solution is
- Solution is mostly smooth
- Can solve any type of constraint without modification

Limited to physics?

- No
- Fitting an internally constrained lattice to a height field
- Calculating optimal steering for a navigation group
- Fluid dynamics
- Any other system of dependant variables that have rules

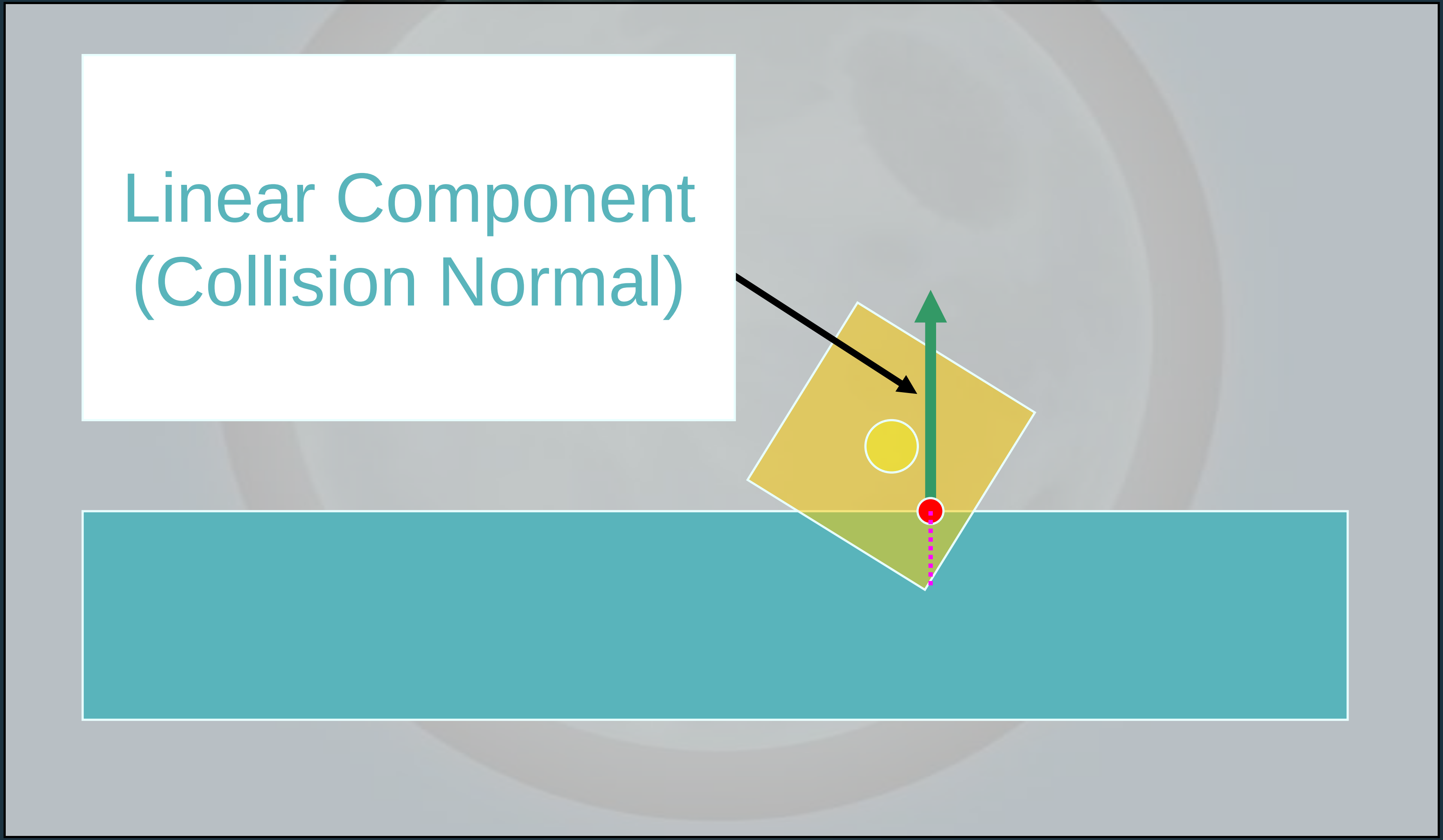
What is Jacobian Data

- Describes constraint along an axis
- Defines some rules about how to solve
- Most important piece of data when building a solution

Building Jacobian Data

- Linear Vector (constrains translation)
- Angular Vector (constrains rotation)
- Bounds (limits)
- Mix coefficient
- Constraint Force (error magnitude)

