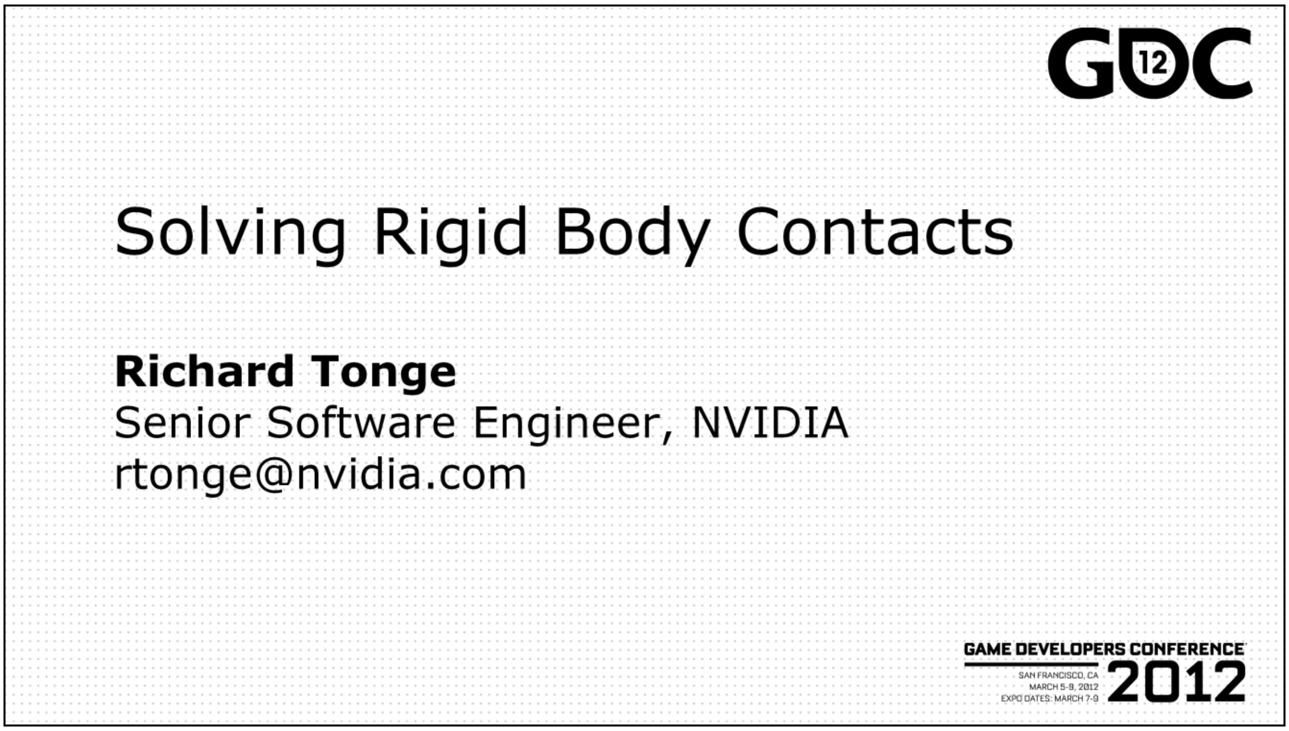


GDC 12

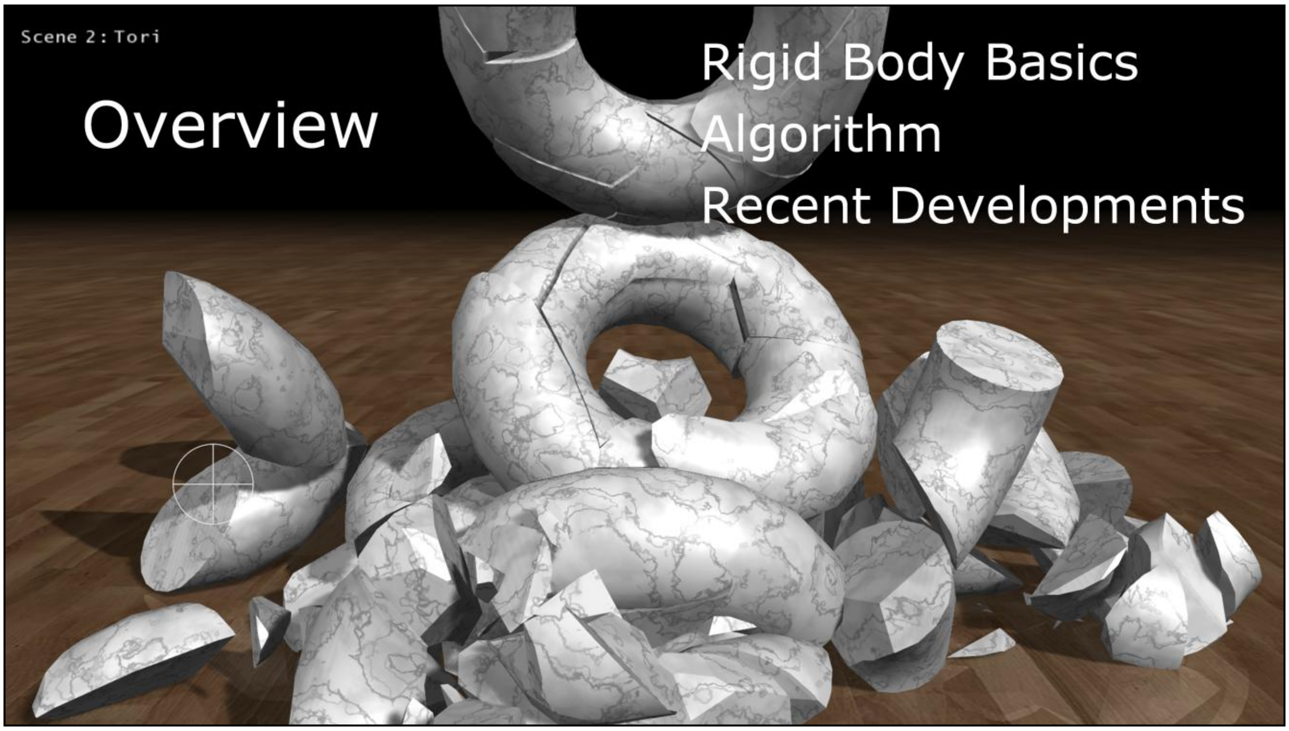
Solving Rigid Body Contacts

Richard Tonge
Senior Software Engineer, NVIDIA
rtonge@nvidia.com

GAME DEVELOPERS CONFERENCE
SAN FRANCISCO, CA
MARCH 5-9, 2012
EXPO DATES: MARCH 7-9
2012

Hello, my name is Richard Tonge and today I'm going to talk about rigid body dynamics for games.

Gino just talked about how to detect if collision has occurred, and I'm going to talk about what to do to stop solid things going through each other.



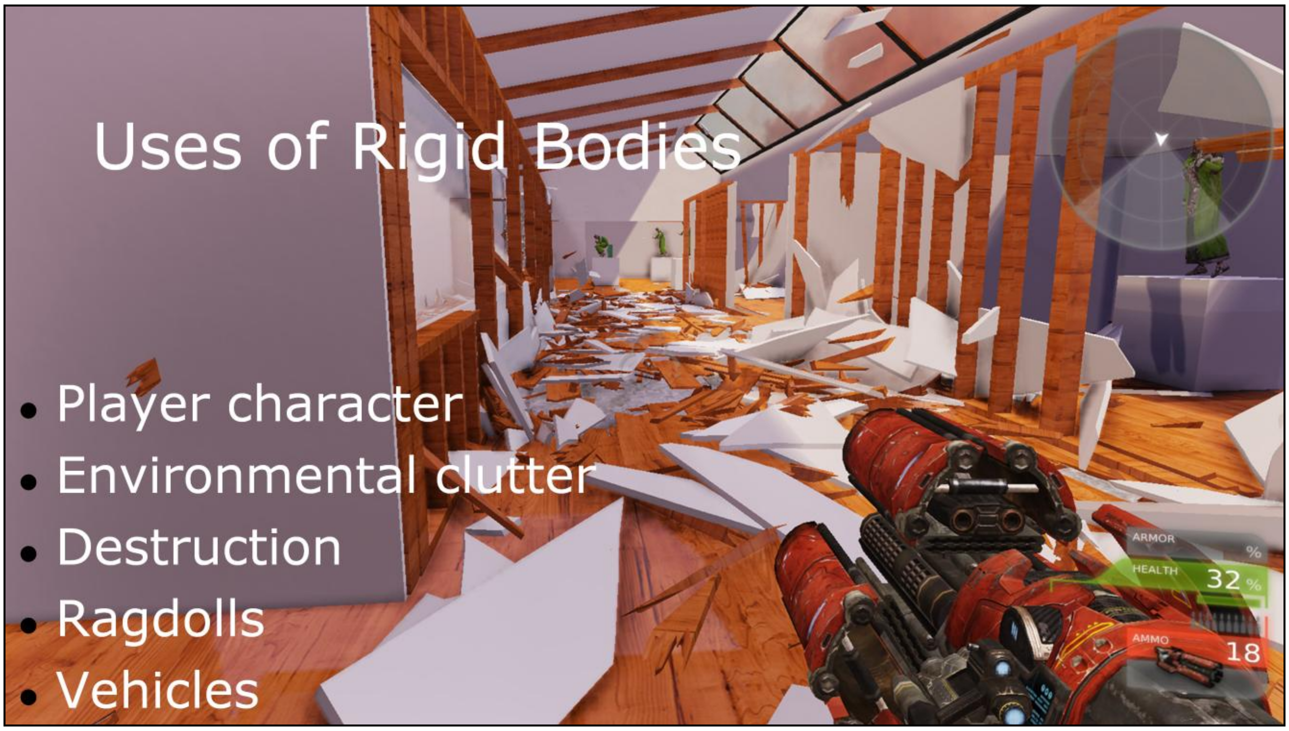
Overview

- Rigid Body Basics
- Algorithm
- Recent Developments

Scene 2: Tori

In previous years, the audience of this tutorial has included a range of people, from those who are learning games physics for the first time to people who have written complete physics engines.

There is a common rigid body solver algorithm that’s been described many times before at GDC. When I was writing the presentation, I tried to include enough material for beginners to make the description of this algorithm accessible, the algorithm itself, and also talk about some recent developments for people who have seen the algorithm before.



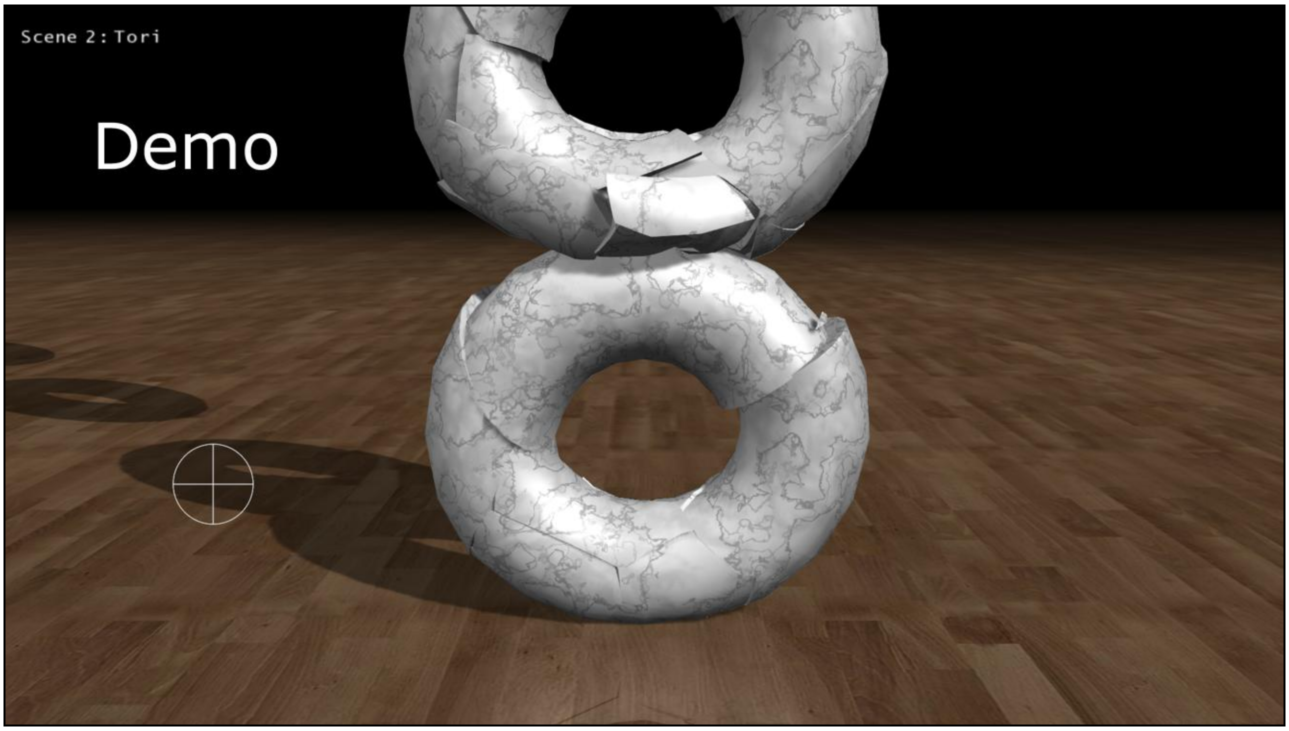
Uses of Rigid Bodies

- Player character
- Environmental clutter
- Destruction
- Ragdolls
- Vehicles

Humans like blowing stuff up

The background of this slide is our art gallery demo that is fully destructible using rigid body simulation. The entire simulation is running on the GPU and this allows us to simulate thousands of debris fragments with a wide range of sizes.

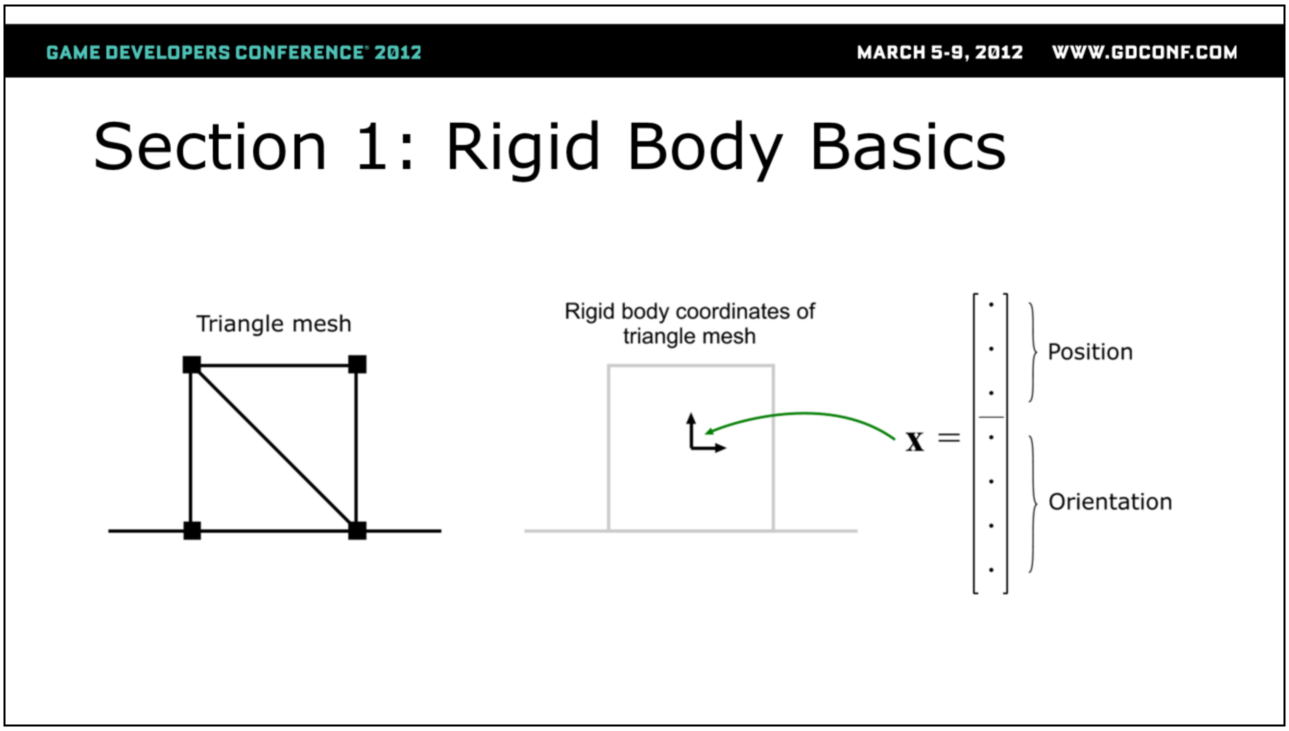
We showed it at GDC last year and we are showing an improved version this year.



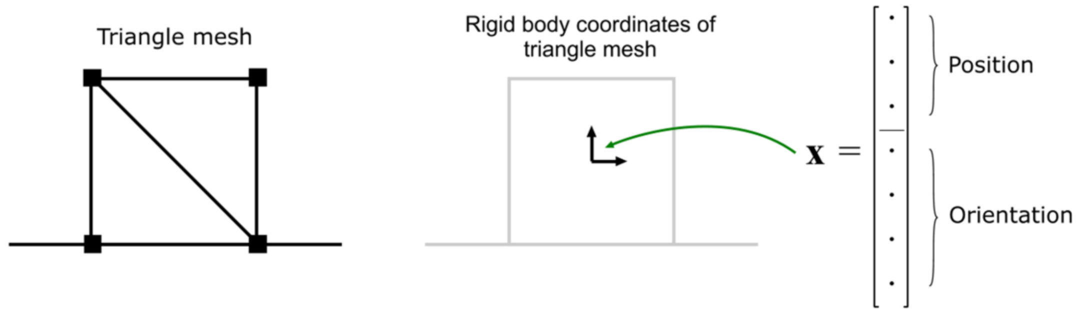
Demo

For the previous demo, the artist had to author how they wanted the floor, walls, windows etc to crack. This is our new demo which calculates the fracture automatically at runtime. This means that you can put destruction into games without generating too much extra work for the artists.

Again the simulation is running entirely on the GPU (except for the part which splits the mesh, which is on the CPU currently). As the fracture is calculated algorithmically, there is in principle no limit to the number of times you can split the pieces, although you will eventually run out of memory. It shows very small destructed pieces interacting stably with large undestructed pieces, showing that you can run simulations with a wide range of object sizes entirely on the GPU.


GAME DEVELOPERS CONFERENCE® 2012
MARCH 5-9, 2012 WWW.GDCONF.COM

Section 1: Rigid Body Basics



Definition of rigid body coordinates

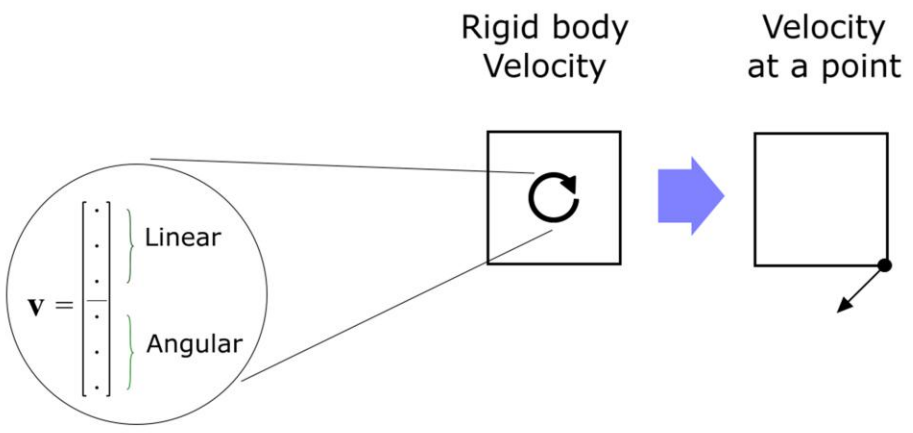
In graphics APIs like OpenGL and DirectX, it's easy to animate rigid objects. Why? It's because we can specify mesh vertices relative to a local coordinate frame. So when we render, we don't have to specify the world coordinate of each vertex each frame, we just change the transformation matrix to move the mesh in the scene.

