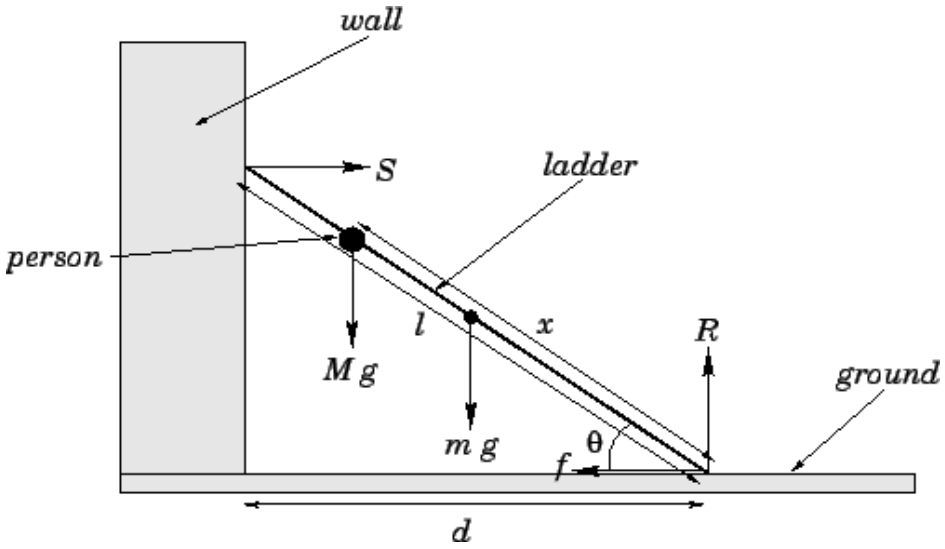
Physics Tutorial 3: Constraints



New Concepts

- Simple interfaces
- Constraints in 1 Dimension
- Constraints in 3 Dimensions
- Adding Energy to the System
- Calculating Lambda
- Updating Object Velocity

What is interface detection?

- We want to know if the region defined by one shape overlaps with a region defined by another
- There are several simple algorithms which determine this
- If we say that this cannot occur, then we are applying a **constraint** to the system

Collision Detection: Overview

- Collision detection, or interface detection/determination, is the second step of our physics loop
- We've moved our objects at this point (Yay, Newton!)
- What we need to know is whether or not we've moved them into an invalid configuration

Collision Detection: Overview

- Think about our Constraint-based approach again – we're saying "These are things you can't do, and so long as you don't do them you're fine".
- An invalid configuration is essentially breaking one of those rules – we're doing something that we've explicitly said we can't.
- And pretty much every case of this involves the intersection of objects – a collision. We know that can't happen in the real world, because we don't interpenetrate with our chair when we sit on it

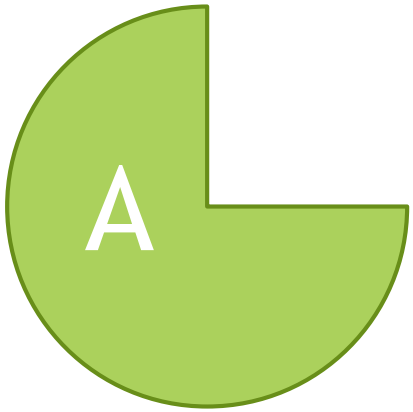
Collision Detection: Overview

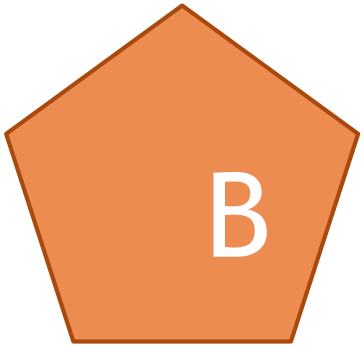
- Identifying these intersections is formally known as ‘interface detection’, and it has application far beyond video games
- The problem itself is one that, on the face of it, looks really simple – it’s a binary, something either does overlap, or it doesn’t – but computationally it’s far more complex than that
- Part of the reason for that complexity is that we don’t just need to know if objects have interfaced; we need to extract enough information about the interface to accurately ‘un-make’ it.

Collision Detection: The Question

- So, moving away from the general description of the problem, what's the real condition we're checking?

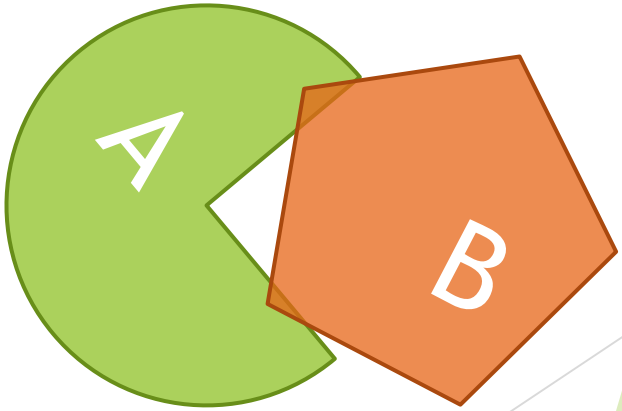
- Given two shapes, A and B, is there any point on the surface of A which exists inside the region defined by B?





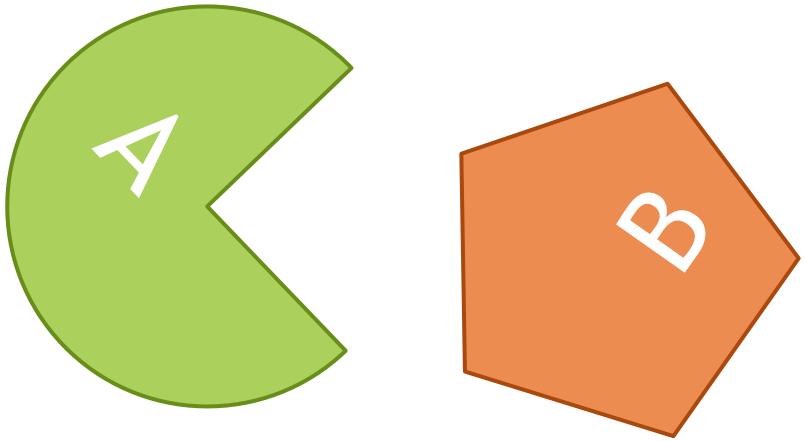
Collision Detection: The Question

- So, moving away from the general description of the problem, what's the real condition we're checking?
- Given two shapes, A and B, is there any point on the surface of A which exists inside the region defined by B?
- If TRUE, COLLISION DETECTED