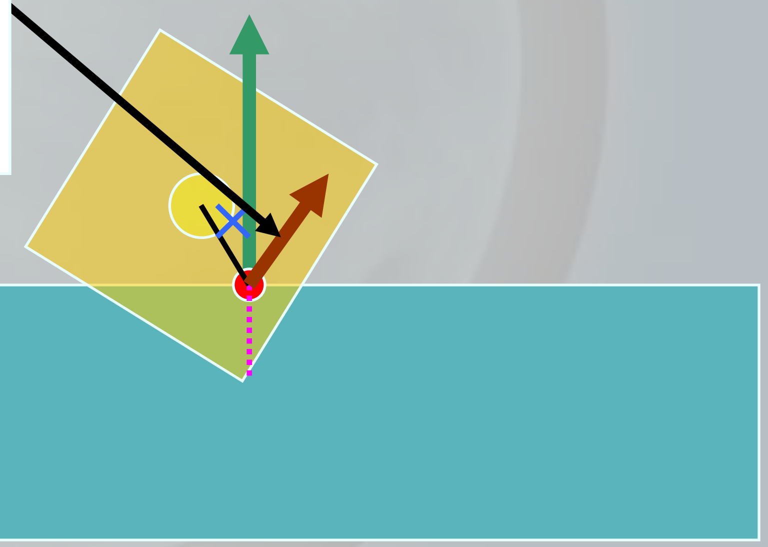
Building Jacobian Data



Building Jacobian Data

Angular Component

(Center Mass –
Intersection Point) cross
(Collision Normal)