Ok, so let's talk about using rigid body physics to move the mesh around the scene. So the first concept I'd like to introduce is the center of mass. In graphics, it doesn't usually matter where the artist places the origin of the mesh. In rigid body physics, the center of mass of a mesh has special significance, so to keep things simple, let's assume that the artist has placed the origin of the mesh at the center of mass. (If this isn't true, we can just store an offset). So a rigid body engine modifies the mesh's transformation matrix each frame to move the center of mass around the scene, and the rest of the mesh follows. Also, the physics engine can rotate the mesh around the center of mass by changing the orientation part of the transformation matrix.

The transformation matrix can be efficiently stored as a position and a quaternion, a 7D vector. We call this 7D vector "the pose of the mesh in rigid body coordinates". I'm going to use the letter x to represent the rigid body pose in this presentation.

GAME DEVELOPERS CONFERENCE® 2012 MARCH 5-9, 2012 WWW.GDCONF.COM

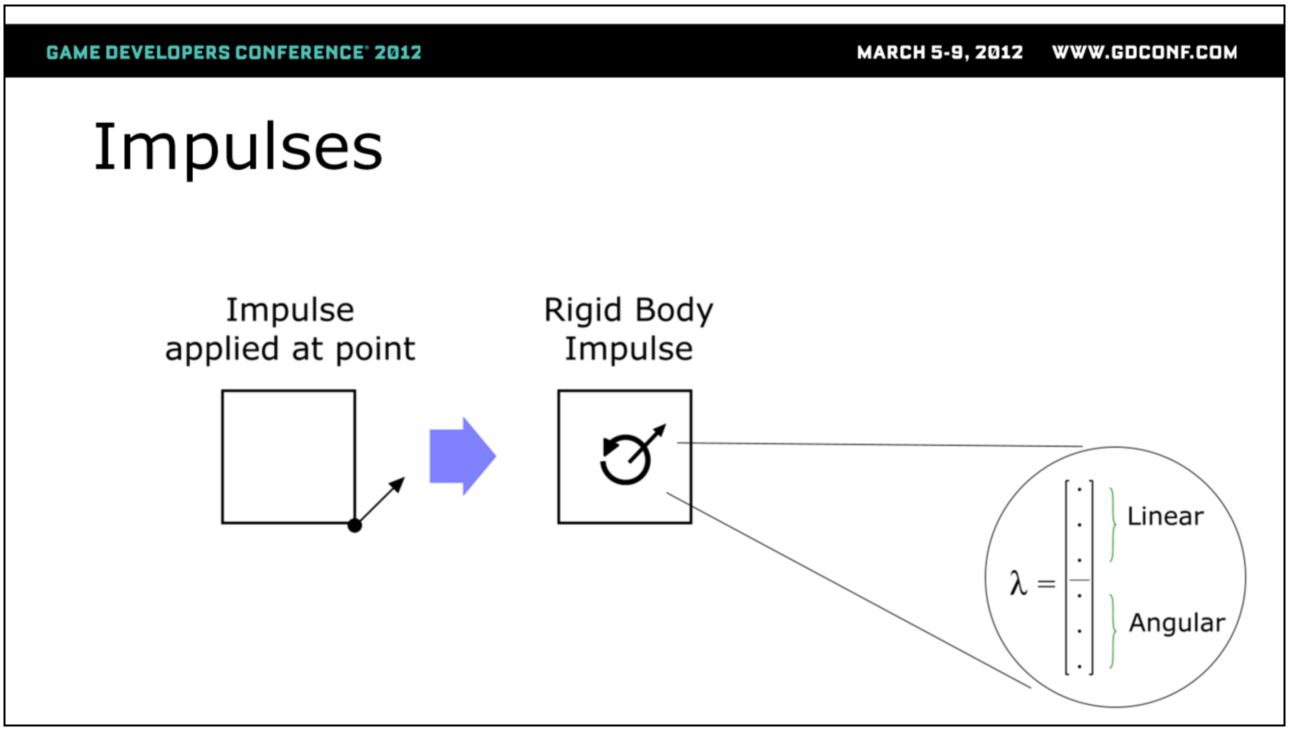
Velocities



Velocities and impulses in rigid body coordinates

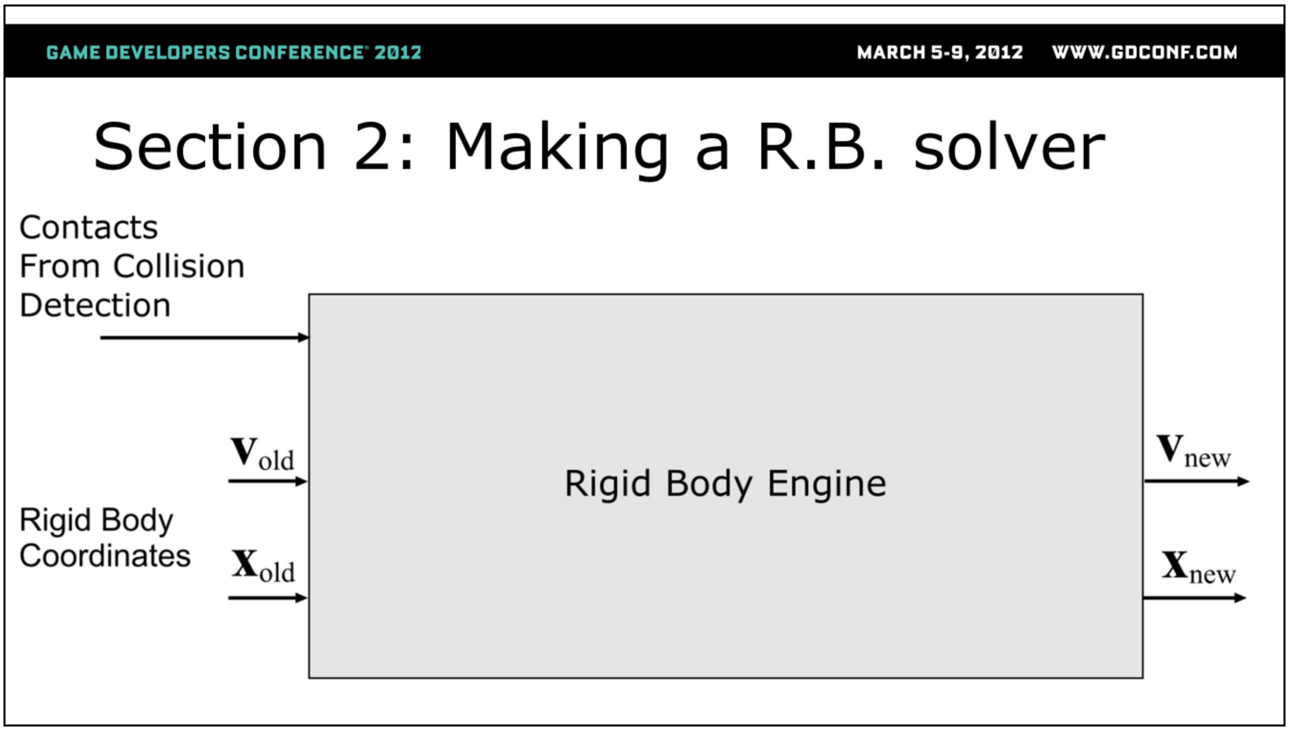
We can express other things in rigid body coordinates, like velocities and impulses. Just as the rigid body pose uniquely determines the position of every vertex of the body, the rigid body velocity (the linear and angular velocity of the center of mass) determines the velocity of every vertex (and also every other point) of the rigid body.

We'll show how to calculate this in a minute.



Impulses

Also, if we want to apply an impulse to a vertex (or other point on the rigid body), we can calculate the equivalent rigid body impulse and apply the impulse by changing the rigid body velocity.


GAME DEVELOPERS CONFERENCE® 2012
MARCH 5-9, 2012 WWW.GDCONF.COM

Section 2: Making a R.B. solver

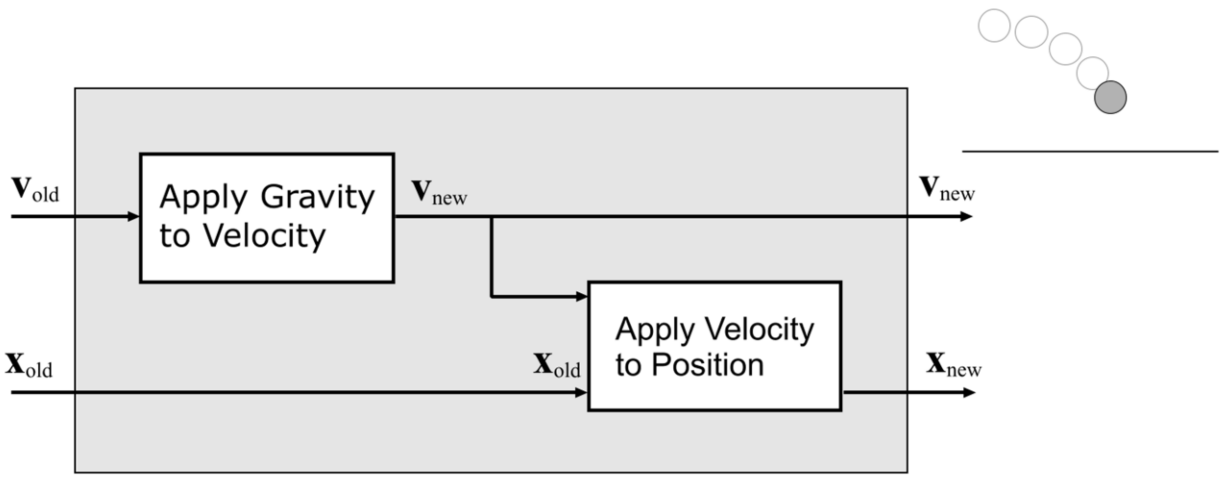
Contacts From Collision Detection
$\mathbf{V}_{\text{old}}$
Rigid Body Coordinates
$\mathbf{X}_{\text{old}}$
Rigid Body Engine
$\mathbf{V}_{\text{new}}$
$\mathbf{X}_{\text{new}}$

So a rigid body engine is just something that updates a pose and velocity in rigid body coordinates each frame, according to some contacts supplied by a collision detection engine.

This slide shows the highest level representation of a rigid body engine. Over the next few slides we'll make the diagram more detailed.

GAME DEVELOPERS CONFERENCE® 2012 MARCH 5-9, 2012 WWW.GDCONF.COM

Moving a Body Without Collisions

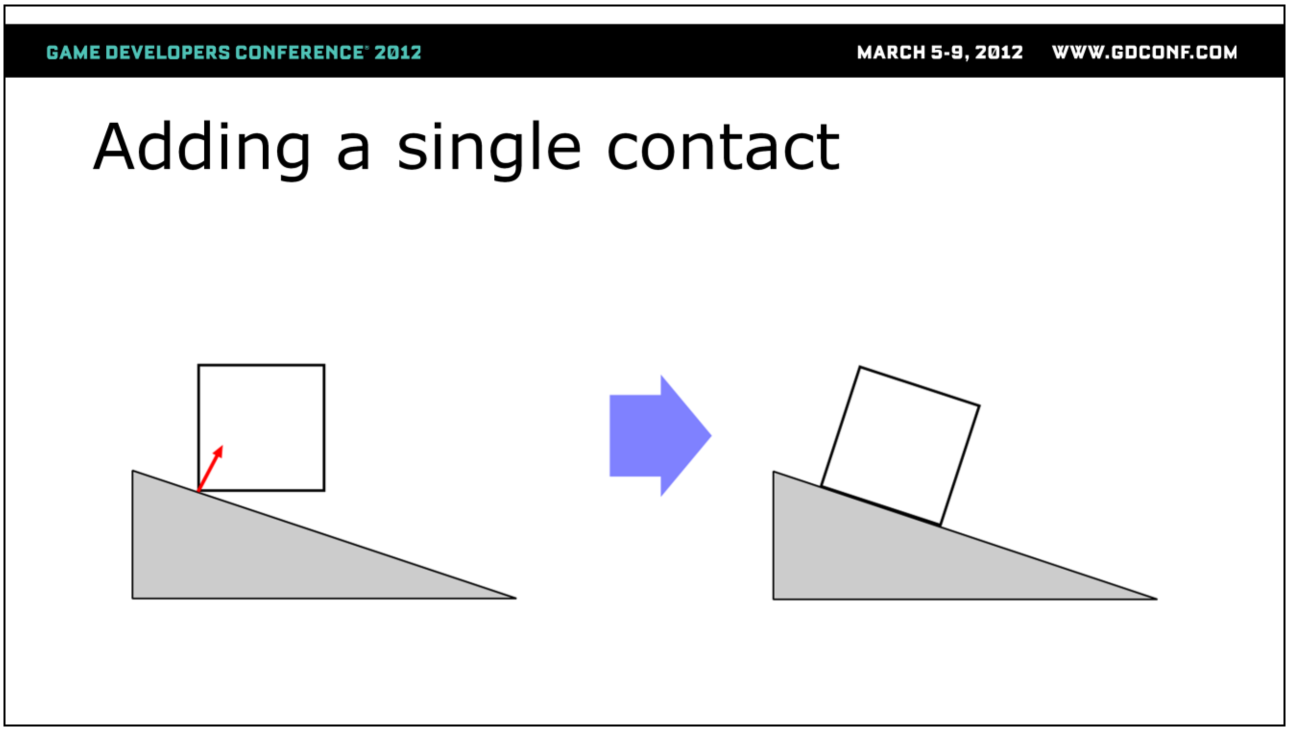

$\mathbf{v}_{\text{old}}$
Apply Gravity to Velocity
$\mathbf{v}_{\text{new}}$
$\mathbf{v}_{\text{new}}$
$\mathbf{x}_{\text{old}}$
$\mathbf{x}_{\text{old}}$
Apply Velocity to Position
$\mathbf{x}_{\text{new}}$

The simplest rigid body physics engine just moves a single body through the air without collisions.

You'll notice that there isn't any math in these boxes, we'll get to that in a moment.

This box shows how we transform the rigid body coordinates each frame.

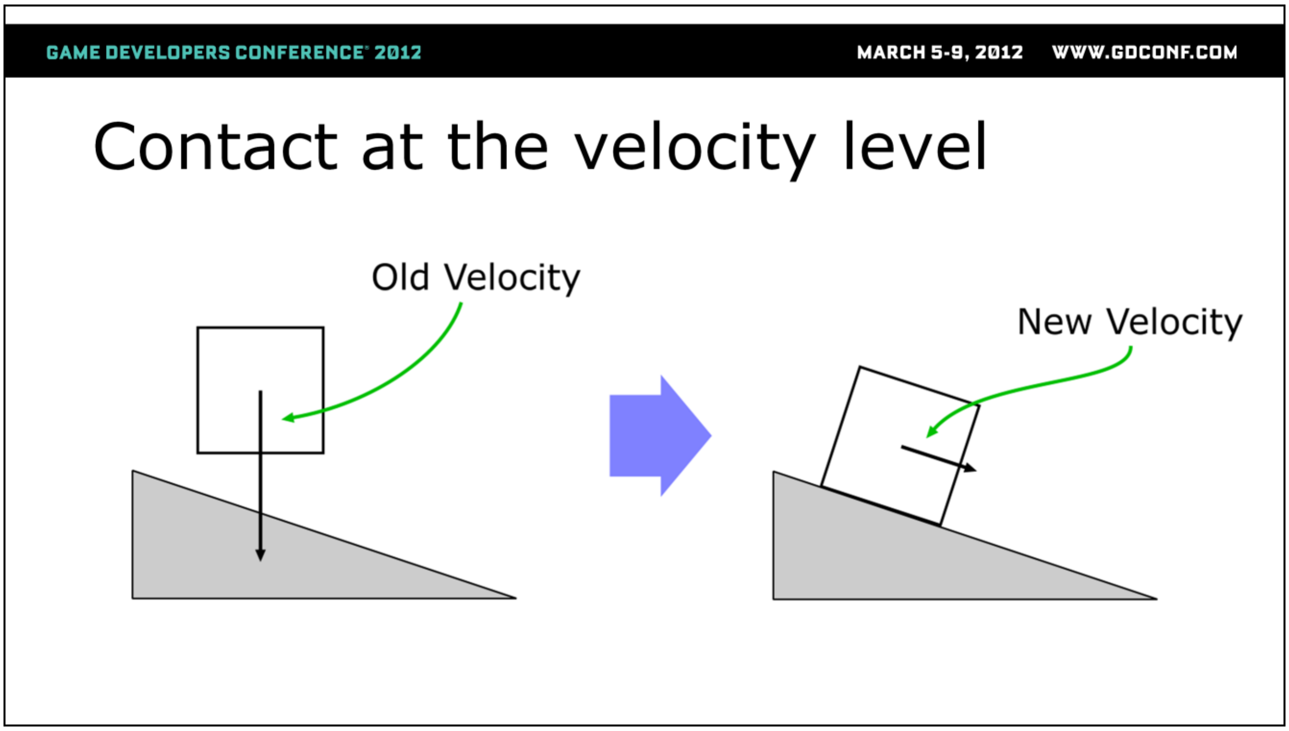
First we update the velocity by applying gravity to it. Then we use the new velocity to update the pose.


GAME DEVELOPERS CONFERENCE® 2012
MARCH 5-9, 2012 WWW.GDCONF.COM

Adding a single contact

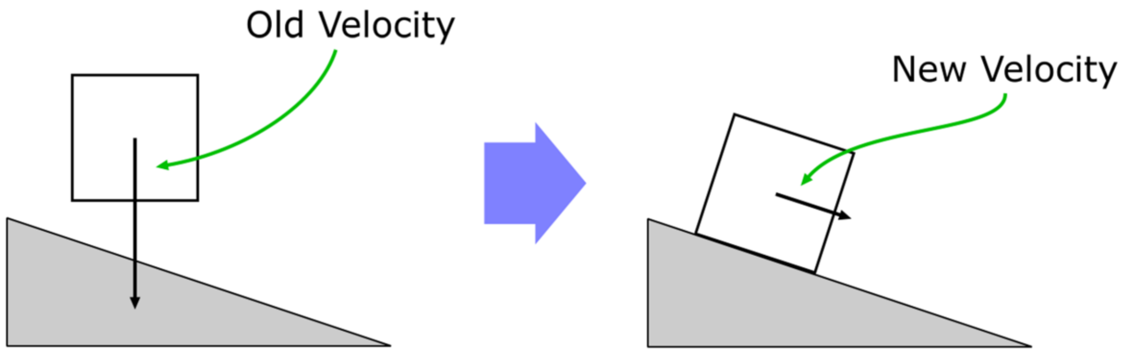
The next simplest situation we could consider is a body colliding with the ground at a single point of contact.

The contact here is shown in red, and the picture on the right shows what we want to happen. To keep things simple we're going to look at an inelastic contact, so we're imagining that the box and slope are so rigid that the box won't bounce when it hits the slope.



GAME DEVELOPERS CONFERENCE® 2012 MARCH 5-9, 2012 WWW.GDCONF.COM

Contact at the velocity level

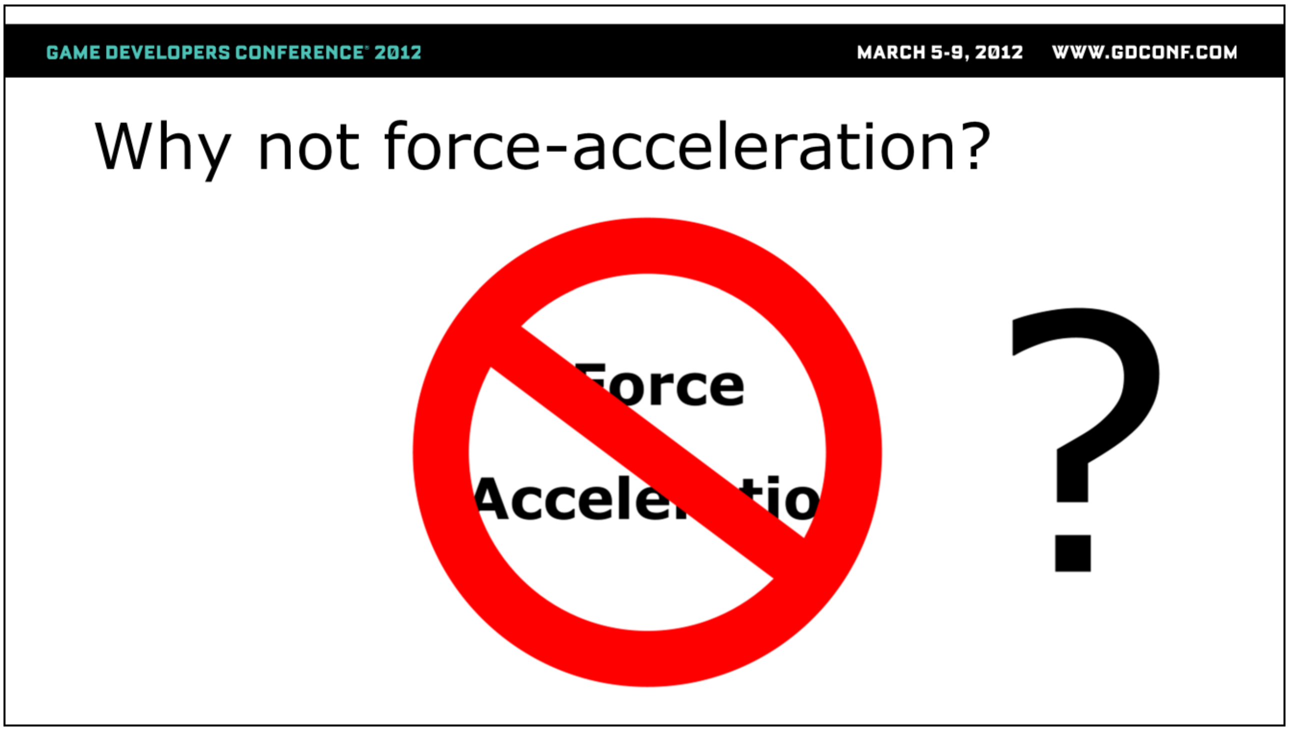


We're going to prevent bodies penetrating each other by applying impulses to change their velocities.