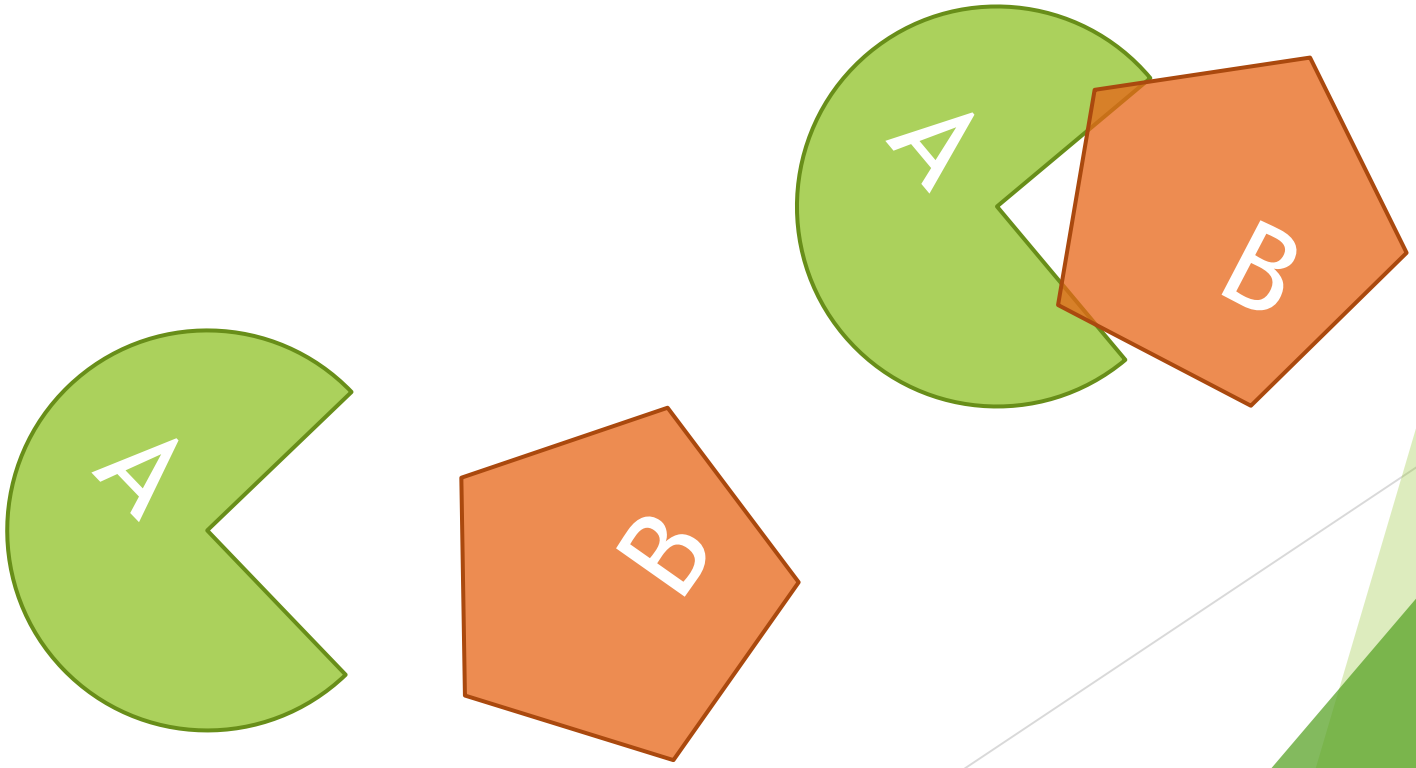
Collision Detection: The Question

- So, moving away from the general description of the problem, what's the real condition we're checking?
- Given two shapes, A and B, is there any point on the surface of A which exists inside the region defined by B?
- If FALSE, COLLISION NOT DETECTED



Collision Detection: The Question

- Looking at these examples, it should begin to get a little clearer as to how guaranteeing FALSE is the case might be easier than computing TRUE



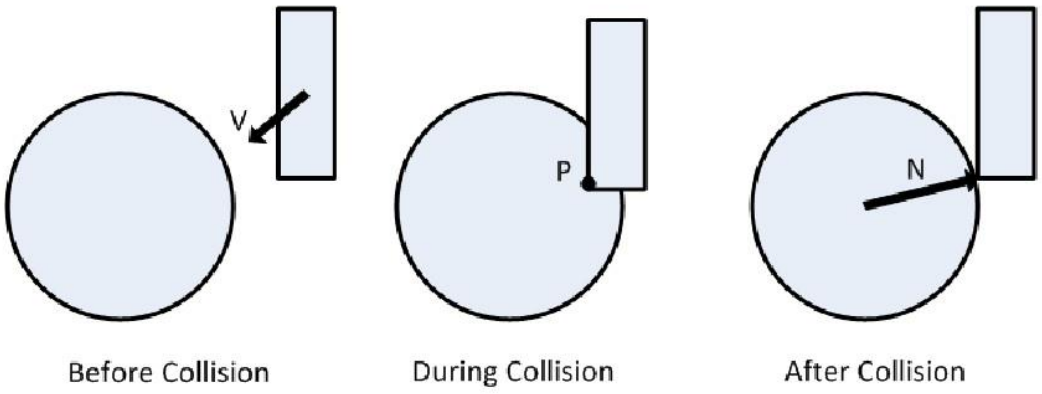
Another reason for exclusion rather than inclusion

- Extracting collision data
- Why this is important will become clear
- For now, just need to remember that the data exists
- Will be used to help satisfy our constraint – e.g., resolve a collision

Collision Data to Permit Resolution

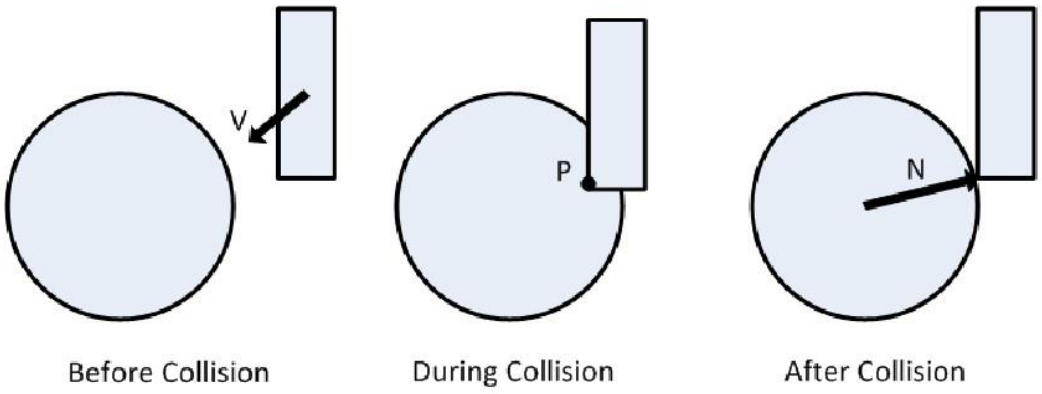
- For many of the simple algorithms we present today, three bits of data need extracting from the collision in order to resolve it.