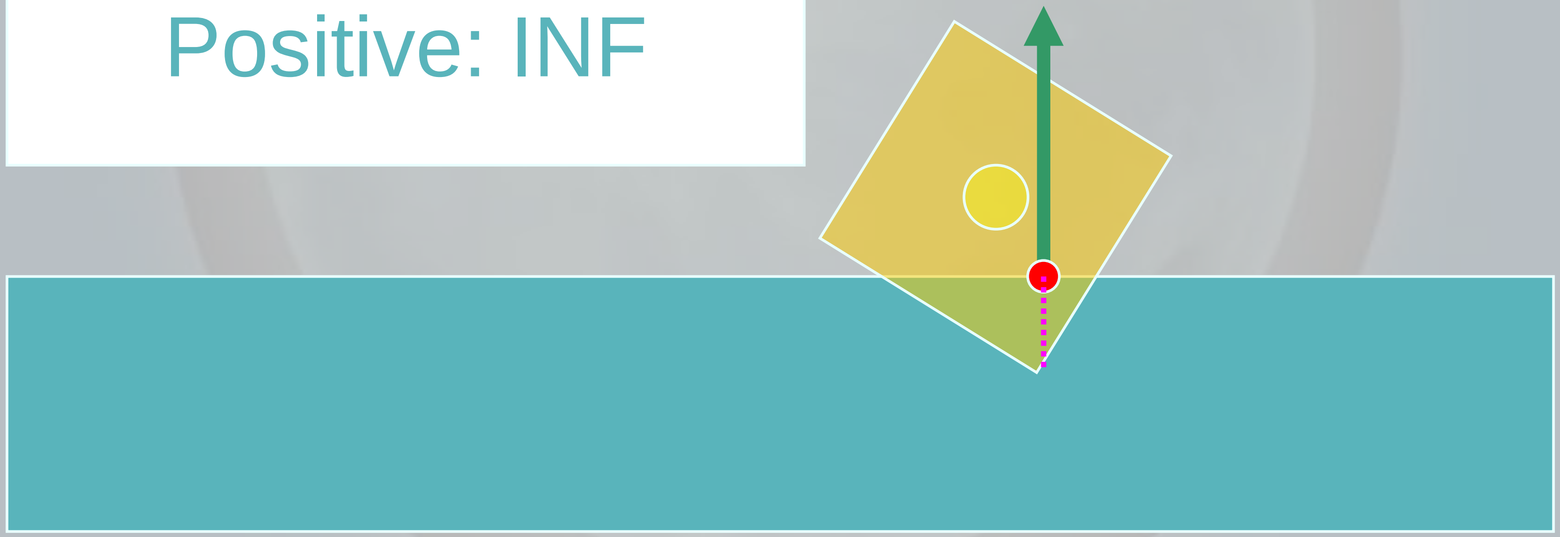


Building Jacobian Data

Bounds

Negative: 0

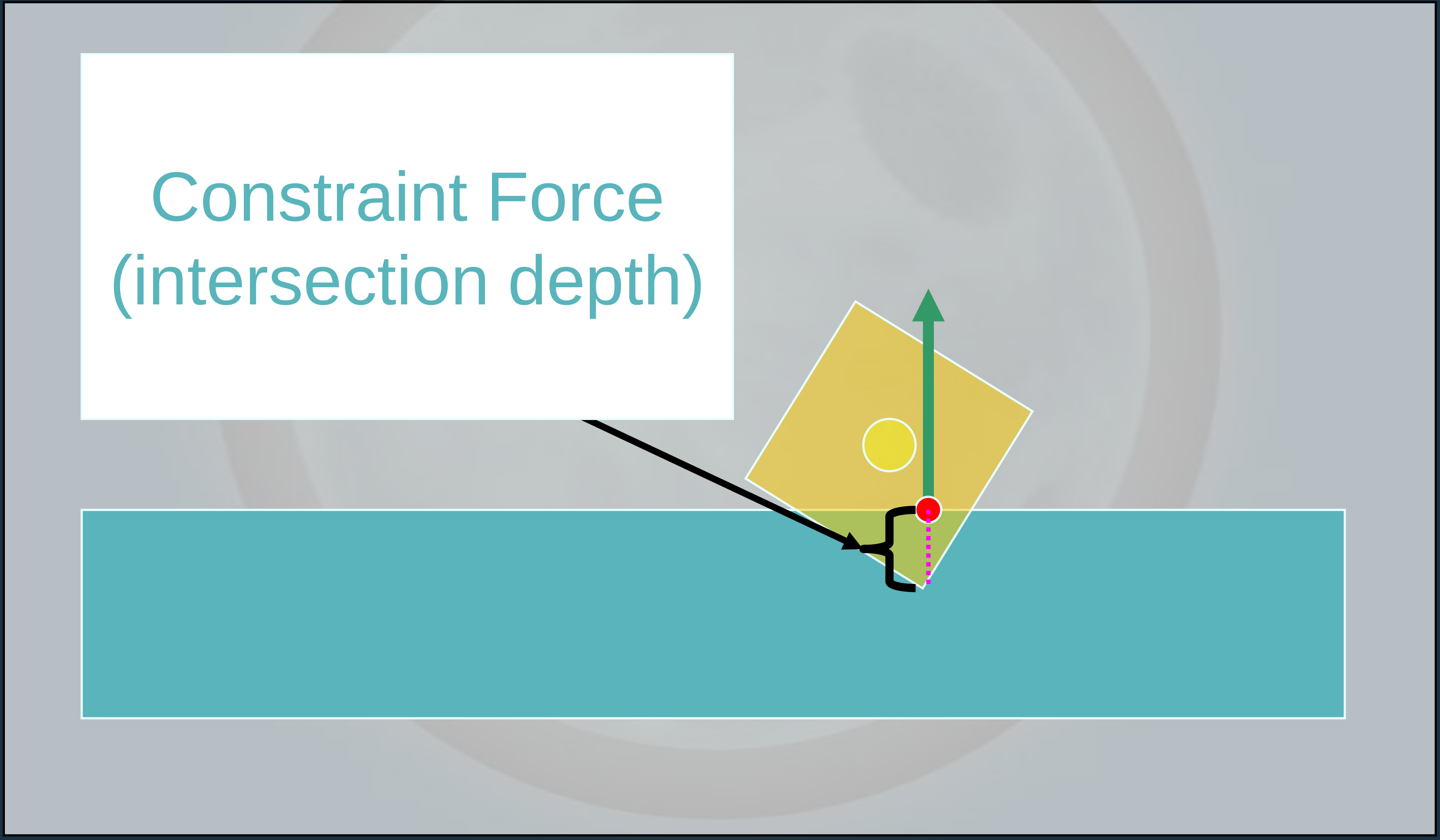
Positive: INF



Bounds (Limits)

- Defines which direction we can travel along component vectors
- Upper = INF, object allowed to move in the direction of normal by constraint force
- Lower = 0, object not allowed to move in the opposite direction of normal

Building Jacobian Data



Mixing Coefficient

- Allows constraint to “give way” to other constraints in the solution
- One way to soften the solution
- Can also scale down constraint force
- Good way to reduce jitter, if any

Preconditioning the Solver

- Ax=b
- Want to find x to satisfy b
- A bit more complicated because A[0]x[0]=b[0] is dependent on A[1]x[1]=b[0]

Setting up b for one rigid body

- Sum of velocity projections onto Jacobian Components
- u = Linear Velocity * Linear Jacobian Vector
- v = Angular Velocity * Angular Jacobian Vector
- b = u + v
- b’ = constraint force – b
- Tells us what our velocity must be to move to positive side

Mixing and Using Mass

- b is modified based on rigid body mass
- Reduces or Augments b for rigid body relative movement
- Heavier rigid body will move less than lighter rigid body

How?

- Create a value that scales our desired target velocity
- First need to multiply jacobian vectors by inverse mass and sum them
- We’ll call it JM^T
- JM^-1 = jacobian linear * mass^-1 + jacobian angular * Inertia Tensor^-1

How?