So when the box hits the slope, we apply an impulse to counteract the effect of gravity and make the velocity parallel with the slope. Making the velocity parallel to the slope will cause the body to slide down the slope in future frames.

Another way of saying this is that we are "projecting the unconstrained velocity onto the space of allowable velocities".

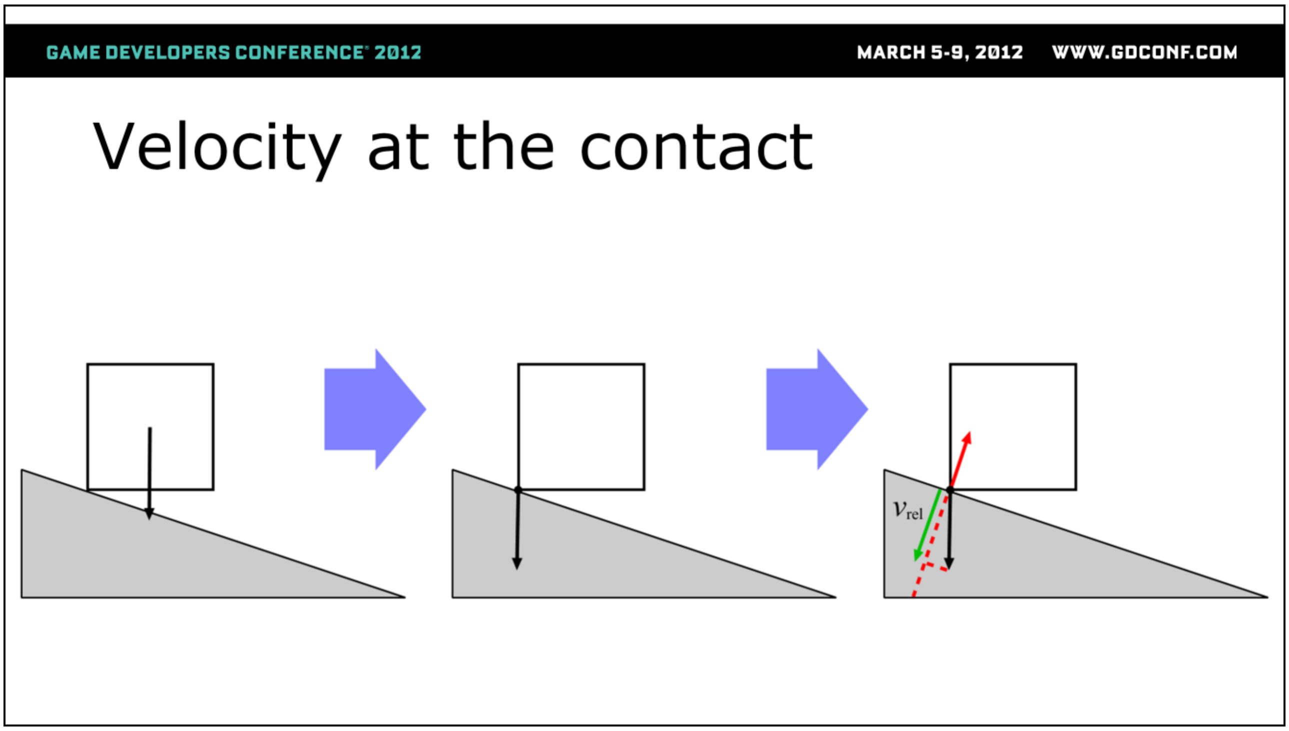
This is called solving the contact constraint at the velocity-impulse level. Collisions will also require positions and rotations to be changed slightly, but we'll get to that in a moment.


GAME DEVELOPERS CONFERENCE® 2012 MARCH 5-9, 2012 WWW.GDCONF.COM

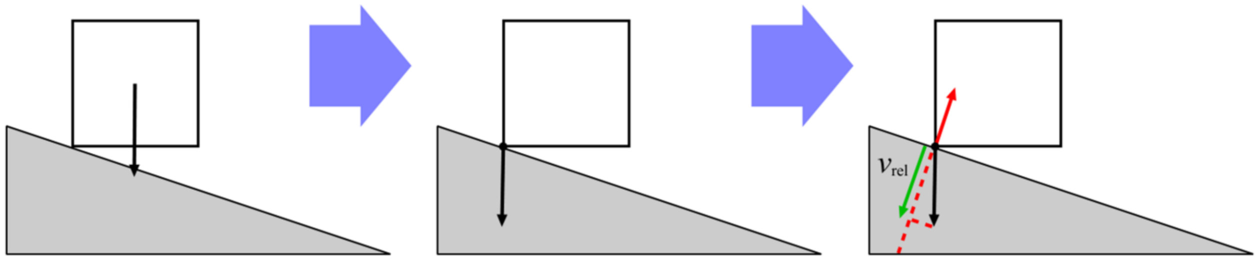
Why not force-acceleration?

Why don't we solve at the force-acceleration level?

Friction is much better behaved at the impulse-velocity level and it allows us to treat resting contact in the same way as colliding contact.

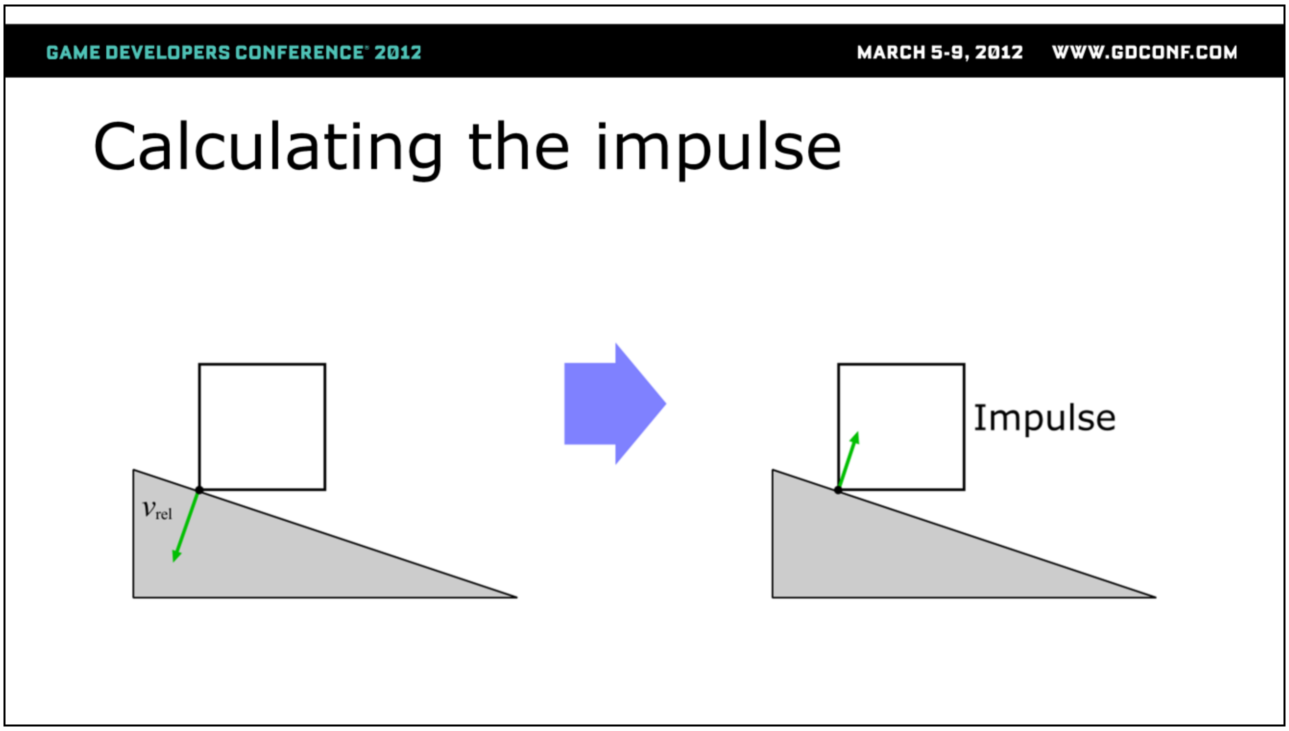


Velocity at the contact

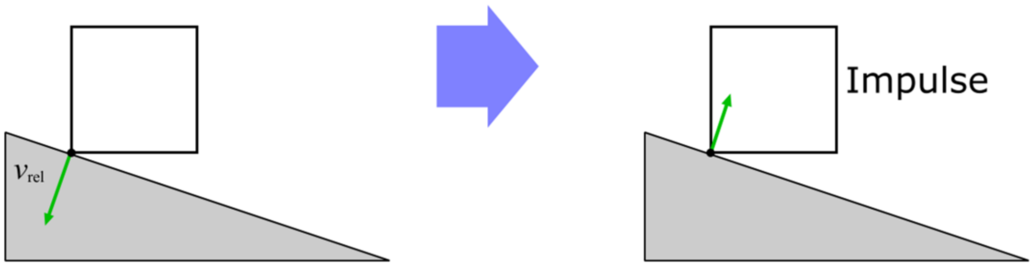


We want to find a new velocity that will cause the body to slide down the slope instead of into penetration. The first picture shows the unconstrained velocity due to gravity in rigid body coordinates. Recall from a few slides earlier that the velocity of the center of mass determines the velocity of every point on the rigid body. We show exactly how later. In this case, the velocity at the contact is the same as the center of mass because the body is not rotating.

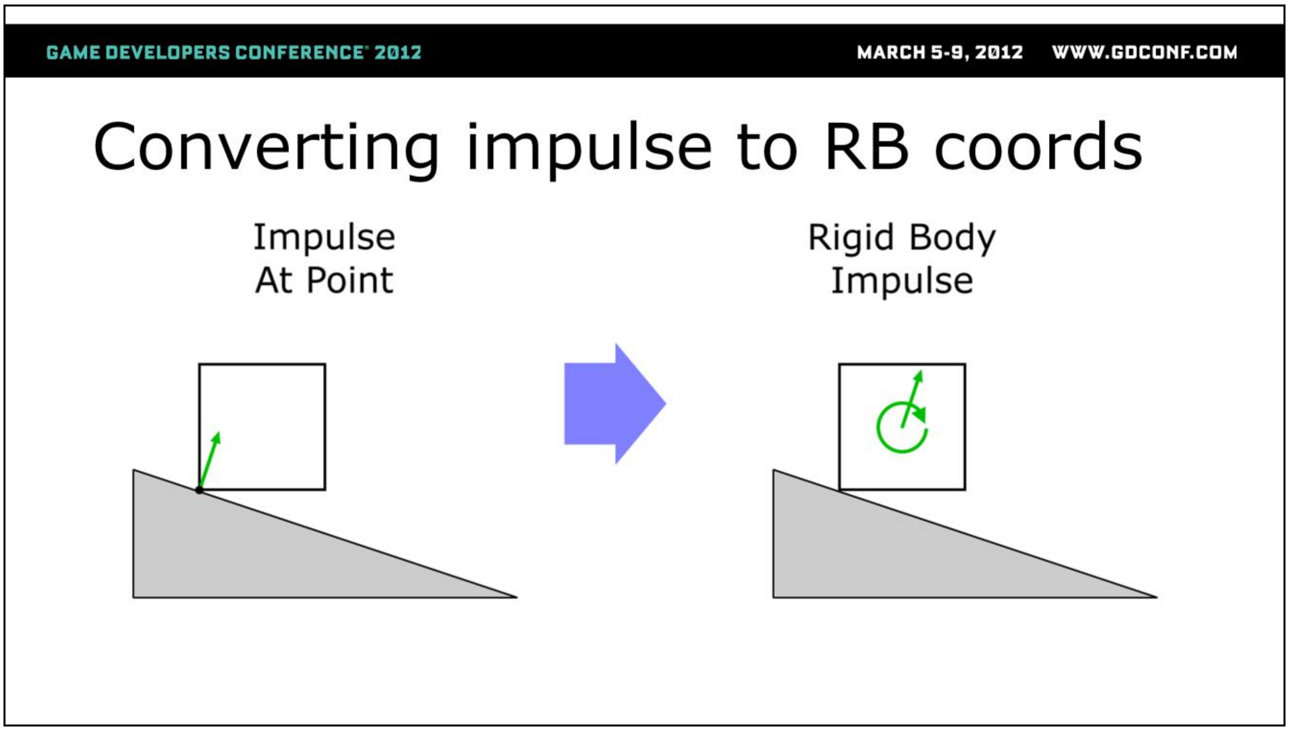
We want to eliminate the component of the velocity that is pulling the box into penetration, so first we need to the magnitude of the velocity component we want to eliminate.



Calculating the impulse

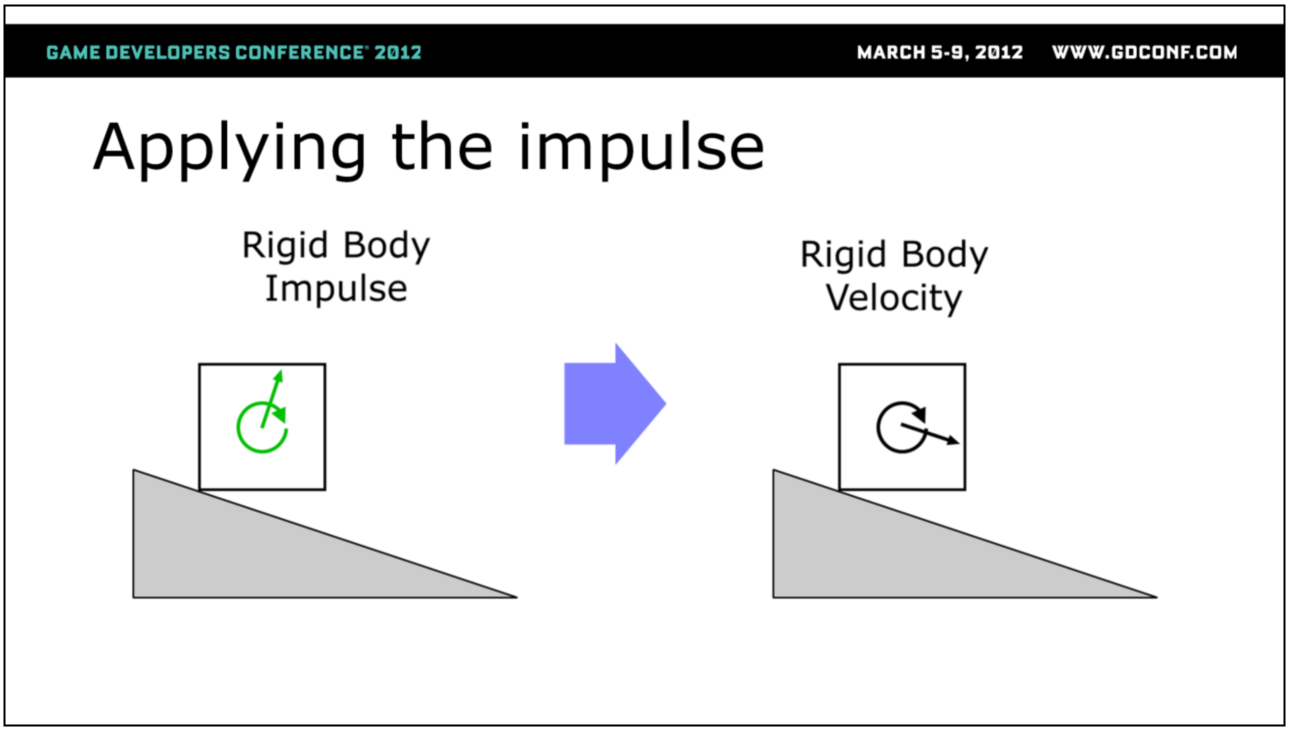


Once we know the direction and magnitude of the velocity component we want to eliminate, we can calculate the impulse required that will eliminate it.



Converting impulse to RB coords

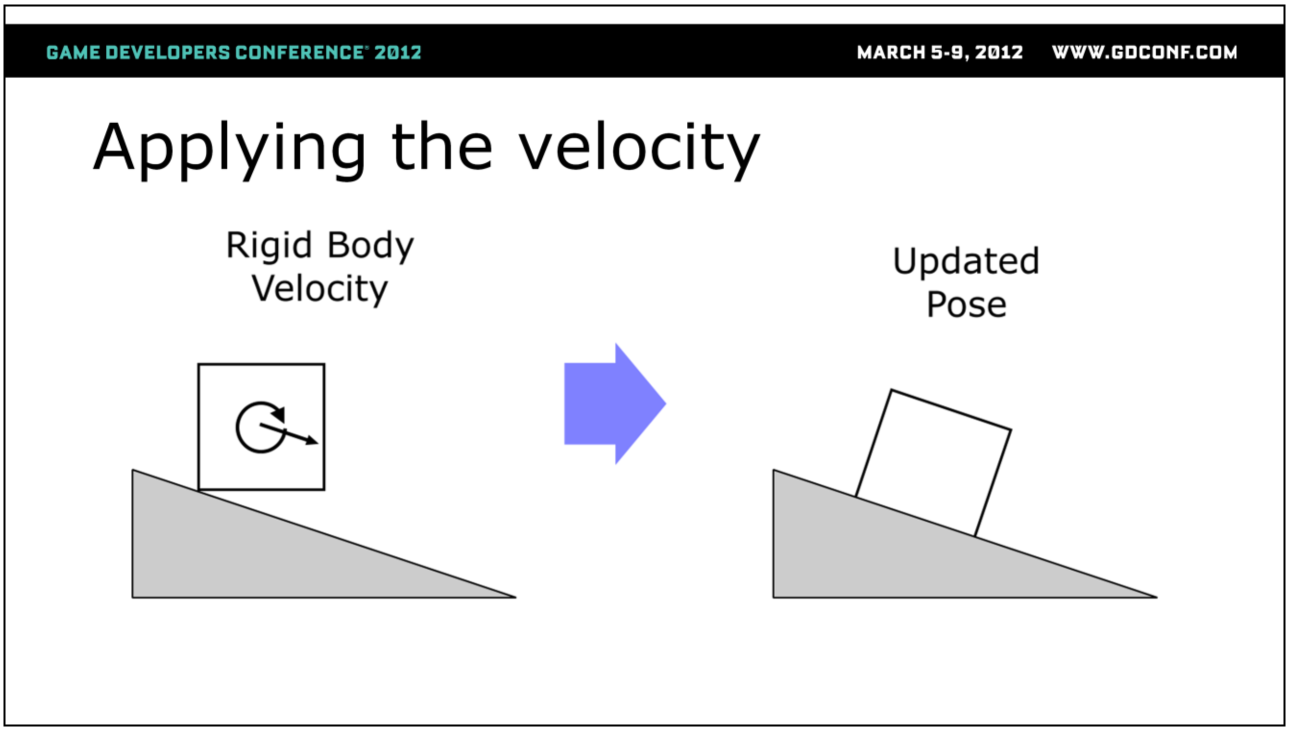
Recall that earlier we said that we can apply an impulse anywhere on a rigid body by calculating the equivalent impulse in rigid body coordinates and applying that. The rigid body impulse is shown in the right hand picture. Notice how applying the impulse off-center causes a rotation as well as a linear impulse.