- Consider the figure below:



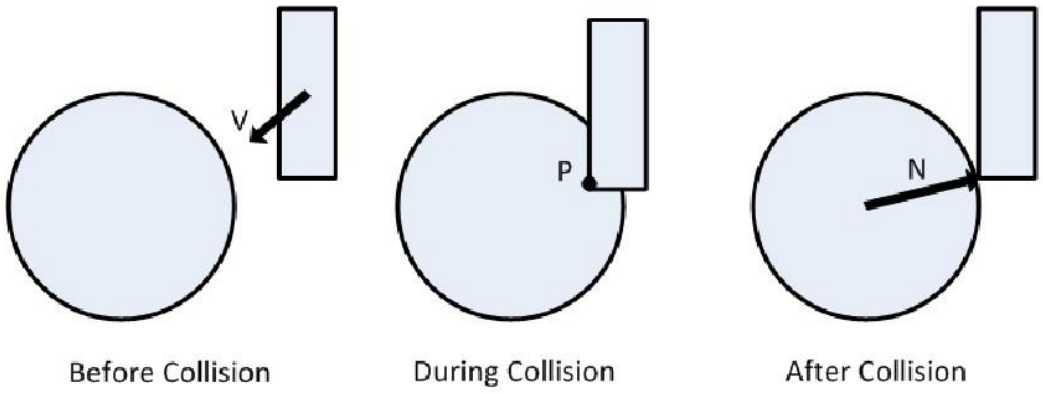
Collision Data to Permit Resolution

- The Contact Point *P* indicates the point where the intersection has been detected
- *N* is the collision normal – the direction along with the intersecting object must move to resolve the collision
- *p* is the penetration depth – the distance along *N* we need to move to resolve the collision



Collision Data to Permit Resolution

- It is ***crucial*** to note that for many collisions, when resolved by our advanced narrow phase algorithms, we must maintain a collision MANIFOLD, not a collision point. That will be the subject of a later tutorial.

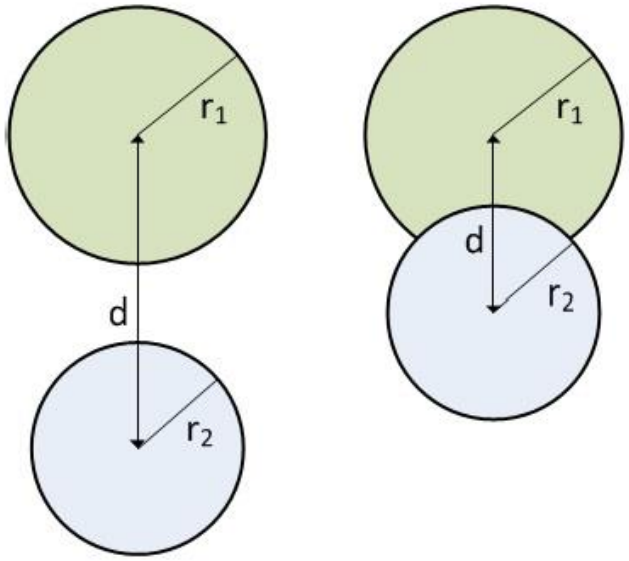


Sphere-Sphere Collision

- Consider again what collision detection is meant to determine.
- Does there exist a point on the surface of Object A which can be found within the volume defined by Object B?
- The sphere-sphere case is probably the simplest such check

Sphere-Sphere Collision

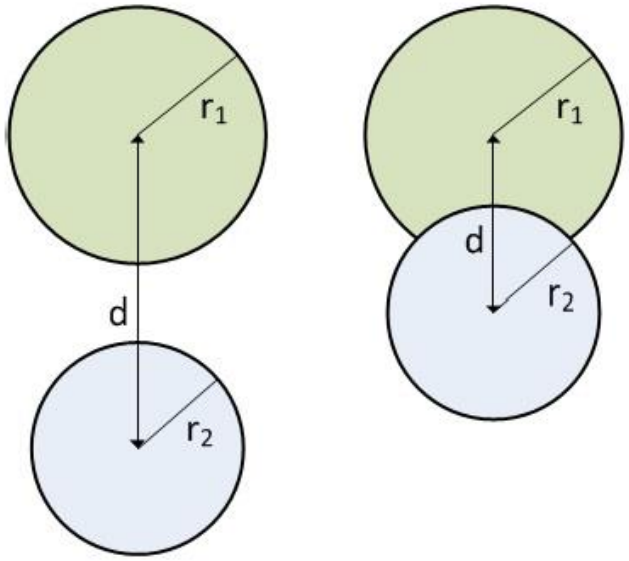
- Consider the following scenario:



Sphere-Sphere Collision

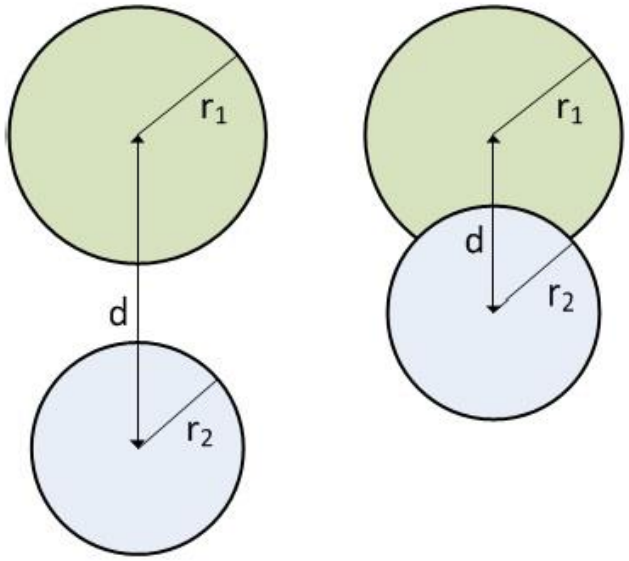
- Let d be the distance between the centres of our spheres
- Our collision condition must be true if

$$d < r_1 + r_2$$



Sphere-Sphere Collision

- This is because $r_1 + r_2$ is, logically, the smallest distance the centres of the spheres can lie apart without interfacing
- What does this algorithm become if $r_2 = 0$?