- Project for our scaling component
- We’ll call it D
- We’re actually constructing the diagonal component of A from Ax=b
- D = JM^-1 * jacobian linear + JM^-1 * jacobian angular
- For a mass of 1, D should be 1
- We also want to add the mixing coefficient

How?

- D’ = D + Mixing Coefficient
- We want to scale down so we get the reciprocal
- D’ = 1 / D
- If we wanted to mix, b should scale down
- b’ = b * D
- Our target velocity might have decreased by a small amount

How?

- Jacobian vectors now must be scaled by D
- This is because we want them to reflect our mass while projecting in the solver
- We must also scale D by our mixing coeff
- This tells the solver to “lie” about our delta thus “mixing”
- D’ = D * mixing coefficient

Now What?

- We now have our mass scaled linear and angular jacobian vectors
- D, our "A" diagonal
- b, our right hand side
- It's time to solve our problem

Solving the problem

- Some extra vectors to deal with
- L, our lambda
- Tells us how much we’ve “moved” in total, relative to our jacobian vectors
- d, our delta
- Tells us how much we’ve moved in a single iteration
- x, resulting force

Solving the Problem

- Loop over each constraint
- “d” has an initial value
- d = b – L * D
- If D = 0, we don’t mix, so lamda has no residual effect
- In this simple case, “d” is our desired movement (which satisfies “b”)

Solving the Problem

- “d” also reflects how much we have already moved
- So we must subtract our projected velocity
- d’ = d - velocity * jacobian linear
- d’ = d - velocity * jacobian angular
- “L” has accumulated how much we’ve moved
- Remember jacobian vectors scaled by mass
- We add “d” to “L” to test against bounds

Remembering Bounds

- Lower and Upper “limits”
- Tells us how we are allowed to move along jacobian vector
- We limit “d” and set “L” to reflect our limit
- L_test = d + L
- if L_test < lower_bounds then d = lower_bounds – L, L = lower_bounds

Delta

- We've modified delta
- Multiply by JM^-1
- Add to "x" (our resulting velocity)
- When the solution is complete, "x" will be added to rigid body velocity

Wash, Rinse, Repeat

- Each iteration brings us closer to a solution
- The solution tells us the best answer that will satisfy our constraints
- For large numbers of arbitrary constraints, solution will not be “perfect”
- It will be close enough for good stability

How much to iterate?

- Typically, for large systems, 10 to 20 iterations
- Mileage may vary
- Can I bail early?
- Yes, in a perfect world “d” will reduce to 0
- Track “d” and jump out when close to 0
- Or some desired epsilon

A Simple 1D Example

u	**Initial Velocity**
v	**Final Velocity**
J	**Jacobian Quantity**
J_s	**Jacobian Quantity Scaled by Diagonal**
b	**Right Hand Side**
μ	**Constraint Force Mixing Coefficient**
F^{ρ}	**Constraint Force**
$L_<$	**Lower Bounds (Lower Limit)**
$D_{e'}$	**Diagonal Conditioning Matrix (Pre-Scale)**
D_e	**Diagonal Conditioning Matrix (Scaled)**
λ	**Lambda**
Δ	**Delta**

A Simple 1D Example

$$u = -1.0$$

$$v = 0.0$$

$$J = 1.0$$

$$F^{\rho} = 5.0$$

$$L_{<} = 0.0$$

$$\omega = 1.0$$

$$\mu = 0.0001$$

$$\lambda = 0.0$$

$$D_{e'} = \frac{1.0}{JJ + \mu} = 0.99989998$$

$$b = D_{e'}(F^{p} - uJ) = 5.9994001$$

$$J_{s} = JD_{e'} = 0.99989998$$

$$D_{e} = D_{e'}\mu = 0.00009999$$

A Simple 1D example

$$\Delta = (b - \lambda D_e) - vJ_s$$

$$\lambda_- = \lambda + \Delta$$

if $\lambda_- < L_<$ **then**

$\Delta = L_< - \lambda$ **and**

$\lambda = L_<$

else

$\lambda = \lambda_1$

$$v = (J\Delta) + v$$

One Iteration (converged)

$$v = 5.9994001$$

Second Iteration (proof of convergence)

$$v = 5.9994001$$

Closing

- Try writing your own solver
- It doesn't matter if it is for physics
- Think of problems that don't have independent solutions
- Example source code is provided on our website
- Recommended Reading:

The Linear Complementarity Problem

By Richard W. Cottle, Jong-Shi Pang, and Richard E. Stone

Eric Christensen
Insomniac Games
Principal Engine Programmer
ec@insomniacgames.com