GAME DEVELOPERS CONFERENCE® 2012
MARCH 5-9, 2012 WWW.GDCONF.COM

Applying the impulse

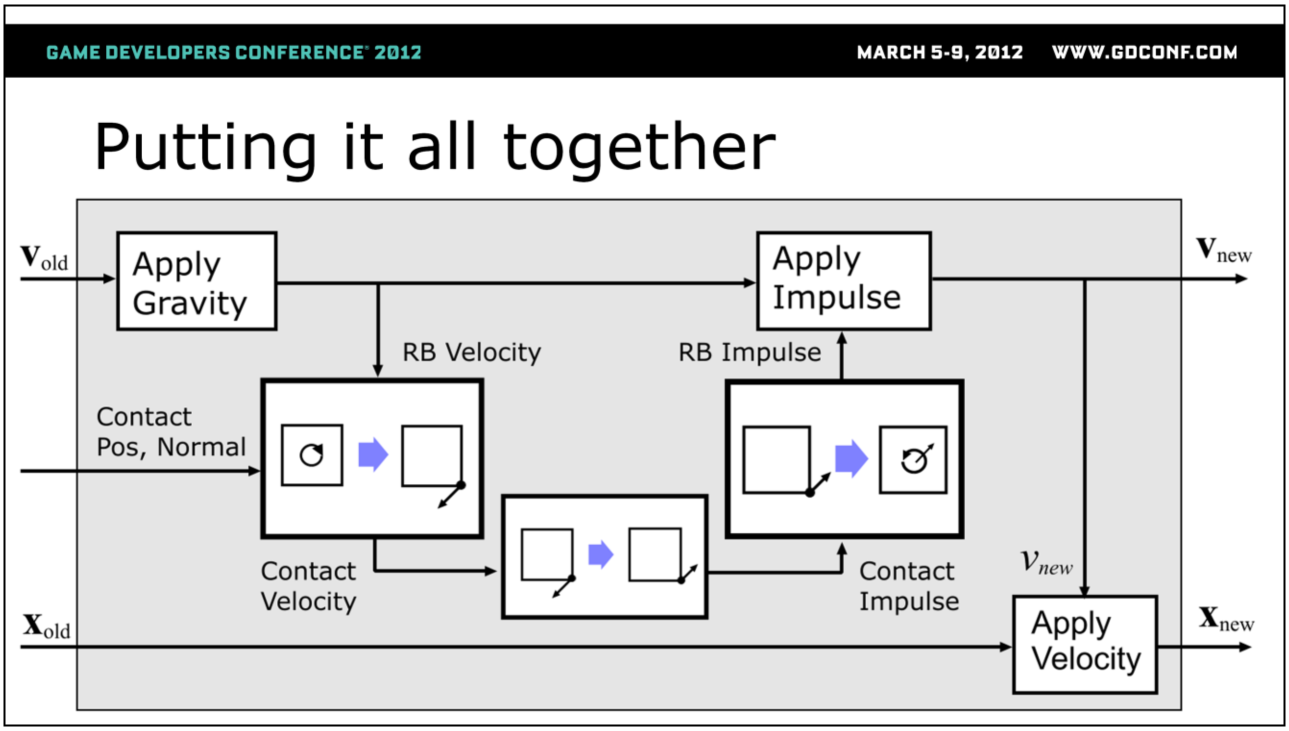
Rigid Body Impulse
Rigid Body Velocity

When we apply the impulse to the unconstrained velocity, the linear part of the new velocity aligns with the slope, just as we had forseen.



Applying the velocity

Now all we need to do is apply the velocity to update the position. The picture shows the box rotated so that it is parallel with the ground. This will probably take many frames, and at some point we are going to get more contacts from the collision detection to stop it rotating further through the slope. We'll talk about multiple contacts later.

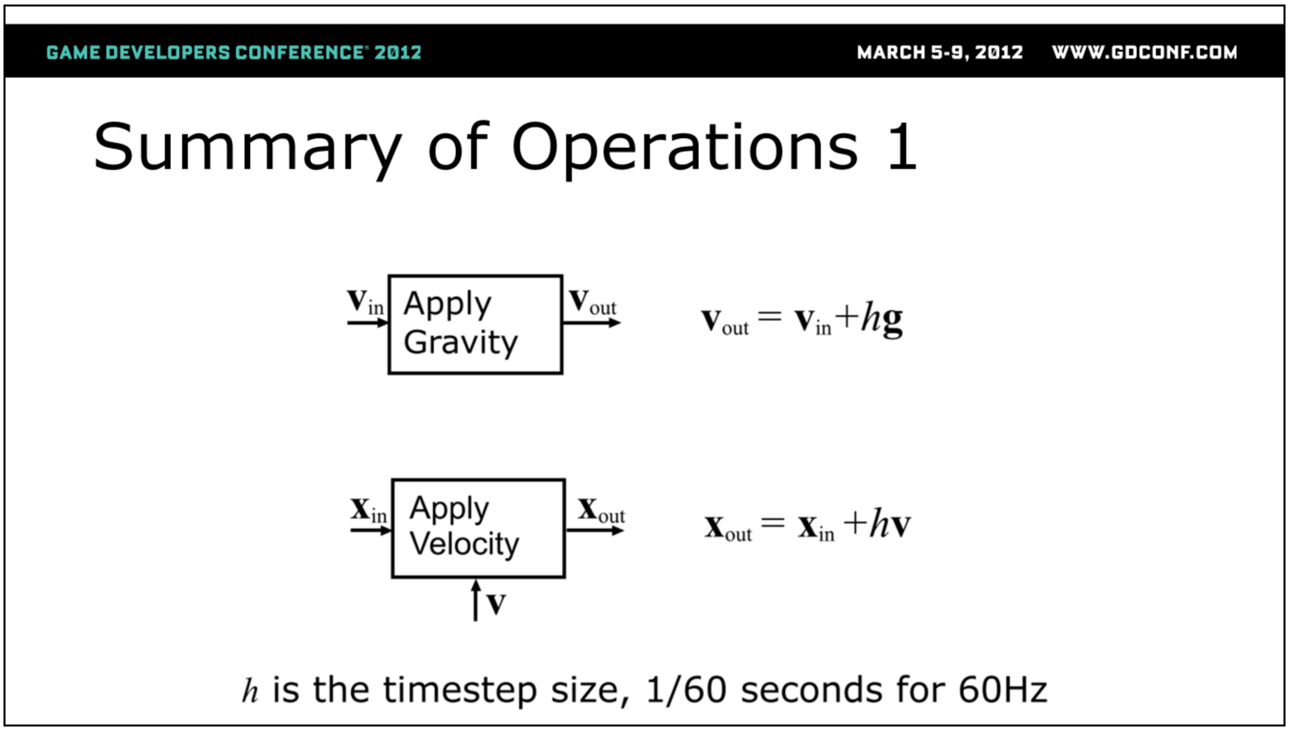


GAME DEVELOPERS CONFERENCE® 2012 MARCH 5-9, 2012 WWW.GDCONF.COM

Putting it all together

Putting all the previous steps together, this is what we get.

- Apply Gravity
- Calculate the relative velocity at the contact point (along the contact normal)
- Calculate the impulse to apply at this point that would make this relative velocity zero
- Calculate this impulse in rigid body coordinates
- Apply this rigid body impulse to the rigid body velocity
- Update the rigid body pose using the rigid body velocity



Summary of Operations 1

$$\begin{tikzcd} {} \arrow[r, "\mathbf{v}_{\text{in}}"] & \boxed{\text{Apply Gravity}} \arrow[r, "\mathbf{v}_{\text{out}}"] & {} \end{tikzcd}$$

$$\mathbf{v}_{\text{out}} = \mathbf{v}_{\text{in}} + h\mathbf{g}$$

$$\begin{tikzcd} {} \arrow[r, "\mathbf{x}_{\text{in}}"] & \boxed{\text{Apply Velocity}} \arrow[r, "\mathbf{x}_{\text{out}}"] & {} \\ & {} \arrow[u, "\mathbf{v}"'] & \end{tikzcd}$$

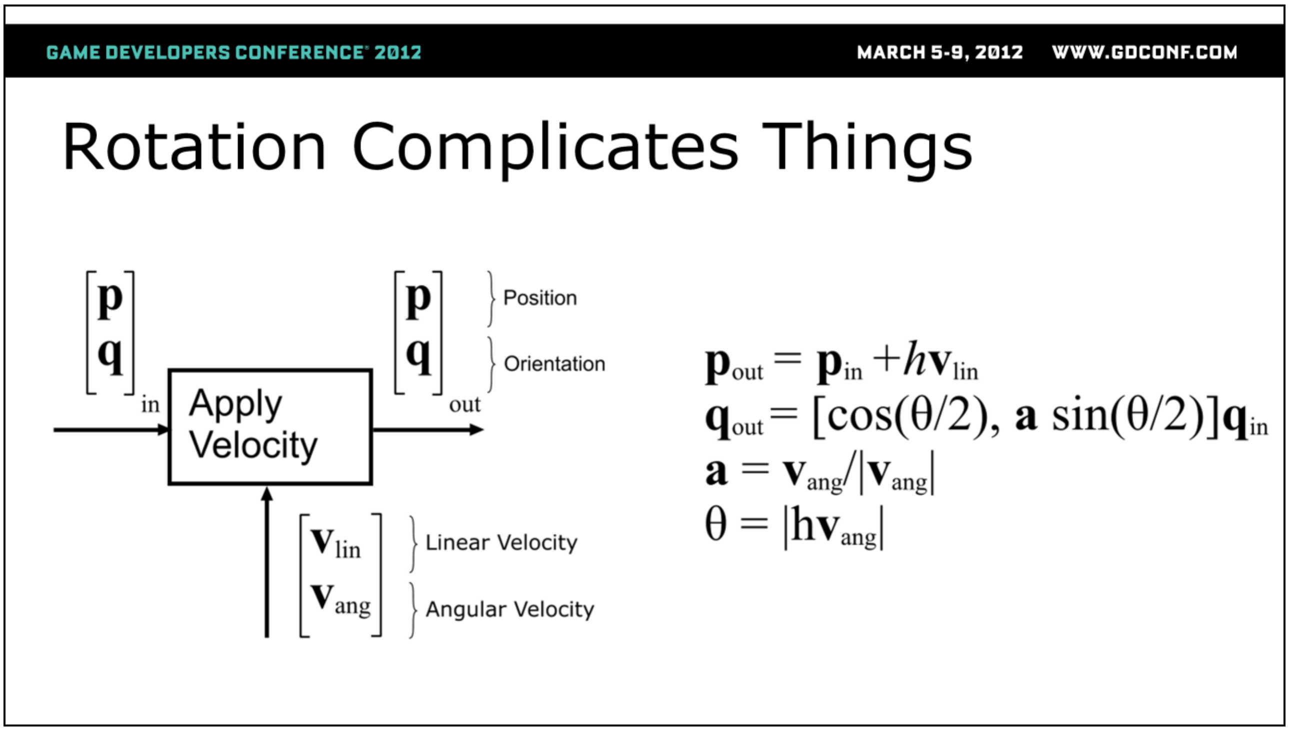
$$\mathbf{x}_{\text{out}} = \mathbf{x}_{\text{in}} + h\mathbf{v}$$

h is the timestep size, 1/60 seconds for 60Hz

Now we'll show how to implement each box in the diagram using math.

The simple update rules for applying gravity and velocity are called Euler integration. For people who know about numerical integration already, from these isolated blocks it may look like we are using explicit Euler, which is only conditionally stable. Overall though, we are doing a semi-implicit Euler which is unconditionally stable. See the time-stepping papers by Anitescu for more information on this.

There are more complicated integrators available, but they don't do well in systems with discontinuous changes like rigid body impacts. Also, even though these integrators are more accurate, in games we generally value stability and speed more than accuracy.


GAME DEVELOPERS CONFERENCE® 2012 MARCH 5-9, 2012 WWW.GDCONF.COM

Rotation Complicates Things

$\begin{bmatrix}\mathbf{p}\\ \mathbf{q}\end{bmatrix}_{\text{in}}$
Apply Velocity
$\begin{bmatrix}\mathbf{p}\\ \mathbf{q}\end{bmatrix}_{\text{out}}$
Position
Orientation
$\begin{bmatrix}\mathbf{v}_{\text{lin}}\\ \mathbf{v}_{\text{ang}}\end{bmatrix}$
Linear Velocity
Angular Velocity

$$\mathbf{p}_{\text{out}} = \mathbf{p}_{\text{in}} + h\mathbf{v}_{\text{lin}}$$
$$\mathbf{q}_{\text{out}} = [\cos(\theta/2),\ \mathbf{a}\sin(\theta/2)]\mathbf{q}_{\text{in}}$$
$$\mathbf{a} = \mathbf{v}_{\text{ang}}/|\mathbf{v}_{\text{ang}}|$$
$$\theta = |h\mathbf{v}_{\text{ang}}|$$

The velocity application in the last slide contains a slight problem. I wrote it the way I think about it, but it’s not actually true.

The rigid body pose, x, is a 7D vector, a position and a quaternion, whereas the rigid body velocity, v is a 6D vector. We can’t add these things together.

So how is it done?

The linear part is just the same as in the last slide, but to apply the angular velocity to the quaternion requires the formulae on this slide.

Apply Impulse

$$\begin{tikzcd}
\mathbf{v}_{in} \arrow[r] & \boxed{\text{Apply Impulse}} \arrow[r] & \mathbf{v}_{out} \\
& \text{Rigid Body Impulse, } \boldsymbol{\rho} \arrow[u] &
\end{tikzcd}$$

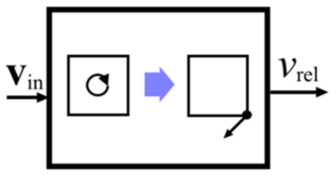
$$\mathbf{v}_{out} = \mathbf{v}_{in} + \mathbf{M}^{-1}\boldsymbol{\rho}$$

Mass Matrix

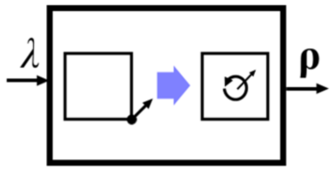
$$\mathbf{M} = \begin{bmatrix} m & & & & & \\ & m & & & & \\ & & m & & & \\ & & & \cdot & \cdot & \cdot \\ & & & \cdot & \cdot & \cdot \\ & & & \cdot & \cdot & \cdot \end{bmatrix} \text{Inertia Tensor}$$

This is how rigid body impulses are applied. In particle dynamics, mass is a single number, but here M is a 6*6 matrix. The first 3 diagonal elements are just the mass, but the bottom right 3*3 block is something called the inertia tensor. Just as the mass specifies how hard it is to move a body linearly, the inertia specifies how hard it is to rotate a body around its center of mass. There are standard formula for the inertia of primitives like cubes, etc, a standard way of calculating the inertia of a triangle mesh (with uniform density), and a standard way of calculating the inertia of rigidly attached components when you know the inertia of each component. I usually just look that stuff up on the internet.

Summary of operations 2



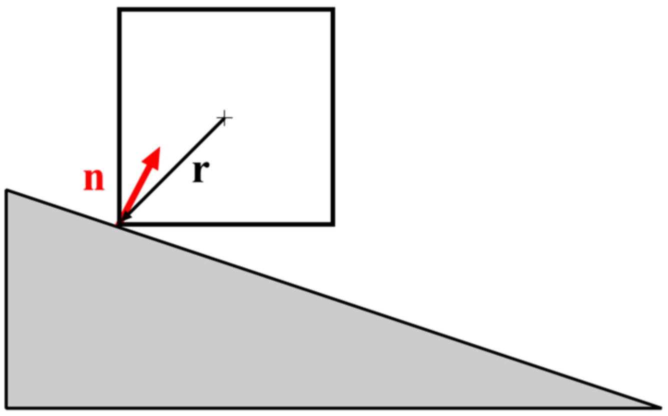
$$v_{\text{rel}} = \mathbf{J}\mathbf{v}_{\text{in}}$$



$$\boldsymbol{\rho} = \mathbf{J}^T\lambda$$

Ok, this seems like a bit of a cheat, saying that v_in is transformed to v_rel by multiplying by matrix J because it is not much more informative than just drawing a black box. I'll show what J is in a moment though. What is interesting though is that to convert an impulse at a point to a rigid body impulse you multiply by the transpose of J. Erin covered why this is the case in one of his previous GDC presentations.

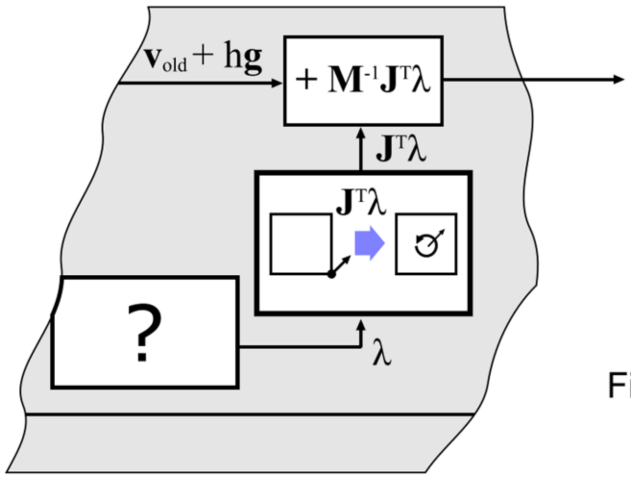
What is **J**?



$$\mathbf{J} = [\mathbf{n} \mid \mathbf{r} \times \mathbf{n}]$$

The red arrow is the collision normal provided by the collision detection system, and r shows the position of the contact point relative to the center of mass. In high school I remember learning that torque is force multiplied by the perpendicular distance, and the cross product (r x n) here is like the 3D equivalent of that.

Calculating the impulse (λ)


$\mathbf{v}_{\text{old}} + \mathbf{hg}$
$+ \mathbf{M}^{-1}\mathbf{J}^{T}\lambda$
$\mathbf{J}^{T}\lambda$
$\mathbf{J}^{T}\lambda$
?
λ

Final velocity in terms of λ

$$\mathbf{v}_{\text{new}} = \mathbf{v}_{\text{old}} + \mathbf{hg} + \mathbf{M}^{-1}\mathbf{J}^{T}\lambda$$

Final relative velocity

$$\mathbf{J}\mathbf{v}_{\text{new}} = \mathbf{J}(\mathbf{v}_{\text{old}} + \mathbf{hg}) + \mathbf{J}\mathbf{M}^{-1}\mathbf{J}^{T}\lambda$$

Final relative velocity is zero when:

$$\lambda = -(\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^{T})^{-1}\,\mathbf{J}(\mathbf{v}_{\text{old}} + \mathbf{hg})$$

There is just one box on the diagram that we have not yet converted to math, the one that takes the relative velocity at the contact point and works out how much impulse to apply at the point to eliminate it.

I said earlier that overall we will make the method semi implicit to ensure that it is unconditionally stable, and this is where we’re going to achieve that.

The way we do this is to ensure that the contact constraint is enforced at the end of the timestep, not at the start. So even though we don’t know the impulse (lambda) yet, we’ll calculate what the velocity will be at the end of the timestep in terms of it, calculate the relative velocity in terms of that, then solve to find out what the impulse should be.