Sphere-Sphere Collision

- For this collision type, our collision data is determined as follows:

$$p = r_1 + r_2 - d$$
$$N = |S_1 - S_2|$$
$$P = S_1 - N(r_1 - p)$$

- Note that spheres are the only objects (with volume) which can collide and guarantee only one contact point

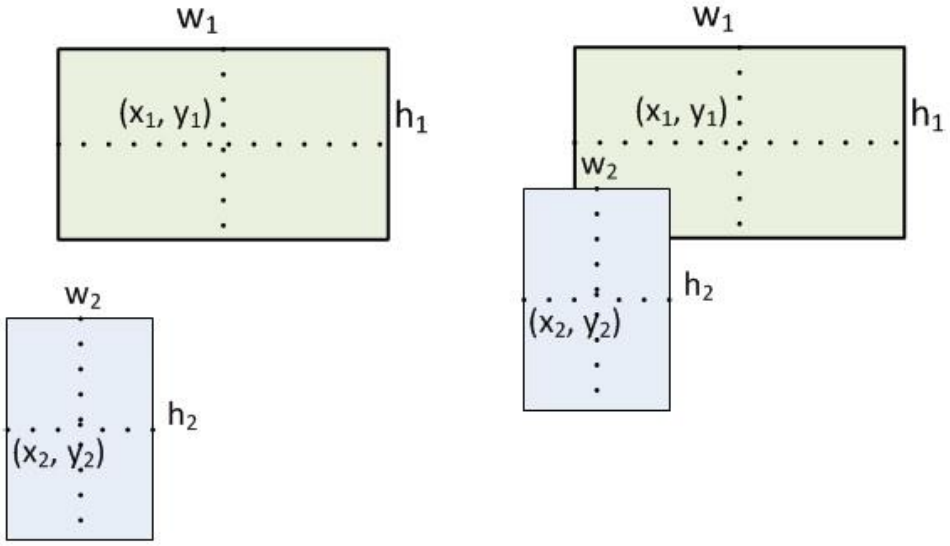
Axis-Aligned Bounding Box Collision Check

- Axis-Aligned Bounding Box (AABB) checks are a very straightforward method of determining if objects have the possibility of collision
- AABBs are normally employed solely in broad phase checks, as the algorithm does not provide collision data
- Limited by the fact that the box has to be axis-aligned – can't be object aligned. As such, objects represented by AABBs can require their AABBs to be dynamically resized.

Axis-Aligned Bounding Box Collision Check

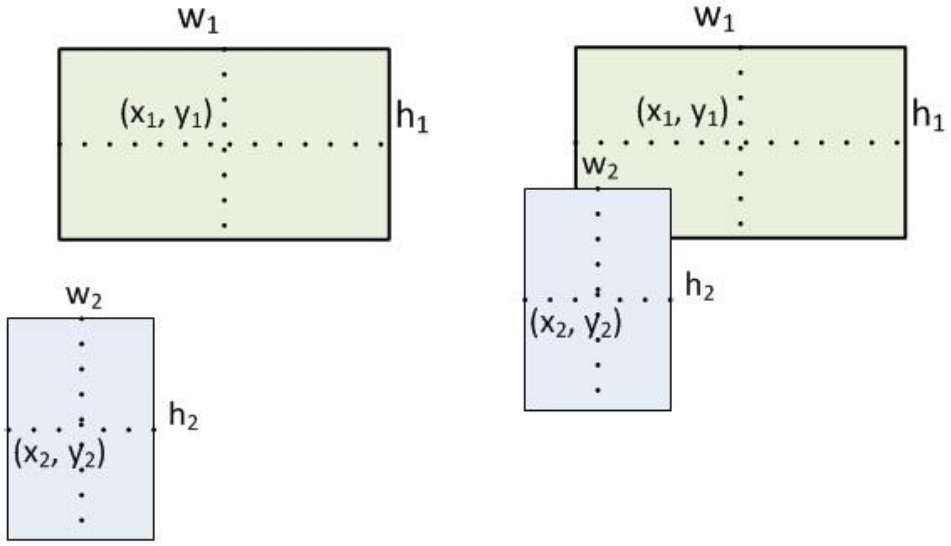
- Interface has occurred if ALL THREE conditions are met:

$$|x_2 - x_1| < \frac{1}{2}(w_1 + w_2)$$
$$|y_2 - y_1| < \frac{1}{2}(h_1 + h_2)$$
$$|z_2 - z_1| < \frac{1}{2}(l_1 + l_2)$$



Axis-Aligned Bounding Box Collision Check

- Note, the algorithm can be made more efficient by breaking out early with a return of FALSE if any of the conditions AREN'T met, without computing the subsequent ones



Sphere-Plane Collision

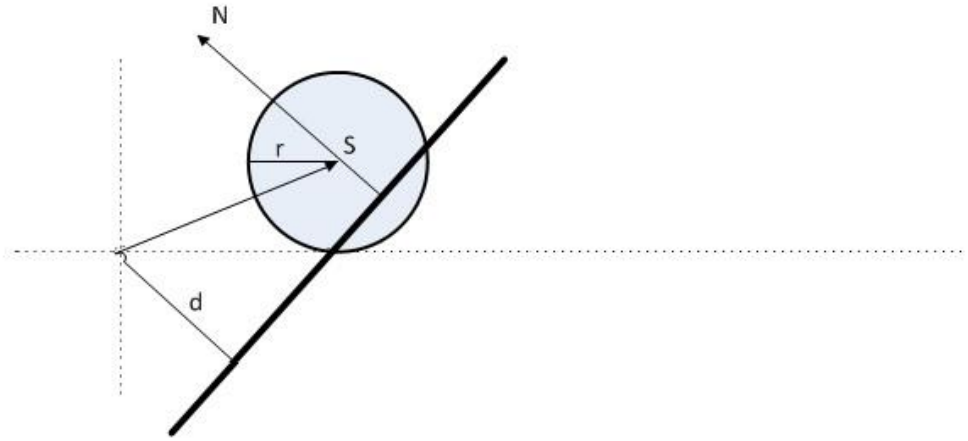
- This mode of detection is based on the plane equation

$$Ax + By + Cz + D = 0$$

- Where (A, B, C) is the normal to the plane, D is the distance of the plane from the origin $(0,0,0)$, and (x, y, z) is the position of the test point
- So, in the case of our sphere, the test point would be the point on its surface which is closest to the plane