First, what is the final velocity in terms of v_rel and lambda

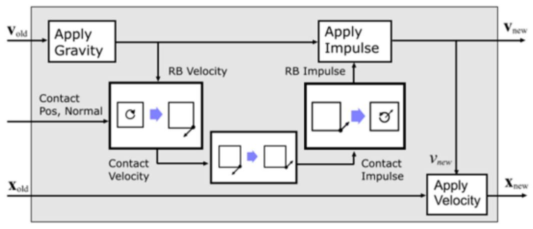
V_new = V_rel + M^{-1}J^T lambda

We want the relative velocity to be zero at the end of the timestep

So we want J v_new = 0

J v_new = J V_rel + JM^{-1}J^T lambda = 0

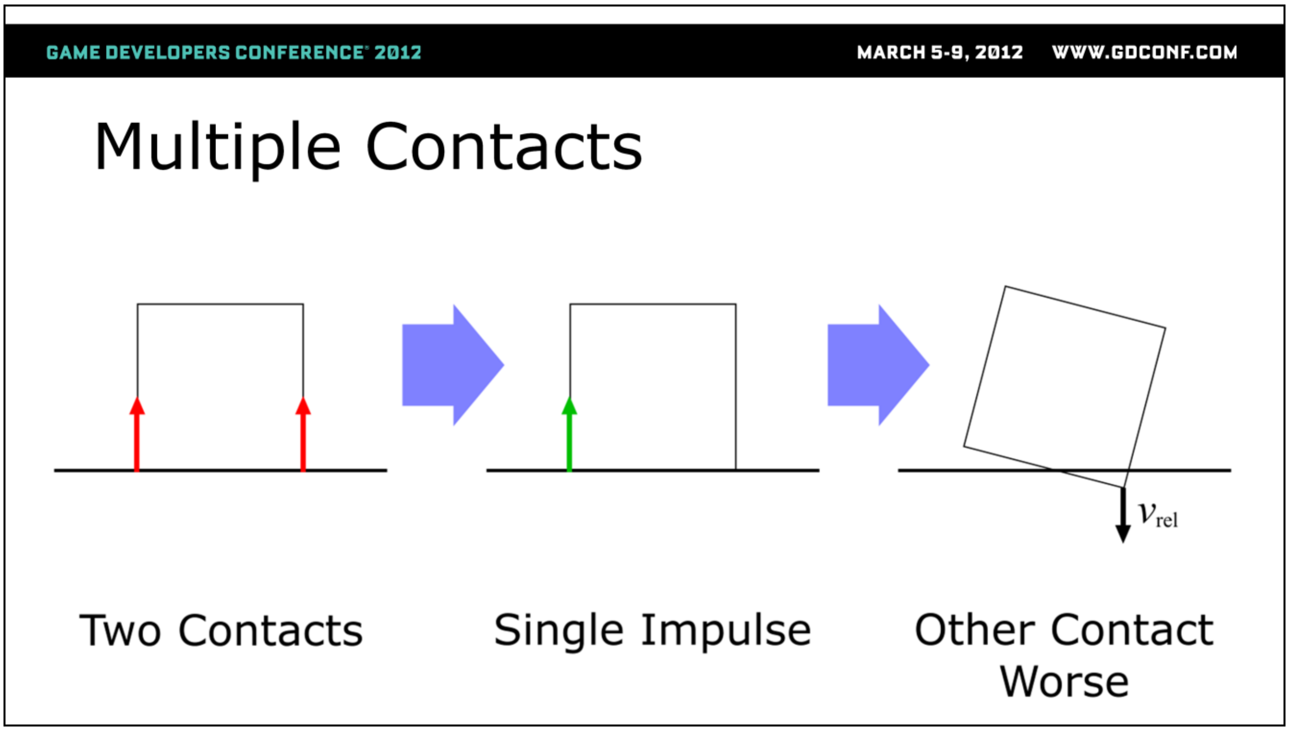
Single Contact Algorithm



=

```
solveSingle($\mathbf{x}, \mathbf{v}, \mathbf{J}, \mathbf{M}, h, \mathbf{g}$)
{
  $\lambda = (-(\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^{T})^{-1}\mathbf{J}(\mathbf{v} + h\mathbf{g}))^{+}$
  $\mathbf{v} = \mathbf{v} + \mathbf{M}^{-1}\mathbf{J}^{T}\lambda$
  $\mathbf{x} = \mathbf{x} + h\mathbf{v}$
}
```

Putting it all together, this is what our diagram looks like in code.



GAME DEVELOPERS CONFERENCE® 2012 MARCH 5-9, 2012 WWW.GDCONF.COM

Multiple Contacts

Two Contacts

Single Impulse

Other Contact Worse

Multiple contact points

This is where things start getting tricky

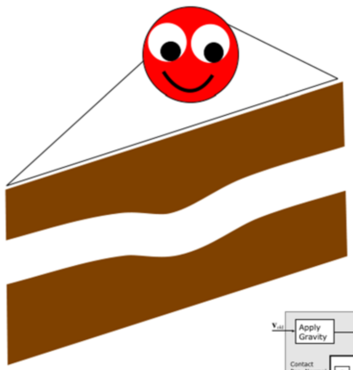
Applying an impulse at one contact point can affect the velocity at many other contact points

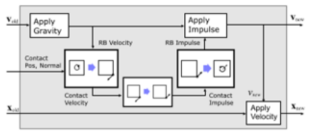
So we need to find a set of impulses, one for each contact so that when they are applied simultaneously, the velocity constraints are satisfied simultaneously (taking into account all the coupling between the contacts)

Multiple contacts

$$0 \leq \mathbf{v}_{\text{rel}} \perp 0 \leq \lambda$$

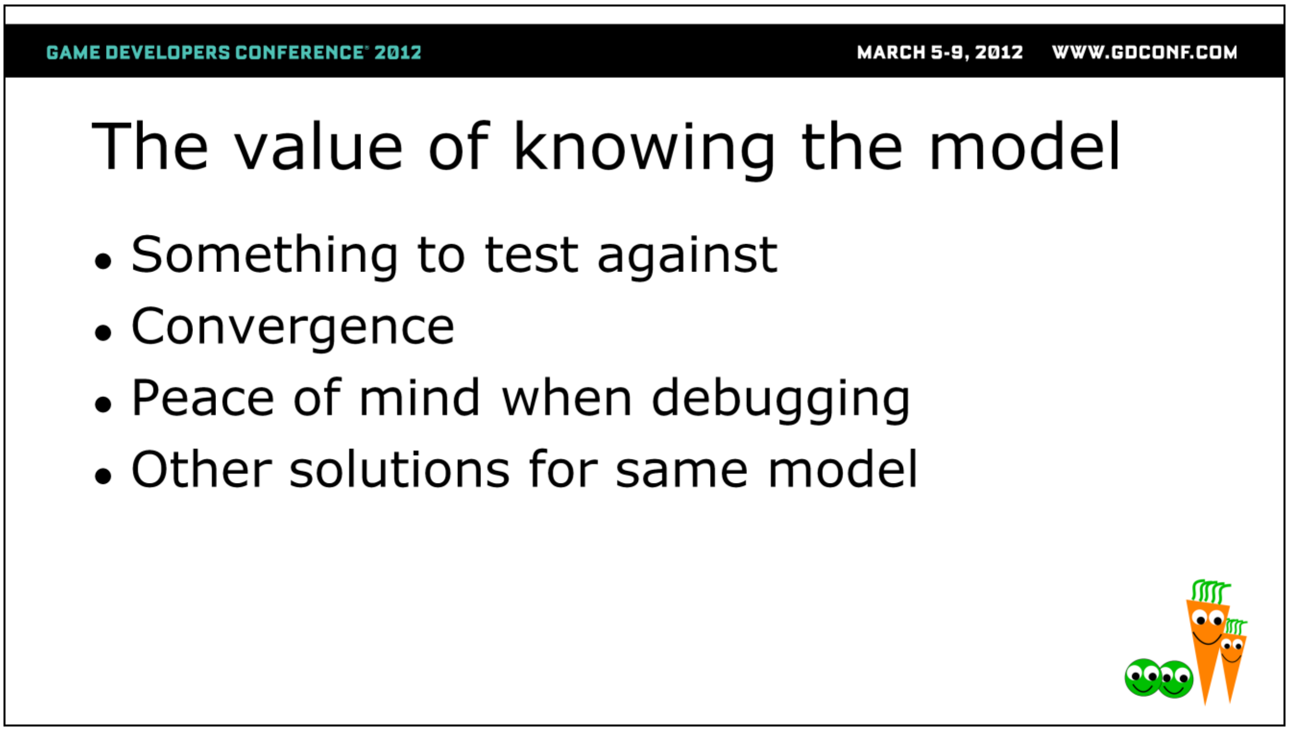
Model first





Algorithm second

Previous presentations have just presented the multiple contact algorithm (which is almost the same as the previous one), but today I'm going to do things a bit differently. I'm first going to show what the high level model the algorithm solves is, then I'm going to show the algorithm.



GAME DEVELOPERS CONFERENCE® 2012 MARCH 5-9, 2012 WWW.GDCONF.COM

The value of knowing the model

- Something to test against
- Convergence
- Peace of mind when debugging
- Other solutions for same model

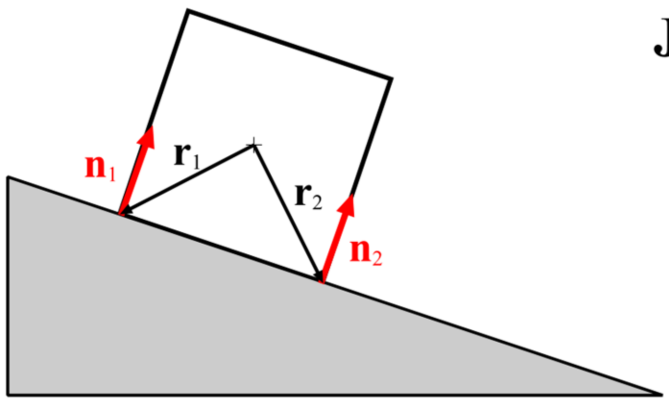
Why am I doing this to you? You just need to know the final algorithm so that you can code it, right?

My experience of writing solvers is that inevitably there is some jitter or other undesirable behavior the first time you run them. At that point you think, hmm, is this a bug, or is it a fundamental problem? How do I know that applying all these impulses locally is going to give a globally stable solution?

So this is the advantage to knowing the model that you are approximately solving, once you know what the perfect solution should be you can measure how close your approximate solution is to it. Also, when you know the model you can prove (or read a proof that was written already) that your approximate algorithm converges to it, and then if something weird happens you can be confident that it is just a bug in your code and not some fundamental math problem.

Also, many people have written solver for similar models outside of games, and if you know the model you have something to pattern match against when reading papers from other fields.

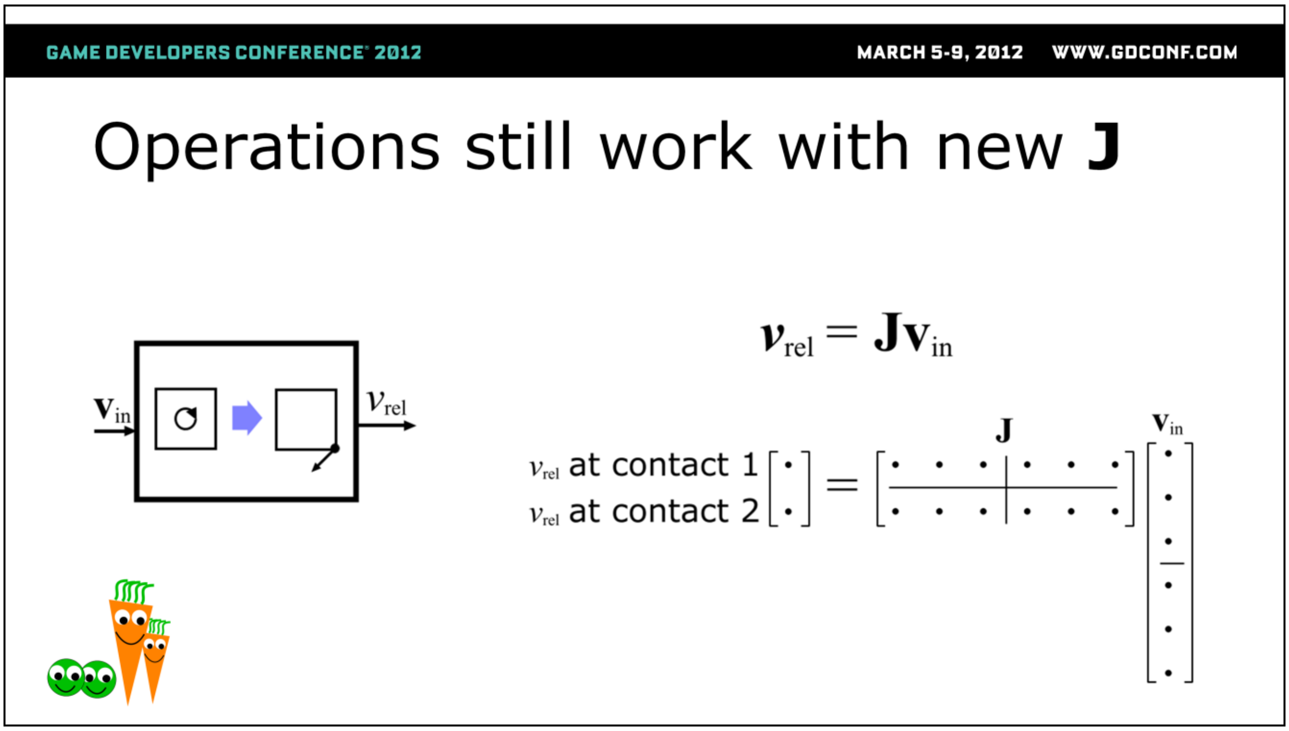
What is **J**?



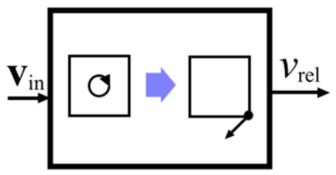
$$\mathbf{J} = \left[\begin{array}{c|c} \mathbf{n}_1 & \mathbf{r}_1 \times \mathbf{n}_1 \\ \hline \mathbf{n}_2 & \mathbf{r}_2 \times \mathbf{n}_2 \end{array}\right]$$

$$= \left[\begin{array}{ccc|ccc} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{array}\right]$$

To build the multiple contact J matrix we just introduce an extra row per contact, and each row is built in the same way as in the single contact case.



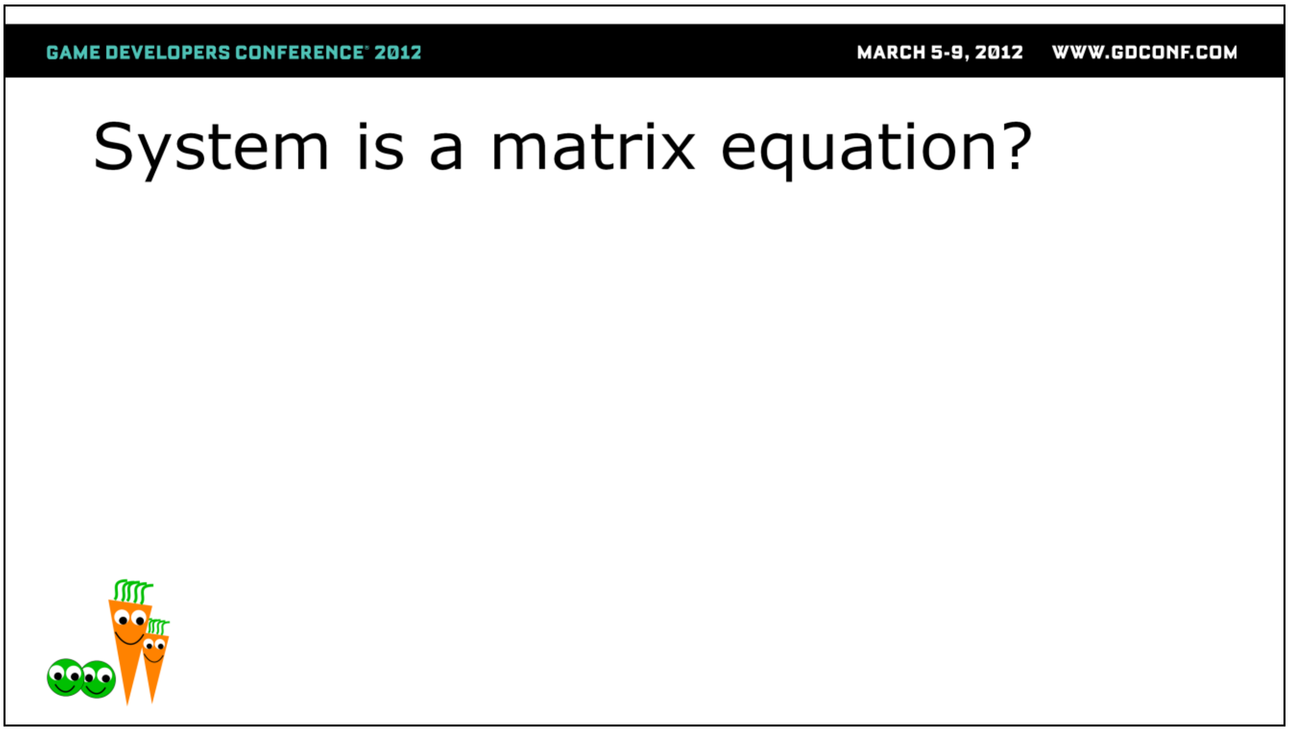
Operations still work with new **J**



$$v_{\text{rel}} = \mathbf{J}\mathbf{v}_{\text{in}}$$

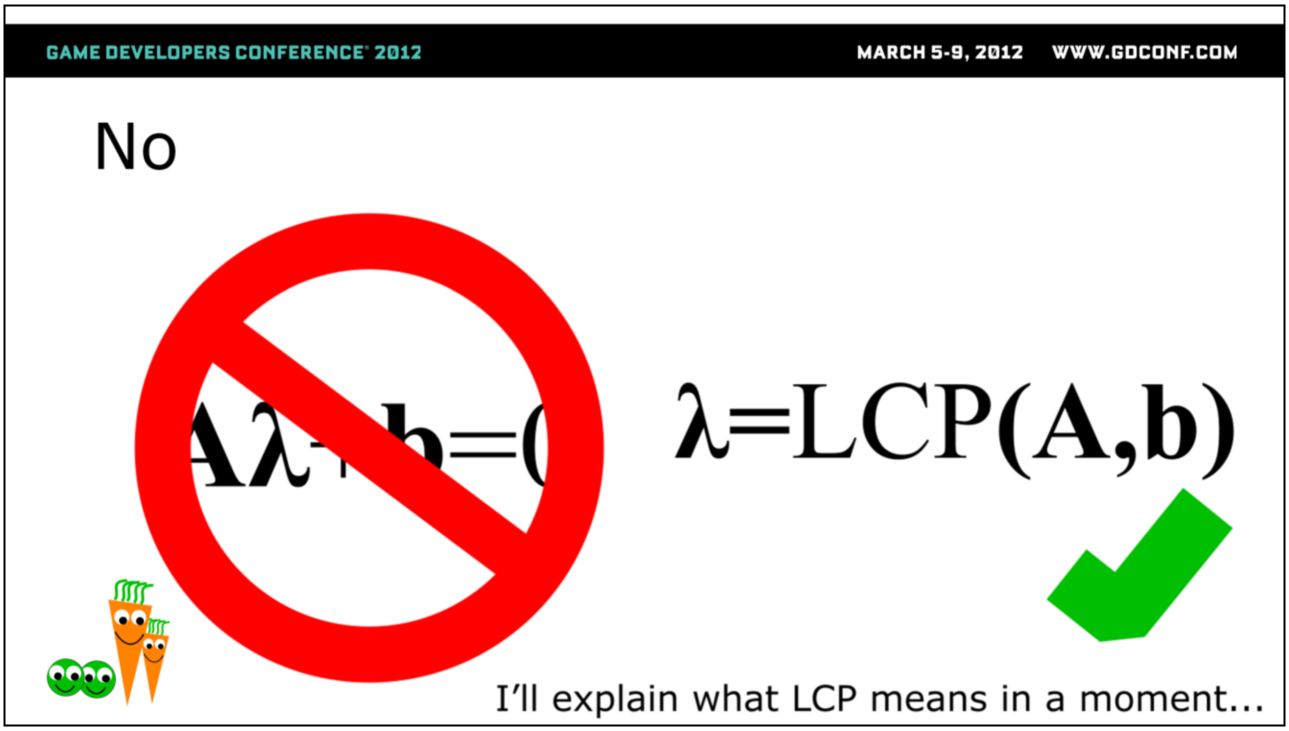
$$\begin{array}{r} v_{\text{rel}} \text{ at contact 1} \\ v_{\text{rel}} \text{ at contact 2} \end{array} \begin{bmatrix} \cdot \\ \cdot \end{bmatrix} = \overset{\mathbf{J}}{\left[\begin{array}{ccc|ccc} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{array}\right]} \overset{\mathbf{v}_{\text{in}}}{\left[\begin{array}{c} \cdot \\ \cdot \\ \cdot \\ \hline \cdot \\ \cdot \\ \cdot \end{array}\right]}$$

All the previous operations work with the new J, except now the result is a vector with one element per contact.



System is a matrix equation?

At this point you might be thinking that the multiple contact problem is a matrix equation that could be solved using a standard linear solver algorithm. Is this right?



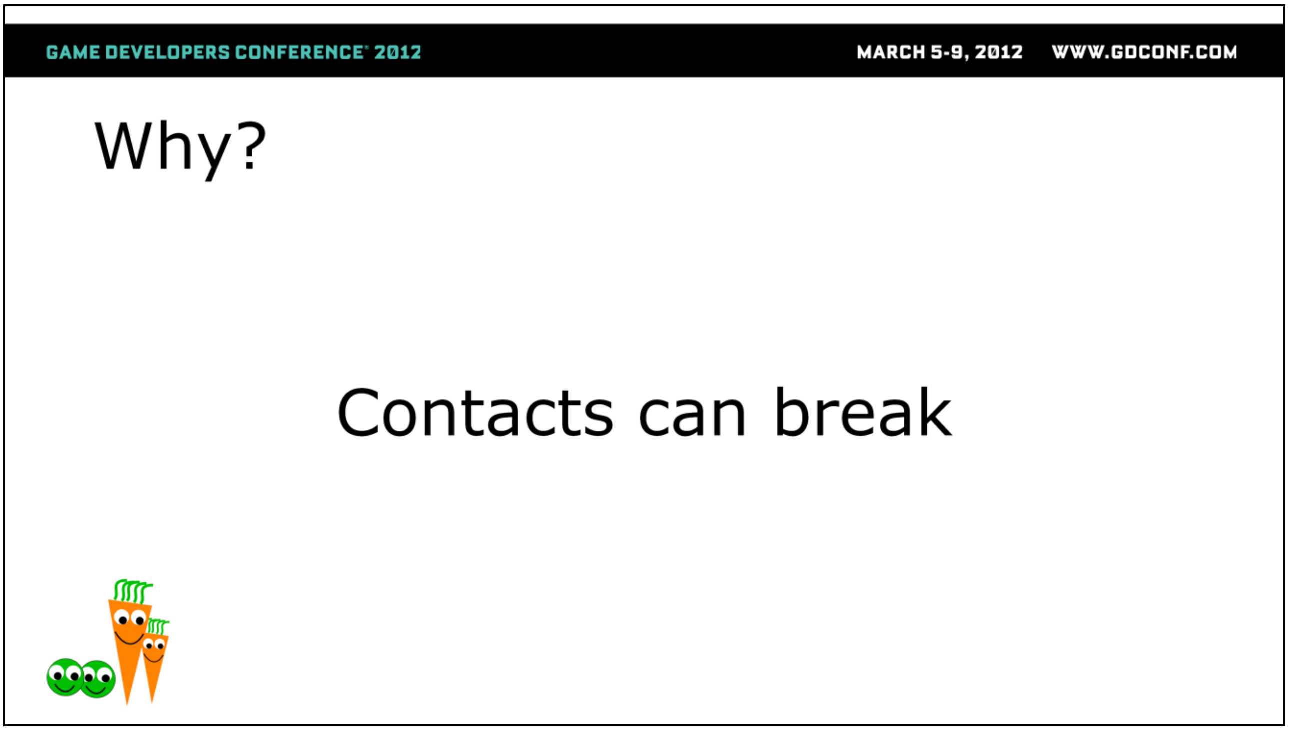
No

$\cancel{A\lambda+b=0}$ $\lambda=\mathrm{LCP}(A,b)$ ✓

I'll explain what LCP means in a moment...

No.

Instead of being a linear system, what we have is something else called a linear complementarity problem (LCP). Don't worry, I'll explain what the expressions on this slide mean in a moment.



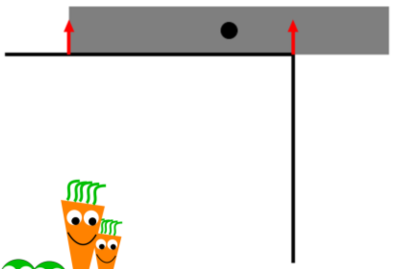
GAME DEVELOPERS CONFERENCE® 2012 MARCH 5-9, 2012 WWW.GDCONF.COM

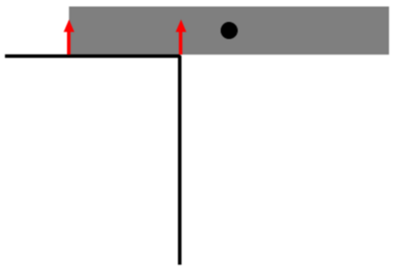
Why?

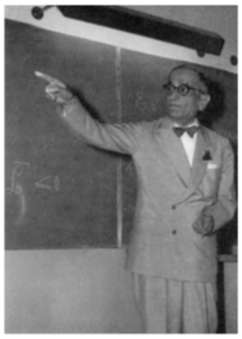
Contacts can break

But first, why is it not a linear system? The answer is that contacts can break.

When Should Contacts Break?



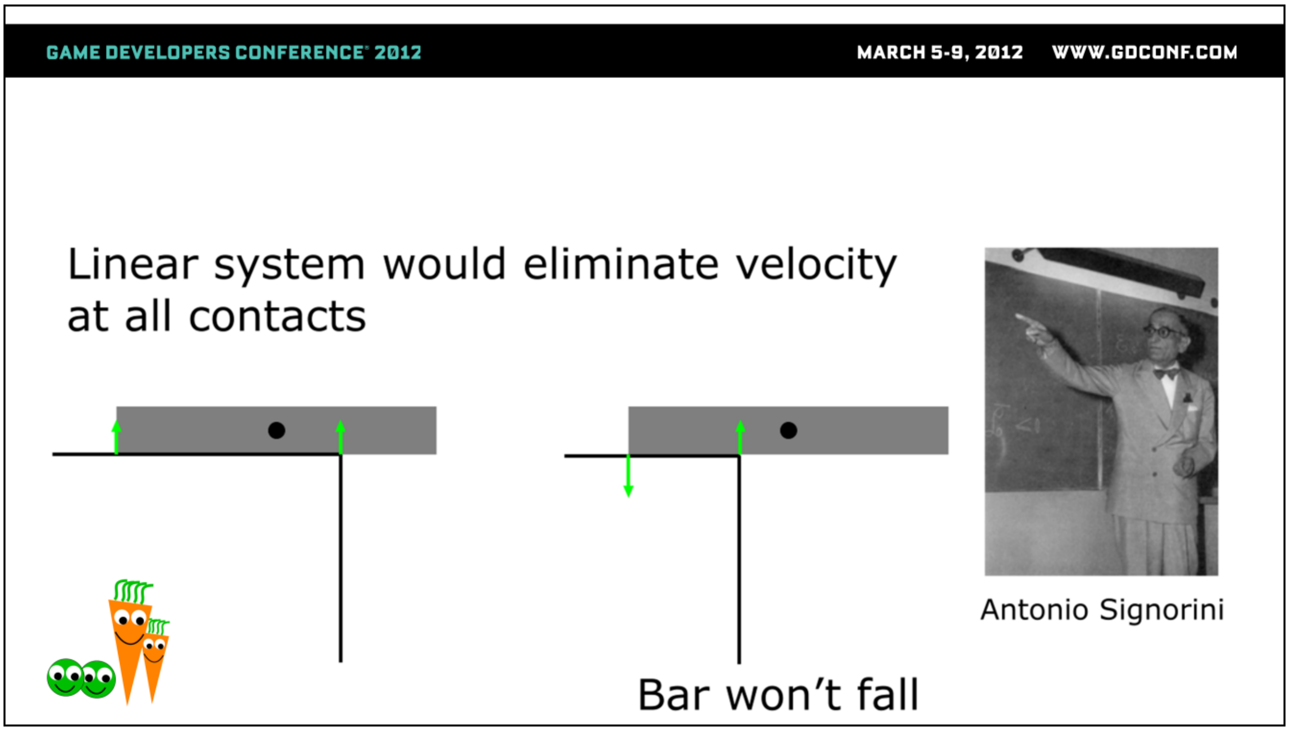




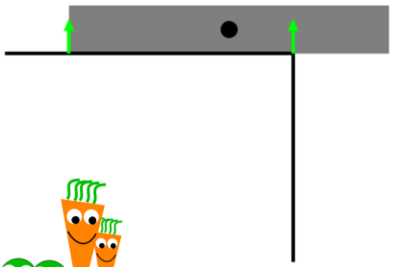
Antonio Signorini

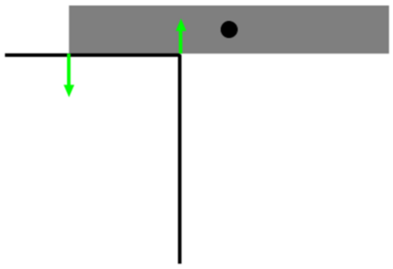
Here we have two pens sitting on the edge of a table. The circle represents the center of mass. The collision detection system has generated two contacts in each case, shown by the red arrows.

Intuitively, the pen on the left should stay on the table, and the one on the right should fall off the table. As the pen falls off the table, the leftmost contact should stop applying force. We call this a breaking contact.

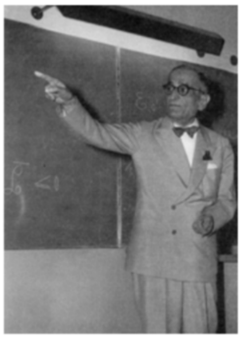


Linear system would eliminate velocity at all contacts





Bar won't fall



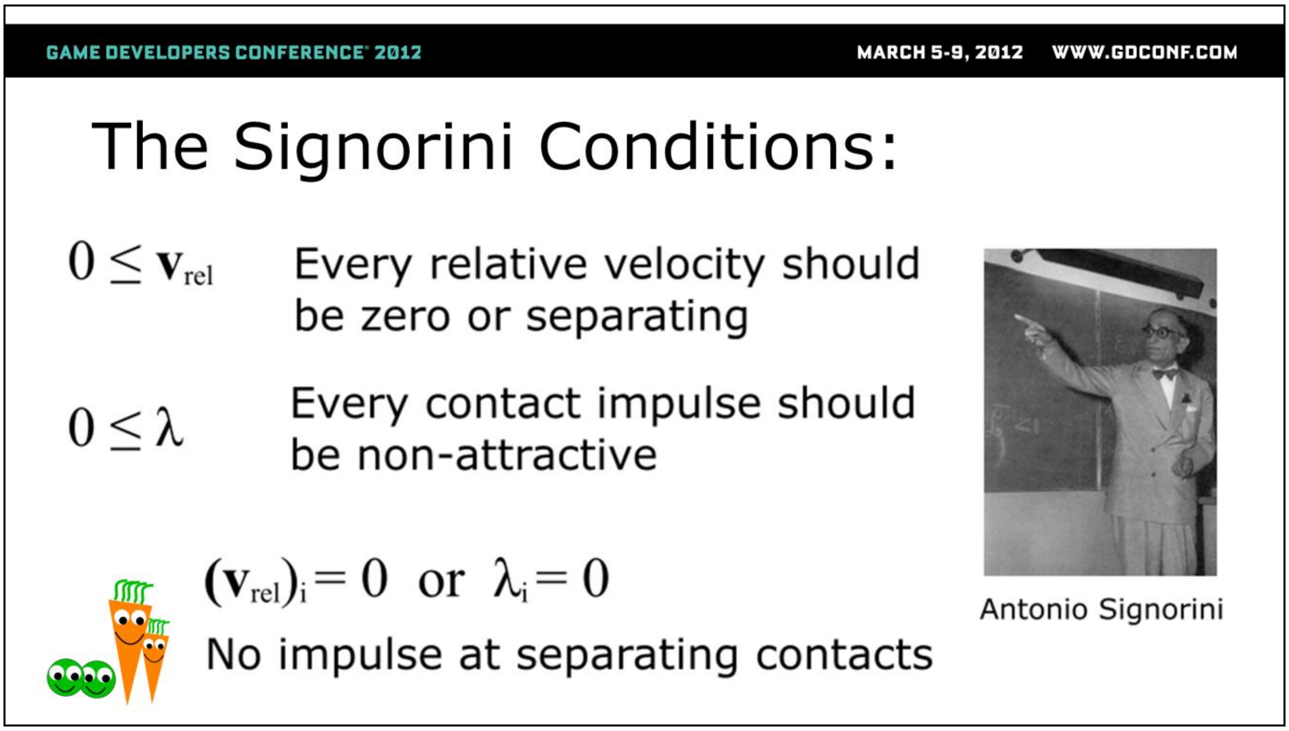
Antonio Signorini

Suppose we model the contact impulses as a linear system.

This means is that we would solve a (matrix) equation to calculate the impulses that when applied simultaneously would set all the relative velocities to zero.

The problem is that the only way the solver can achieve this in the right hand picture is to apply an attractive force on the left contact. This is shown by the downward green arrow. The attractive force and zero relative velocity mean that the bar won't fall.

So a linear system can give attractive impulses, which is fine for simulating a pen that is jointed to the table, or if the table and pen are magnetic, but that's not what we have here.



GAME DEVELOPERS CONFERENCE® 2012 MARCH 5-9, 2012 WWW.GDCONF.COM

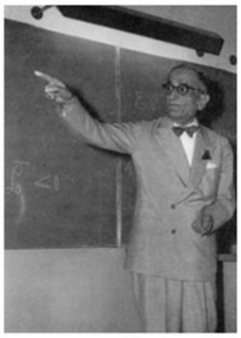
The Signorini Conditions:

$0 \le \mathbf{v}_{\text{rel}}$ Every relative velocity should be zero or separating

$0 \le \lambda$ Every contact impulse should be non-attractive

$(\mathbf{v}_{\text{rel}})_i = 0 \text{ or } \lambda_i = 0$

No impulse at separating contacts



Antonio Signorini

How do we specify what we do want to happen in terms of impulses and relative velocities?

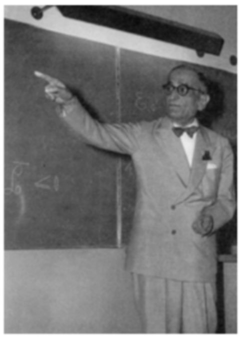
We've seen that for contacts, we want impulses to be non attractive (non negative), and we want relative velocities to be zero or separating (also non negative).

There is one other condition that isn't obvious from this example, as soon as a contact is separating, no more force impulse should be applied. A formal way of saying this is that constraints must do no work, which is a law that has many names, like Gauss' principle of least constraint, D'alembert's principle and the principle of virtual work.

This slides is just these three conditions written in math. They are called the Signorini conditions after Antonio Signorini who first formalized them. Here is a picture of him.

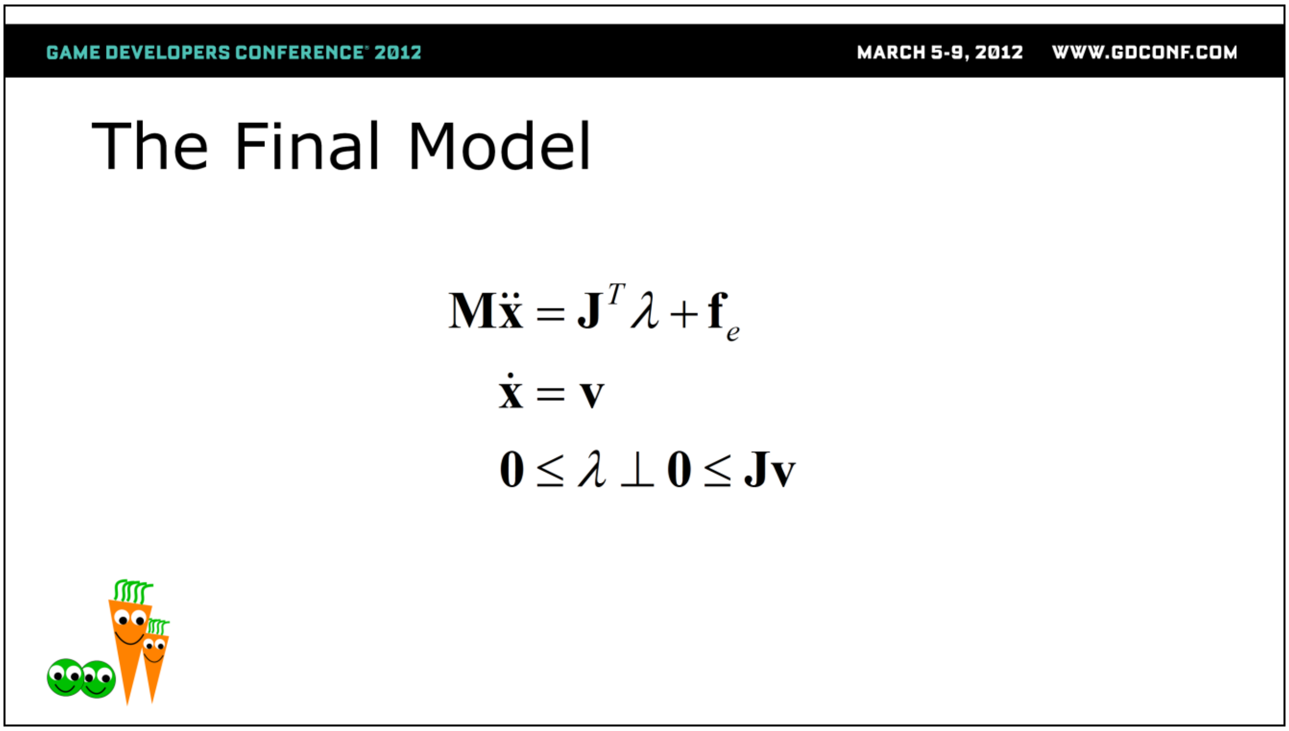
A compact way to write the three conditions in one line of math:

$$0 \leq \mathbf{v}_{\text{rel}} \perp 0 \leq \lambda$$


Antonio Signorini

The meaning of this expression is exactly the same as the three Signorini conditions from the previous slide, it’s just a more compact way of writing them.

The upside down T means “is complementary to” and velocity is complementary to impulse has the same meaning as the third Signorini condition.



GAME DEVELOPERS CONFERENCE® 2012 MARCH 5-9, 2012 WWW.GDCONF.COM

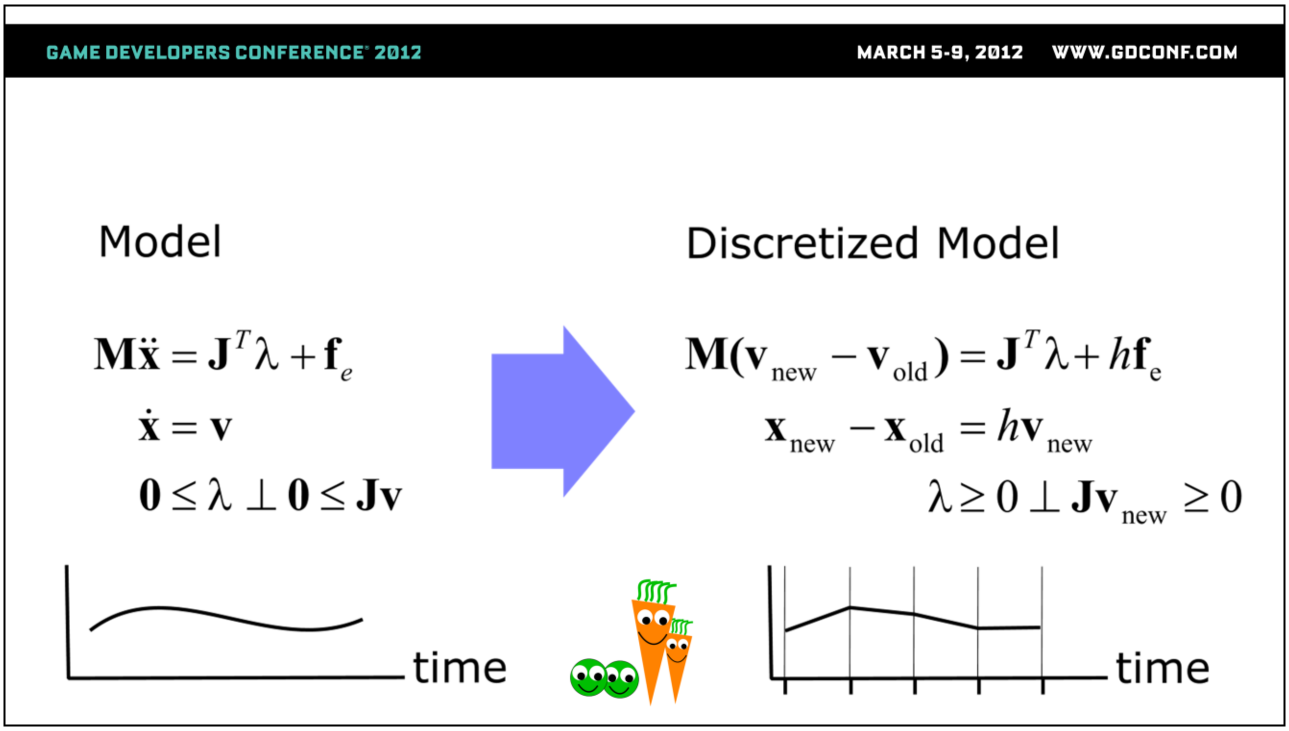
The Final Model

$$\mathbf{M}\ddot{\mathbf{x}} = \mathbf{J}^T\lambda + \mathbf{f}_e$$

$$\dot{\mathbf{x}} = \mathbf{v}$$

$$\mathbf{0} \le \lambda \perp \mathbf{0} \le \mathbf{J}\mathbf{v}$$

So this is our final model. The first line is Newton’s second law of motion, the second line is the definition of velocity, and the third line is the Signorini condition from the previous slide.