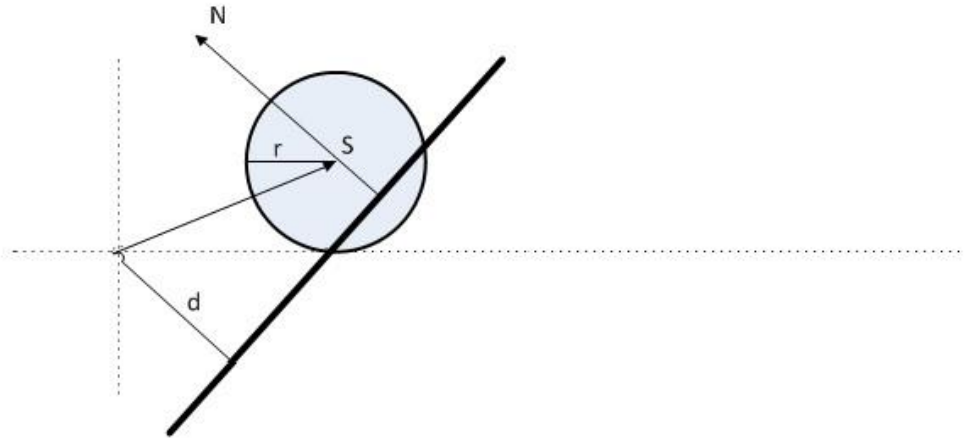
Sphere-Plane Collision

- We can rationalise out a lot of what happens in this situation – consider the scenario below:



Sphere-Plane Collision

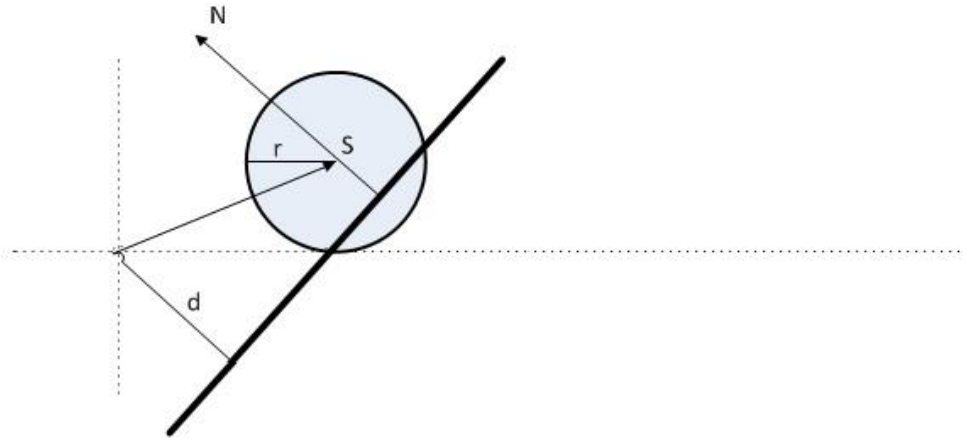
- The contact normal will always be the normal of the plane, no matter how our sphere has intersected with it – e.g., $N = (x, y, z)$



Sphere-Plane Collision

- The contact normal will always be the normal of the plane, no matter how our sphere has intersected with it – e.g.,

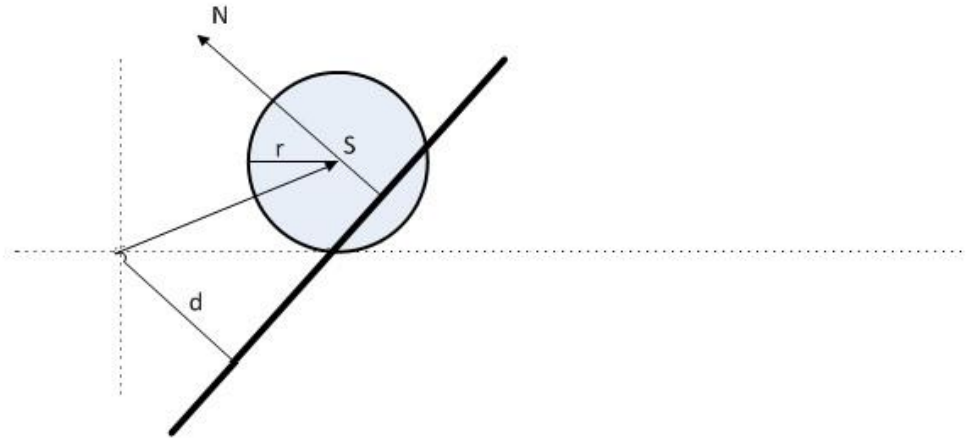
$$N = (x, y, z)$$



Sphere-Plane Collision

- D is the distance of the plane from the origin, ergo an interface has occurred if the sphere has position S such that

$$N \cdot S - D < r$$



Sphere-Plane Collision

- Remembering that $N = (x, y, z)$, remaining collision data is computed:

$$p = r - (N \cdot S - d)$$

$$P = S - N(r - p)$$

The Simple 1D Problem

('Simple' being a relative term)

A note on Partial Differentiation

- You'll have noticed that some of the hand-outs which discuss this work have used a ∂ notation (i.e., $\frac{\partial C}{\partial x}$)
- This notation indicates a 'partial derivative'
- Sure, I'm partial to derivatives, too – if I weren't, I'd not be doing this job - but what does it actually mean?

A note on Partial Differentiation

- Imagine a function of multiple variables – fluid motion is a good example.
- You have to consider density of the fluid, flow velocity, pressure, etc. – all of these are interconnected in terms of how the fluid moves, and changing one impacts all

- The problem here is identifying how changing just one variable affects the collective system.
- Remembering that we're considering an instant in time when we take a gradiant (differentiate), partial differentiation is basically extending the idea that "if we change one variable, the other variables can be treat as constant for that instant.
- They won't have had 'time' to change

A note on Partial Differentiation

- This is a slight oversimplification, but it gets the premise across – where we see ∂C divided by $\partial Whatever$, we mean we're only changing $Whatever$ and every other element of C isn't changing

- An arbitrary example would be some function f, where $f(x, y, z)$
- In this scenario $\frac{\partial f}{\partial x}$ is the change in f calculated assuming that y and z don't change

A note on Partial Differentiation

- Another notation we might come across from time to time, especially if we're interested in fluid dynamics, is $\nabla Whatever$.

- This is 'del' notation, represented normally using the nabla (∇) symbol, and represents the vector differential operator

- In real terms, all this means is the collective partial differentials of every variable in a system.

A note on Partial Differentiation

- So, in Cartesian space, ∇ can be written

$$\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right)$$

- Again, we're simplifying slightly, but not by that much.

- The important thing to take away from this is that if we want to program physics simulations, we'll be faced from time to time with maths which is new to us.
- That maths is very rarely beyond our ability to comprehend, so we shouldn't be disheartened encountering it.

So, the 1D Problem

- Consider a simple constraint, restricted to a single axis.
- We have variables x and y, which are positions (values, really, in 1D – e.g., 2 and 7 along a number line)
- We define our constraint C as requiring that the distance between x and y shall always be L

$$C(x, y) = \frac{1}{2}\left((x - y)^2 - L^2\right)$$

So, the 1D Problem

$$C(x,y) = \frac{1}{2}\left((x-y)^2 - L^2\right)$$

- We differentiate C with respect to time in order to determine how it's changing (just as in this morning's notes). This requires us to use that partial differentiation notation we just discussed, because C is a function of both x and y.
- We use something called the chain rule (we won't go into the whys and wherefores of this, but it's results should look fairly intuitive – see Tutorial 3):

$$\frac{dC}{dt} = \frac{\partial C}{\partial x}\frac{dx}{dt} + \frac{\partial C}{\partial y}\frac{dy}{dt}$$

So, the 1D Problem

$$\frac{dC}{dt} = \frac{\partial C}{\partial x}\frac{dx}{dt} + \frac{\partial C}{\partial y}\frac{dy}{dt}$$

- In dot notation, $\frac{dx}{dt}$ and $\frac{dy}{dt}$ are referred to as $\dot{x}$ and $\dot{y}$, respectively
- In real terms, they represent the change in time of a point – which, we recall from this morning, makes them elements of our Vector $\mathbf{V}$
- From our original function, we can determine that

$$\frac{\partial C}{\partial x} = x - y$$

$$\frac{\partial C}{\partial y} = (-1)(x - y)$$

So, the 1D Problem

$$\frac{\partial C}{\partial x} = x - y$$

$$\frac{\partial C}{\partial y} = (-1)(x - y) = (y - x)$$

- By simple rearrangement, this gives us

$$\dot{C} = (x - y)\dot{x} + (-1)(x - y)\dot{y}$$

- Going back to $\dot{C} = \mathbf{JV}$, bearing in mind that $\mathbf{V} = \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix}$, we can set our Jacobian to have the form

$$\mathbf{J} = \begin{pmatrix} x - y & y - x \end{pmatrix}$$

So, the 1D Problem

$$\mathbf{J} = (x - y \quad y - x)$$

- From this morning, we know that
$$\mathbf{F} = \mathbf{J}^T \lambda$$
- Where $\mathbf{F}$ is the Constraint Force, $\mathbf{J}^T$ is the **transpose** of the Jacobian, and λ is our change. Transposing our Jacobian just means reflecting it along the diagonal:

$$\mathbf{J} = (x - y \quad y - x) \text{ ergo } \mathbf{J}^T = \begin{pmatrix} x - y \\ y - x \end{pmatrix}$$

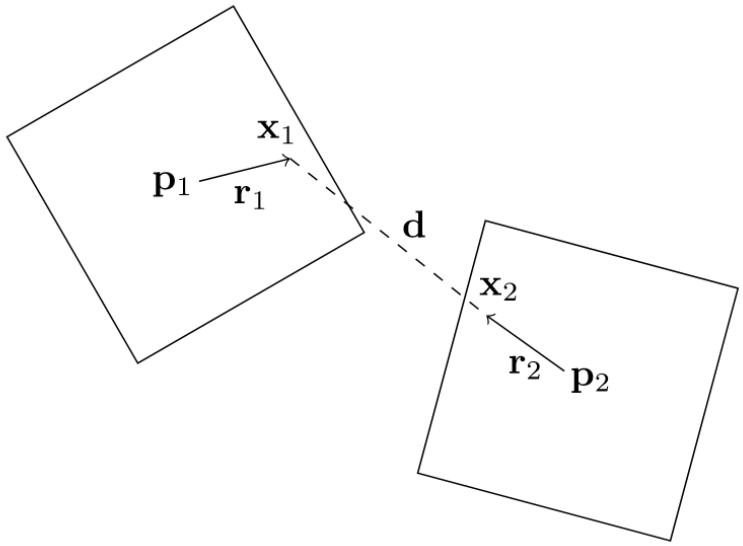
So, the 1D Problem

Thus $\mathbf{F} = \begin{pmatrix} x - y \\ y - x \end{pmatrix} \lambda$ for some value λ

- Note that the force is proportional to the distance between the points x and y, but equal and opposite for each object (Newton's Third Law)

Expanding this to 3D

- Consider this example. Two boxes are connected by a distance constraint at $\mathbf{x_1}$ and $\mathbf{x_2}$, where these are three-element vectors denoting positions



Expanding this to 3D

- In-keeping with our 1D example, we define this constraint as

$$\mathrm{C}(\mathbf{x_1}, \mathbf{q_1}, \mathbf{x_2}, \mathbf{q_2}) = \frac{1}{2}\left((\mathbf{x_2} - \mathbf{x_1})^2 - L^2\right)$$

where $\mathbf{x}$ denotes a position and $\mathbf{q}$ an orientation

- Differentiating $\mathbf{x_1}$ and $\mathbf{x_2}$ is a complicated prospect at first sight, factoring in rotation, so we redefine them as vectors from their centre of rotation, e.g. $\mathbf{x_1} = \mathbf{p_1} + \mathbf{r_1}$
- From this, we differentiate with respect to time, getting a velocity, and the product of an angular velocity with our vector from centre of rotation:

$$\frac{d\mathbf{x_1}}{dt} = \mathbf{v_1} + \omega_1 \times \mathbf{r_1} \quad \frac{d\mathbf{x_2}}{dt} = \mathbf{v_2} + \omega_2 \times \mathbf{r_2}$$

Expanding this to 3D

- We recall from the 1D example that

$$\frac{dC}{dt} = \frac{\partial C}{\partial x}\frac{dx}{dt} + \frac{\partial C}{\partial y}\frac{dy}{dt} \rightarrow \frac{dC}{dt} = \frac{\partial C}{\partial \mathbf{x}_1}\frac{d\mathbf{x}_1}{dt} + \frac{\partial C}{\partial \mathbf{x}_2}\frac{d\mathbf{x}_2}{dt}$$

- Subbing in the results from earlier, we're left with

$$\dot{C} = (\mathbf{x}_2 - \mathbf{x}_1) \cdot (\mathbf{v}_2 + \omega_2 \times \mathbf{r}_2 - \mathbf{v}_1 + \omega_1 \times \mathbf{r}_1)$$

- Which can be simplified down (since $\mathbf{x}_2 - \mathbf{x}_1 = \mathbf{d}$, as per the diagram) and the vector identity in the handout giving:

$$\dot{C} = -\mathbf{d} \cdot \mathbf{v}_1 - (\mathbf{r}_1 \times \mathbf{d}) \cdot \omega_1 + \mathbf{d} \cdot \mathbf{v}_2 + (\mathbf{r}_2 \times \mathbf{d}) \cdot \omega_2$$

Expanding this to 3D

$$\dot{C} = -\mathbf{d} \cdot \mathbf{v}_1 - (\mathbf{r}_1 \times \mathbf{d}) \cdot \omega_1 + \mathbf{d} \cdot \mathbf{v}_2 + (\mathbf{r}_2 \times \mathbf{d}) \cdot \omega_2$$

- So, this looks a little meaningless until we remember that our Jacobian is **meant** to be coefficients of our linear and angular velocities – one of each, for each constrained object. From Tutorial 3's handout,

$$\dot{C} = \mathbf{j}_1 \cdot \mathbf{v}_1 + \mathbf{j}_2 \cdot \omega_1 + \mathbf{j}_3 \cdot \mathbf{v}_2 + \mathbf{j}_4 \cdot \omega_2$$

- Transposing out, and substituting the $\dot{C} = \mathbf{JV}$ form again, we wind up with

$$\mathbf{J} = \begin{pmatrix} -\mathbf{d}^T & -(\mathbf{r}_1 \times \mathbf{d})^T & \mathbf{d}^T & (\mathbf{r}_2 \times \mathbf{d})^T \end{pmatrix}$$

Adding Energy and Corrections/Damping

- So far, focus has been on constraints which add no energy to the system, or

$$\dot{C} = 0$$

- We can easily envision a scenario in our game where we want a constraint to add energy to the system.