Model

$$\mathbf{M}\ddot{\mathbf{x}} = \mathbf{J}^T\lambda + \mathbf{f}_e$$
$$\dot{\mathbf{x}} = \mathbf{v}$$
$$\mathbf{0} \le \lambda \perp \mathbf{0} \le \mathbf{J}\mathbf{v}$$

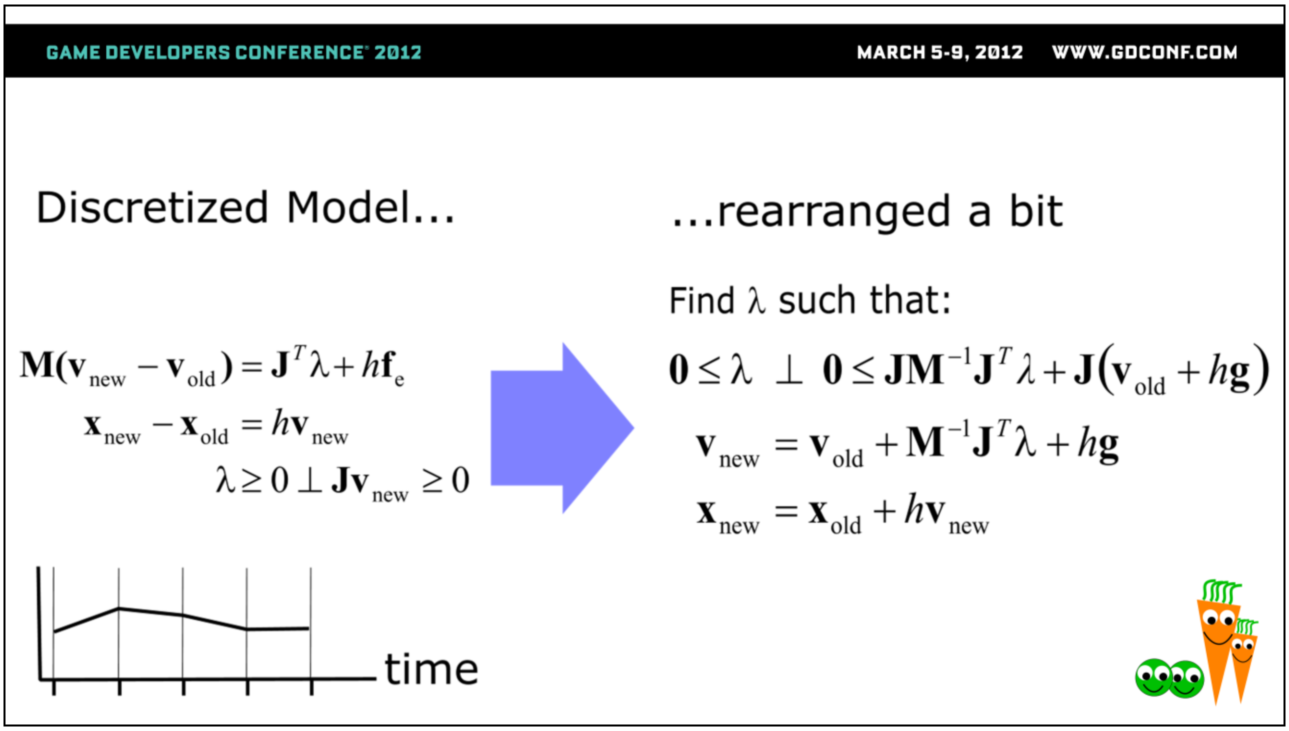
Discretized Model

$$\mathbf{M}(\mathbf{v}_{\text{new}} - \mathbf{v}_{\text{old}}) = \mathbf{J}^T\lambda + h\mathbf{f}_{\text{e}}$$
$$\mathbf{x}_{\text{new}} - \mathbf{x}_{\text{old}} = h\mathbf{v}_{\text{new}}$$
$$\lambda \ge 0 \perp \mathbf{J}\mathbf{v}_{\text{new}} \ge 0$$

Originally I didn't have the next three slides in the presentation, but in the rehearsal somebody pointed out that I didn't specify what LCP means or where it comes from. So here it is.

On the left we have this ideallized model which shows with infinite resolution how position and velocity vary over time, between collisions these graphs of these things are perfectly smooth. There is an example of such a function of time on the bottom left.

It is not possible to solve this model exactly in all interesting cases, and we only need to know the answer once per frame anyway. So we cut time into frame sized chunks and approximate the functions as straight lines between them. This is called doing a time discretization of the model. So you can see that we've just replaced acceleration with (v_new-v_old)/h etc.

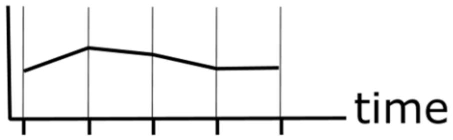


Discretized Model...

$$\mathbf{M}(\mathbf{v}_{\text{new}} - \mathbf{v}_{\text{old}}) = \mathbf{J}^T\lambda + h\mathbf{f}_{\text{e}}$$

$$\mathbf{x}_{\text{new}} - \mathbf{x}_{\text{old}} = h\mathbf{v}_{\text{new}}$$

$$\lambda \geq 0 \perp \mathbf{J}\mathbf{v}_{\text{new}} \geq 0$$



...rearranged a bit

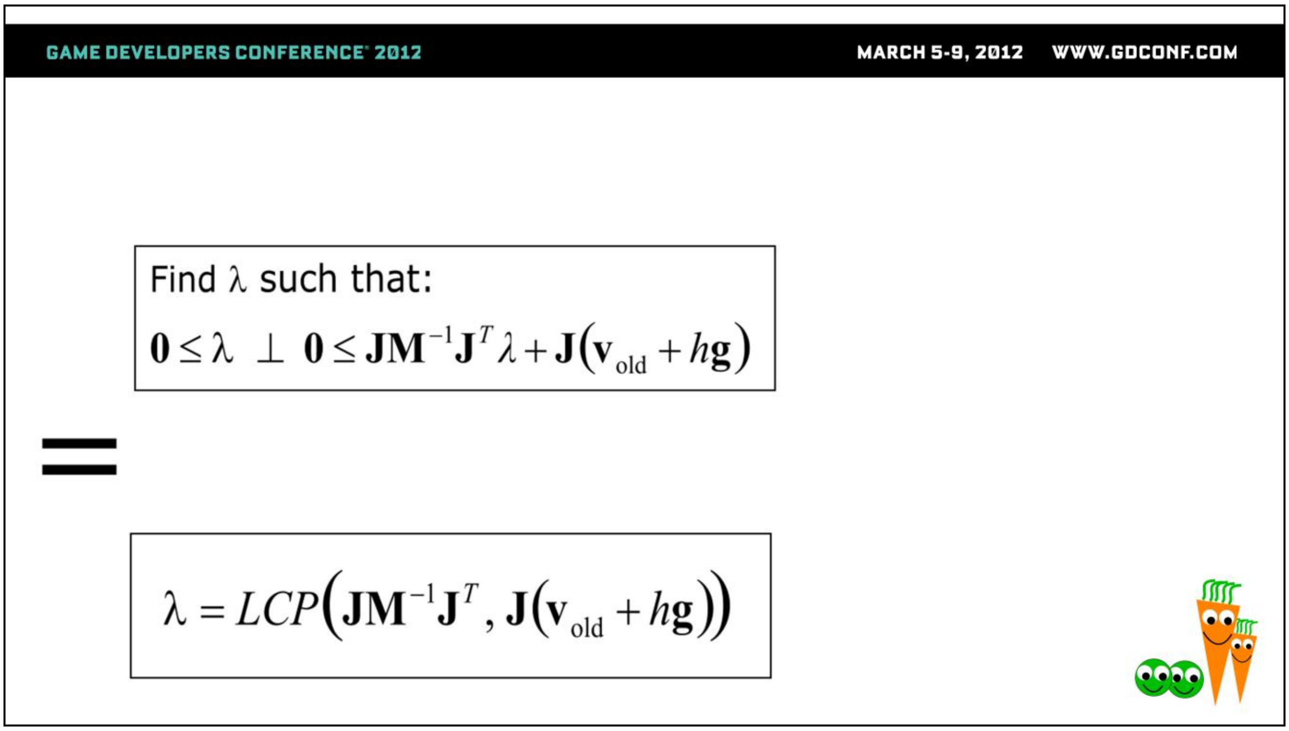
Find λ such that:

$$\mathbf{0} \leq \lambda \ \perp \ \mathbf{0} \leq \mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T\lambda + \mathbf{J}\left(\mathbf{v}_{\text{old}} + h\mathbf{g}\right)$$

$$\mathbf{v}_{\text{new}} = \mathbf{v}_{\text{old}} + \mathbf{M}^{-1}\mathbf{J}^T\lambda + h\mathbf{g}$$

$$\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} + h\mathbf{v}_{\text{new}}$$

We want to calculate impulses (lambda), apply them and then update position, so we can rearrange the discretized model to show us how to do this. The result is shown on the right.



Find λ such that:

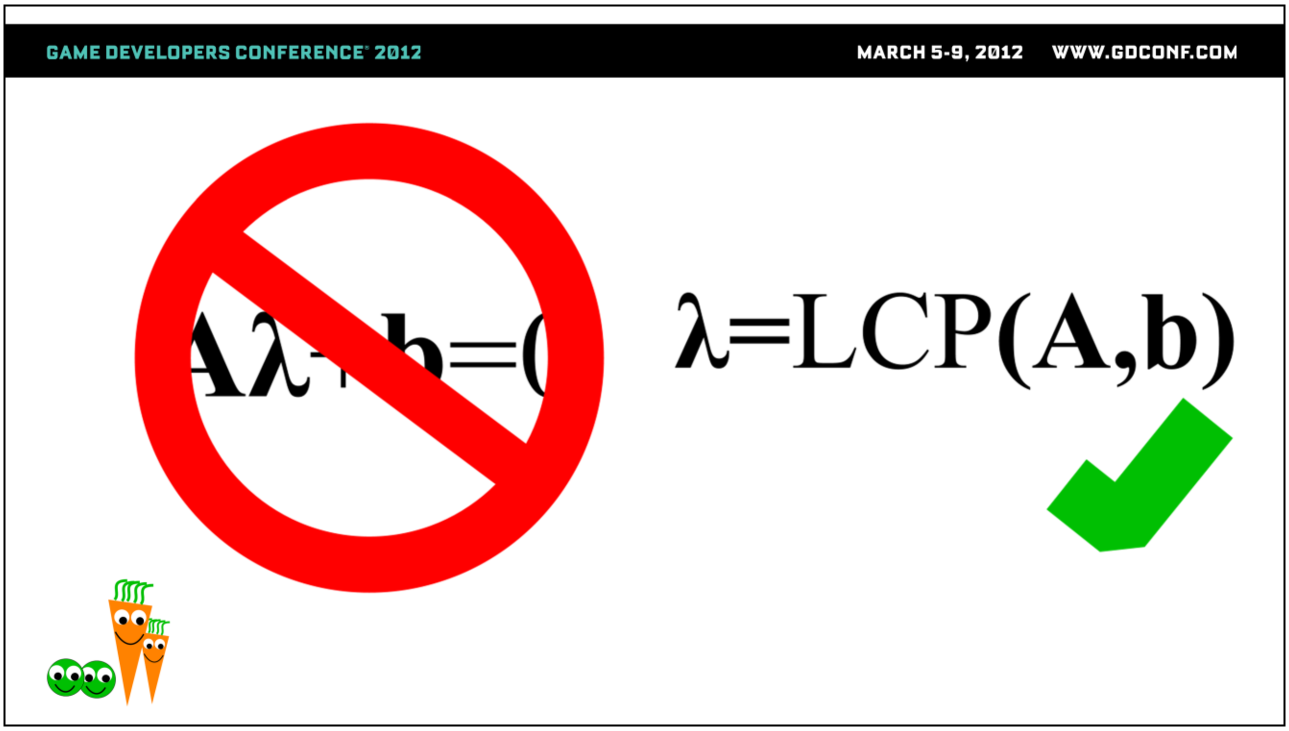
$$\mathbf{0} \le \lambda \;\perp\; \mathbf{0} \le \mathbf{J}\mathbf{M}^{-1}\mathbf{J}^{T}\lambda + \mathbf{J}\left(\mathbf{v}_{\text{old}} + h\mathbf{g}\right)$$

$$=$$

$$\lambda = LCP\left(\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^{T}, \mathbf{J}\left(\mathbf{v}_{\text{old}} + h\mathbf{g}\right)\right)$$

In the last slide we rearranged to move a bunch of stuff into the line that has the upside down T, the complementarity condition.

There is a shorthand way of writing this, the second line on this slide. So this is our LCP, or linear complementarity problem. It is linear because in this discretized model relative velocities are linear functions of the impulses applied at the points (lambda).



GAME DEVELOPERS CONFERENCE® 2012 MARCH 5-9, 2012 WWW.GDCONF.COM

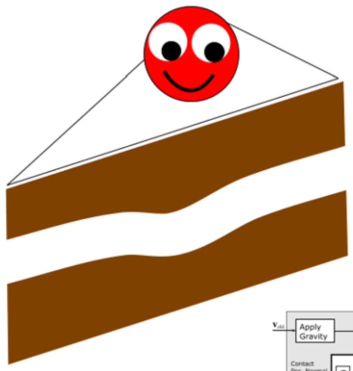
$\cancel{A\lambda+b=0}$

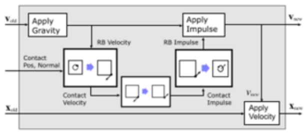
$\lambda = \mathrm{LCP}(A,b)$ ✓

So we went through all that so that I can say: what we are solving is not a linear system, it is an LCP.

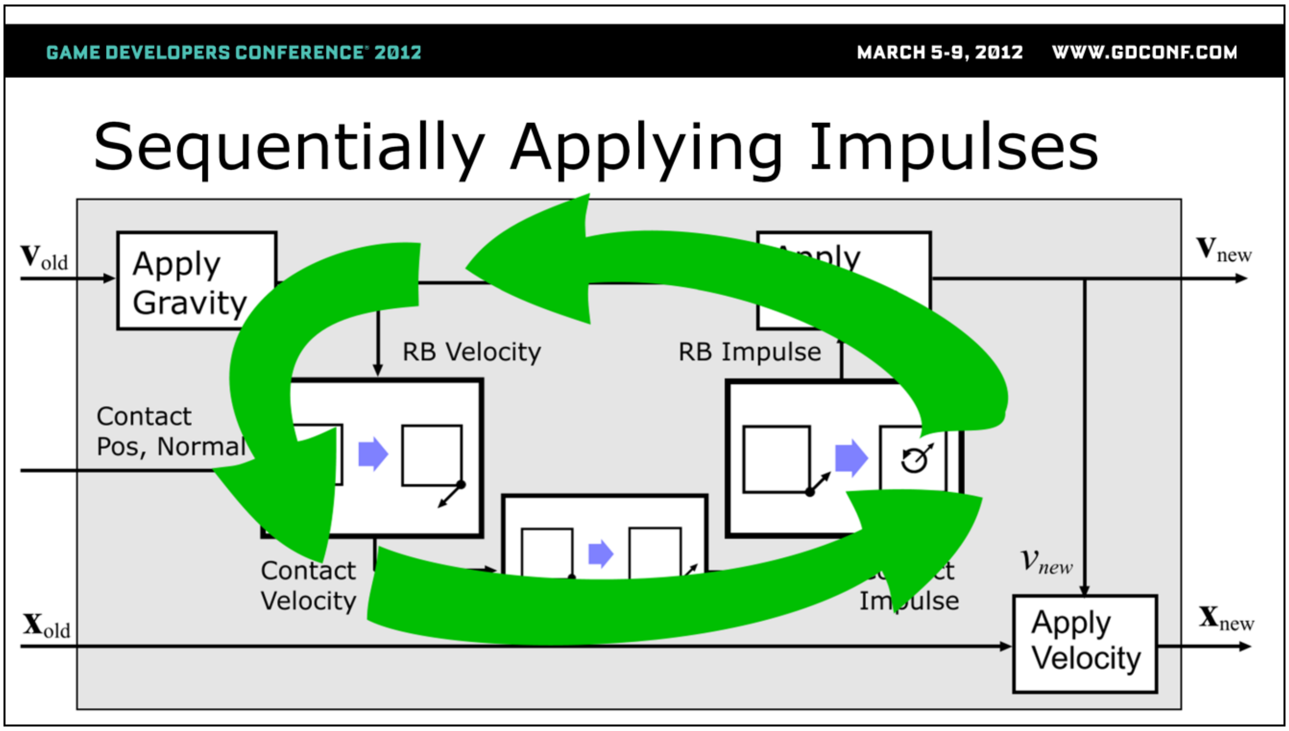
This unfortunately means that any existing linear system solvers that you might know about are not going to work.

Just Sequentially Apply Impulses



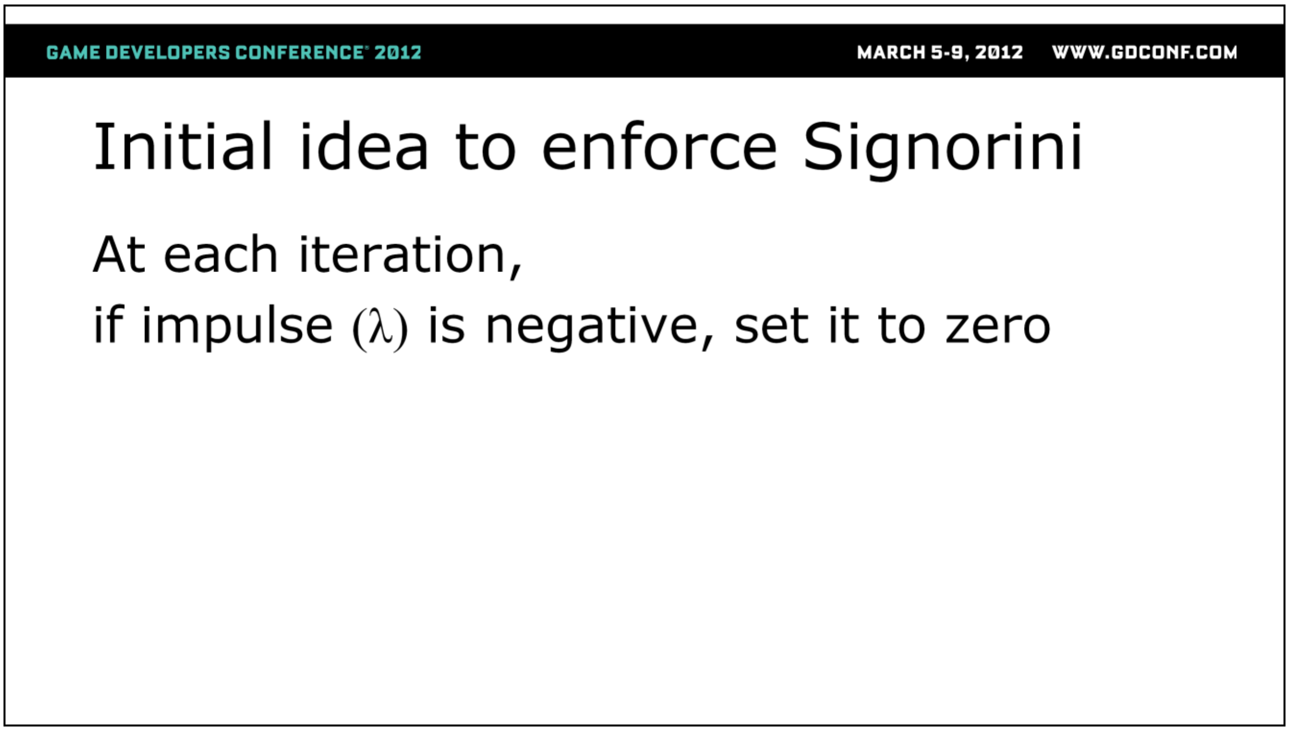


The good news though is that there is something that does solve this LCP model, and it is almost exactly the same as the simple one contact algorithm we talked about earlier.