- A drive train, for example, which introduces energy to the environment without us having to do all of the energetic and mechanical computations involved in the internal combustion engine.

Adding Energy and Corrections/Damping

- In such situations, we simply define a vector function, e.g., ζ, and set $\dot{C} = \zeta$
- This is known as a bias vector, and can relate to position, angle, and time.
- Intuitively, we can also introduce a bias vector to take energy out of the system.
- This is one way of dealing with the errors inherent to our iterative update process

Practical Computation of Velocity Update

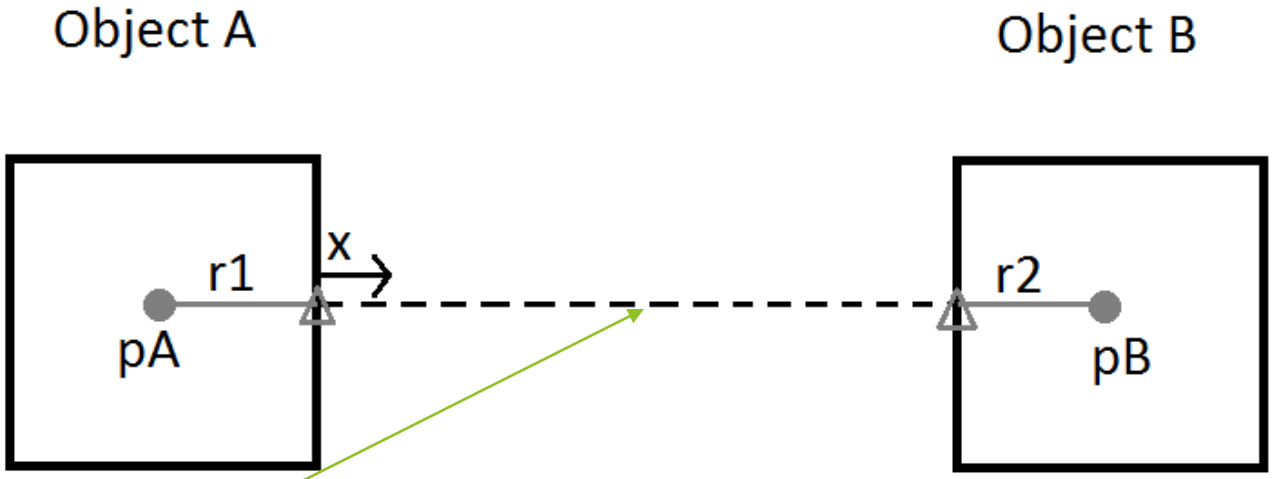
Overview

- For the remainder of this lecture, we'll explore what all of this maths actually means for our software
- We'll consider an example constraint scenario, and step through the elements of code which resolve it
- The important thing to remember is that the purpose of the distance constraint is to ensure that both entities about the constraint maintain relative distance; as such, we are ensuring that the velocities about the direction of the constraint remain identical.

The Scenario

- Objects A and B are connected by a distance constraint between the grey triangles
- pA and pB represent respective centres of mass
- r1 and r2 are vectors connecting those CoMs to respective triangles
- x is the unit vector in the direction of the constraint

The Jacobian




```
Vector3 ab = globalOnB - globalOnA;
Vector3 abn = ab;
abn.Normalise();

Vector3 r1 = (globalOnA - objA->GetPosition());
Vector3 r2 = (globalOnB - objB->GetPosition());

this->j1 = -abn;
this->j2 = Vector3::Cross(-r1, abn);
this->j3 = abn;
this->j4 = Vector3::Cross(r2, abn);
this->b = 0.0f;

this->distance = ab.Length();
```

The Jacobian


```
Vector3 ab = globalOnB - globalOnA;
Vector3 abn = ab;
abn.Normalise();

Vector3 r1 = (globalOnA - objA->GetPosition());
Vector3 r2 = (globalOnB - objB->GetPosition());

this->j1 = -abn;
this->j2 = Vector3::Cross(-r1, abn);
this->j3 = abn;
this->j4 = Vector3::Cross(r2, abn);
this->b = 0.0f;

this->distance = ab.Length();
```

The Jacobian


```
Vector3 ab = globalOnB - globalOnA;
Vector3 abn = ab;
abn.Normalise();

Vector3 r1 = (globalOnA - objA->GetPosition());
Vector3 r2 = (globalOnB - objB->GetPosition());

this->j1 = -abn;
this->j2 = Vector3::Cross(-r1, abn);
this->j3 = abn;
this->j4 = Vector3::Cross(r2, abn);
this->b = 0.0f;

this->distance = ab.Length();
```

The Jacobian


```
Vector3 ab = globalOnB - globalOnA;
Vector3 abn = ab;
abn.Normalise();

Vector3 r1 = (globalOnA - objA->GetPosition());
Vector3 r2 = (globalOnB - objB->GetPosition());

this->j1 = -abn;
this->j2 = Vector3::Cross(-r1, abn);
this->j3 = abn;
this->j4 = Vector3::Cross(r2, abn);
this->b = 0.0f;

this->distance = ab.Length();
```

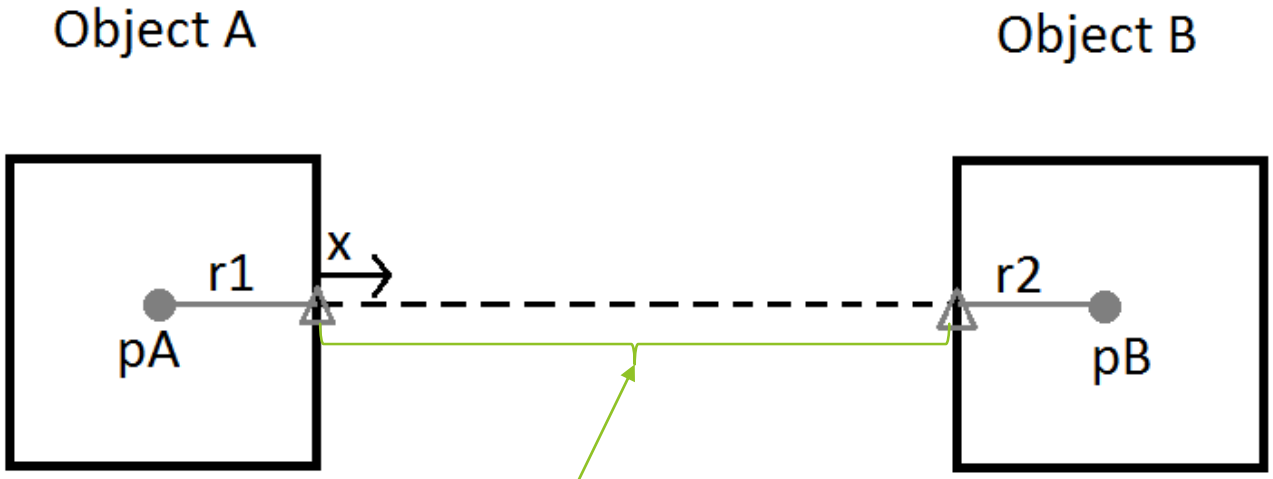
$$\mathbf{J} = \begin{pmatrix} \mathbf{v}_1 \\ \omega_1 \\ \mathbf{v}_2 \\ \omega_2 \end{pmatrix}$$

The Jacobian


```
Vector3 ab = globalOnB - globalOnA;
Vector3 abn = ab;
abn.Normalise();

Vector3 r1 = (globalOnA - objA->GetPosition());
Vector3 r2 = (globalOnB - objB->GetPosition());

this->j1 = -abn;
this->j2 = Vector3::Cross(-r1, abn);
this->j3 = abn;
this->j4 = Vector3::Cross(r2, abn);
this->b = 0.0f;

this->distance = ab.Length();
```

$$\dot{C} = \mathbf{JV}$$
$$= -\beta C$$

The Jacobian




```
Vector3 ab = globalOnB - globalOnA;
Vector3 abn = ab;
abn.Normalise();

Vector3 r1 = (globalOnA - objA->GetPosition());
Vector3 r2 = (globalOnB - objB->GetPosition());

this->j1 = -abn;
this->j2 = Vector3::Cross(-r1, abn);
this->j3 = abn;
this->j4 = Vector3::Cross(r2, abn);
this->b = 0.0f;

this->distance = ab.Length();
```

The Jacobian

- Now we have j1, j2, j3 and j4, everything we do in terms of operations will refer to these elements – it's the entire point of computing the Jacobian in the first place

Constraint Mass

- We define this as a float (scalar), and compute it as follows:

```
float constraint_mass = objA->GetInverseMass() * Vector3::Dot(j1, j1)
    + Vector3::Dot(j2, (objA->GetInverseInertia() * j2))
    + objB->GetInverseMass() * Vector3::Dot(j3, j3)
    + Vector3::Dot(j4, (objB->GetInverseInertia() * j4));
```

- We leverage inverse mass and inverse inertia of each object in turn.
- Notice that we're basically gathering magnitudes, either multiplying by dotted vectors, or dotting vectors against the vector product of matrix multiplication
- You can see how these map to the properties obtained previously – inverse mass of Object A to linear Jacobian component for Object A, and so forth

Differentiated Constraint ($\dot{C}$)

$$\dot{C} = \mathbf{JV}$$

```
float jv = Vector3::Dot(j1, objA->GetLinearVelocity())
+ Vector3::Dot(j2, objA->GetAngularVelocity())
+ Vector3::Dot(j3, objB->GetLinearVelocity())
+ Vector3::Dot(j4, objB->GetAngularVelocity());
```

- This one really is what it says on the tin…

Compute Lambda

- We notionally have a β, but it's set to 0.0f
- Lambda is the negative of cee-dot, over the constraint mass (so, basically, the last two steps divided)

```
float denom = -(jv + b);
float lambda = denom / constraint_mass;
```

Update Velocity

- We recall that

$$\mathbf{F} = \mathbf{J}^T \lambda$$

```
objA->SetLinearVelocity(objA->GetLinearVelocity()
    + (j1 * lambda) * objA->GetInverseMass());
objA->SetAngularVelocity(objA->GetAngularVelocity()
    + objA->GetInverseInertia() * (j2 * lambda));
objB->SetLinearVelocity(objB->GetLinearVelocity()
    + (j3 * lambda) * objB->GetInverseMass());
objB->SetAngularVelocity(objB->GetAngularVelocity()
    + objB->GetInverseInertia() * (j4 * lambda));
```

$$v_{n+1} = v_n + a_n \Delta t \quad a = \frac{F}{m}$$

- At this point, we're functionally integrating to obtain the new velocity components based on our constraint

Update Velocity

- We recall that

$$\mathbf{F} = \mathbf{J}^T \lambda$$

```
objA->SetLinearVelocity(objA->GetLinearVelocity()
    + (j1 * lambda) * objA->GetInverseMass());
objA->SetAngularVelocity(objA->GetAngularVelocity()
    + objA->GetInverseInertia() * (j2 * lambda));
objB->SetLinearVelocity(objB->GetLinearVelocity()
    + (j3 * lambda) * objB->GetInverseMass());
objB->SetAngularVelocity(objB->GetAngularVelocity()
    + objB->GetInverseInertia() * (j4 * lambda));
```

$$v_{n+1} = v_n + a_n \Delta t \quad a = \frac{F}{m}$$

- At this point, we're functionally integrating to obtain the new velocity components based on our constraint

Update Velocity

- We recall that

$$\mathbf{F} = \mathbf{J}^T \lambda$$

```
objA->SetLinearVelocity(objA->GetLinearVelocity()
   + (j1 * lambda) * objA->GetInverseMass());
objA->SetAngularVelocity(objA->GetAngularVelocity()
   + objA->GetInverseInertia() * (j2 * lambda));
objB->SetLinearVelocity(objB->GetLinearVelocity()
   + (j3 * lambda) * objB->GetInverseMass());
objB->SetAngularVelocity(objB->GetAngularVelocity()
   + objB->GetInverseInertia() * (j4 * lambda));
```

$$v_{n+1} = v_n + a_n \Delta t \qquad a = \frac{F}{m}$$

- At this point, we're functionally integrating to obtain the new velocity components based on our constraint

Within the Framework

- This simplifies down… Check out the code to see how.

Summary

- Reviewed constraints in 1D and extended that into 3D
- Discussed introducing energy to our system
- Stepped through the implementation of a solver
- Touched on Global Solvers