GAME DEVELOPERS CONFERENCE 2012
MARCH 5-9, 2012 WWW.GDCONF.COM

Sequentially Applying Impulses

So all we do is apply the one contact algorithm to each contact in sequence, and then iterate through the whole contact list a small number of times. The default number of times in PhysX is four.

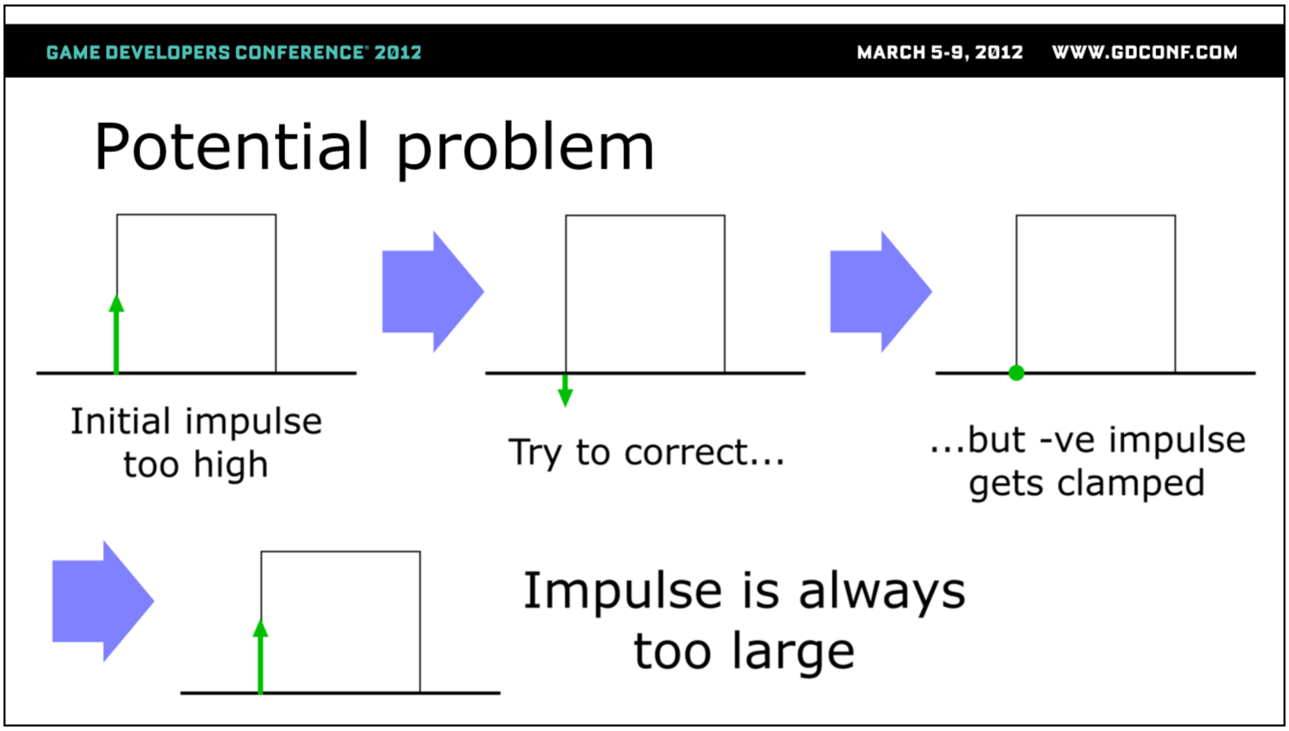


Initial idea to enforce Signorini

At each iteration,
if impulse (λ) is negative, set it to zero

The question though is how we ensure that the Signorini condition is met so that we can make sure that our objects don't all look like magnets.

The simplest thing you might think of is just take each impulse you apply at each iteration and set it to zero if it is negative. Remember that negative impulses are attractive impulses and positive impulses are repulsive impulses.



GAME DEVELOPERS CONFERENCE® 2012 MARCH 5-9, 2012 WWW.GDCONF.COM

Potential problem

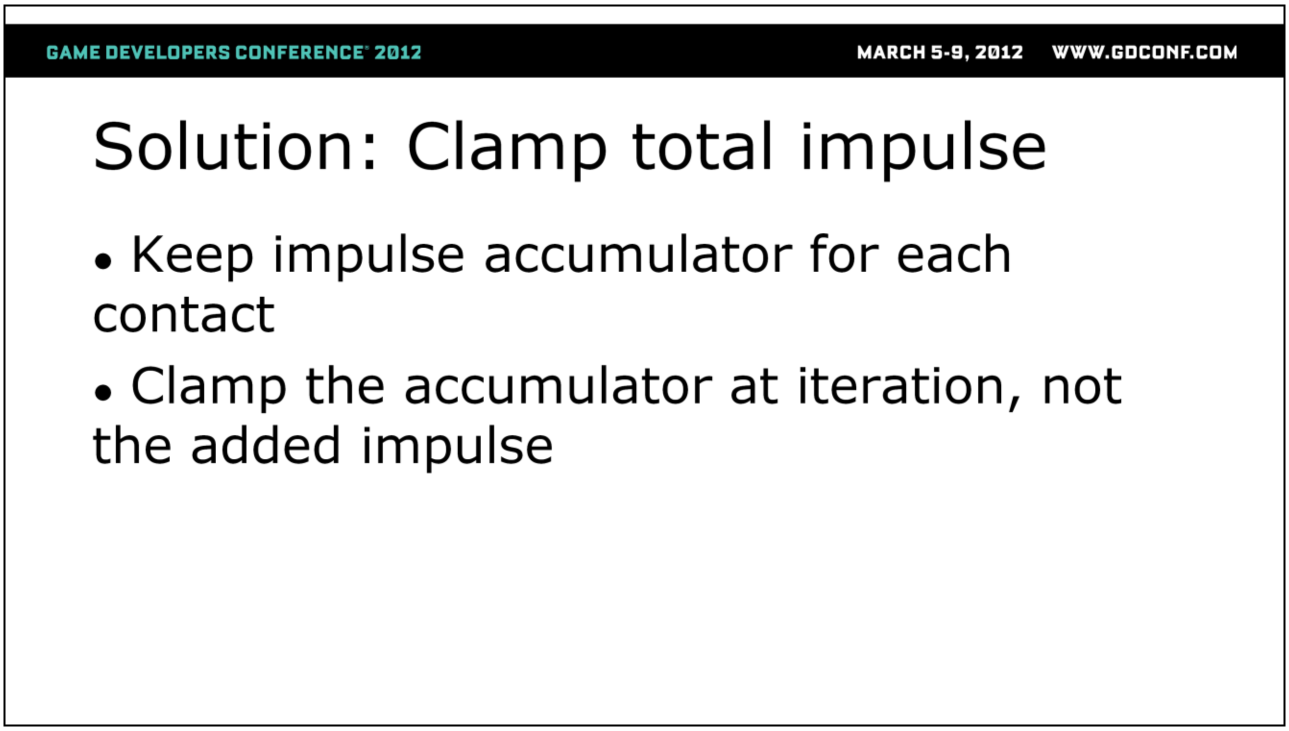
Ok, the problem is that this doesn't work, here is why.

We will need to iterate over all the contacts many times to converge to the correct solution. The default number of iterations in PhysX is four.

What the model tells us is that it is the total impulse applied in the frame that must obey the Signorini conditions, not the individual impulses.

These means that we need to keep an impulse accumulator for each contact and clamp that each frame, not the impulse from the current iteration.

Suppose that on the first iteration we apply too much impulse at a contact. If we clamped the impulse applied on each iteration, then we would never be able a negative correction to reduce the impulse that was too large.

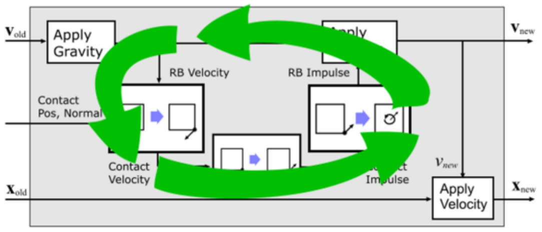

GAME DEVELOPERS CONFERENCE® 2012 MARCH 5-9, 2012 WWW.GDCONF.COM

Solution: Clamp total impulse

- Keep impulse accumulator for each contact
- Clamp the accumulator at iteration, not the added impulse

So instead, all we do is keep accumulators that track how much impulse was applied to each contact this frame and clamp those.

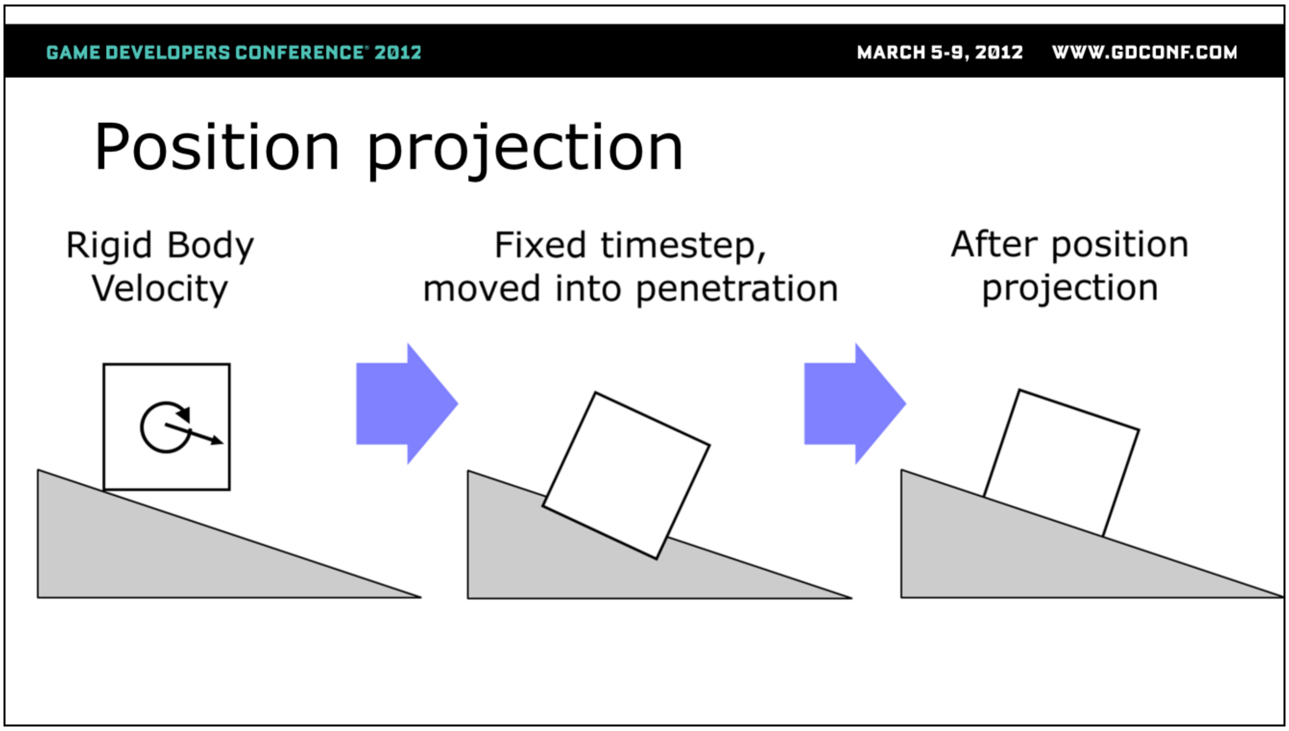
Multiple Contact Algorithm



=

```
solveMultiple($\mathbf{x}, \mathbf{v}, \mathbf{J}, \mathbf{M}, h, \mathbf{g}$)
{
  for $j$=0; $j$<iterCnt; $j$++
  {
    for $i$=0; $i$<contactCnt; $i$++
    {
      $t = \lambda_i$
      $m = \mathbf{J}_i \mathbf{M}^{-1} \mathbf{J}_i^T$
      $\boldsymbol{\lambda}_i = \lambda_i - (1/m)(\mathbf{J}\mathbf{v} + \mathbf{b})$
      $\boldsymbol{\lambda}_i = \max(0, \boldsymbol{\lambda}_i)$
      $\mathbf{v} = \mathbf{v} + \mathbf{M}^{-1}\mathbf{J}^T(\boldsymbol{\lambda}_i - t)$
    }
  }
}
$\mathbf{x} = \mathbf{x} + h\mathbf{v}$
```

So here is the final algorithm in code.



Position projection

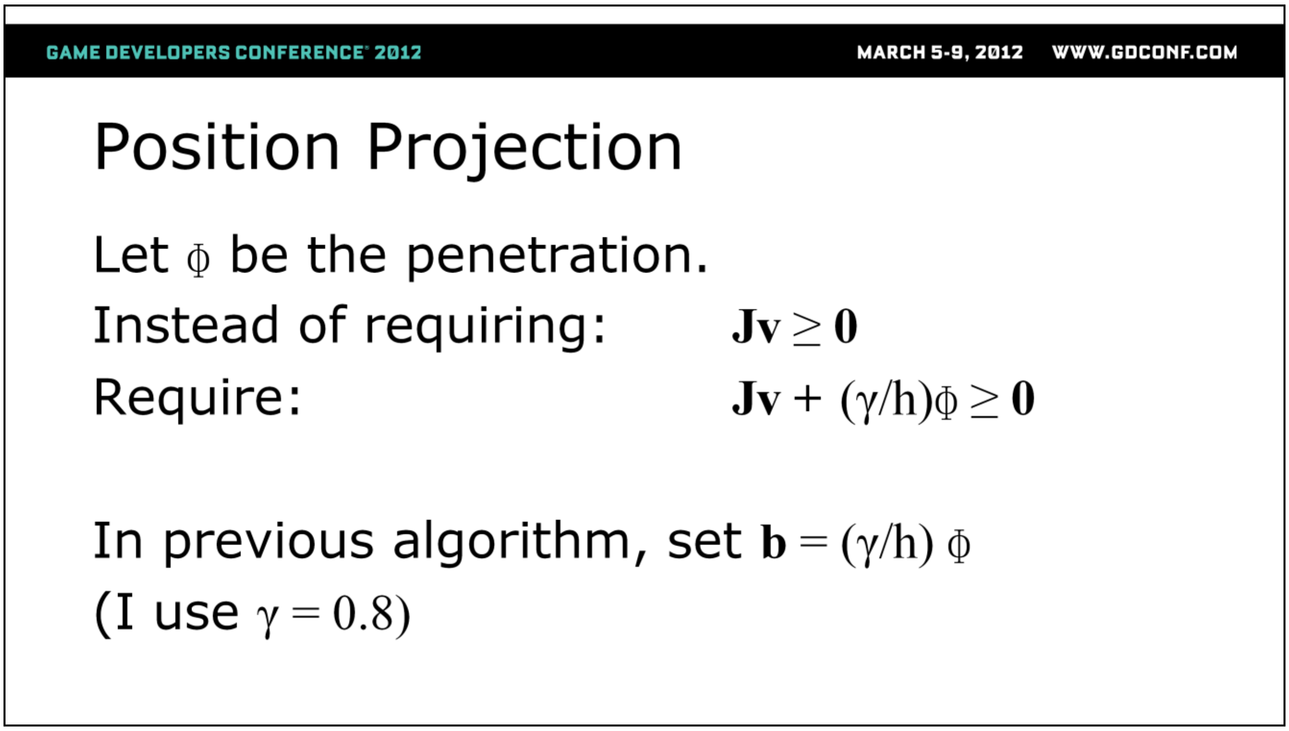
Rigid Body Velocity
Fixed timestep, moved into penetration
After position projection

Earlier I said that we could mainly think about using impulses to correct the velocities.

As our timestep size is fixed we can't completely ignore position errors though.

The middle diagram shows what might happen if we apply the corrected velocity with a fixed timestep. You can see here that there is both a linear position error and that the box has rotated too much.

So we need a way to pop the box out of the slope and rotate it to the correct orientation, as shown in the right hand diagram. This process is called position projection.



GAME DEVELOPERS CONFERENCE® 2012 MARCH 5-9, 2012 WWW.GDCONF.COM

Position Projection

Let ϕ be the penetration.

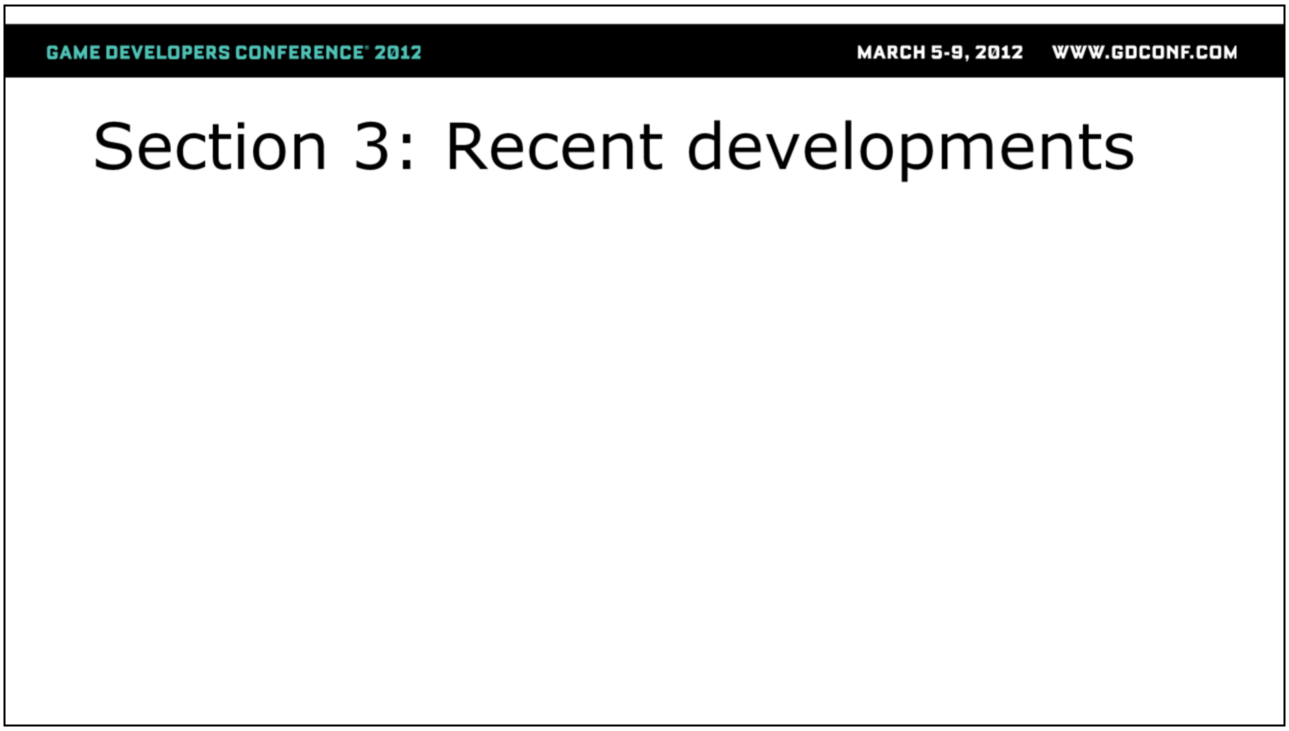
Instead of requiring: $\mathbf{Jv} \geq \mathbf{0}$

Require: $\mathbf{Jv} + (\gamma/h)\phi \geq \mathbf{0}$

In previous algorithm, set $\mathbf{b} = (\gamma/h)\,\phi$

(I use $\gamma = 0.8$)

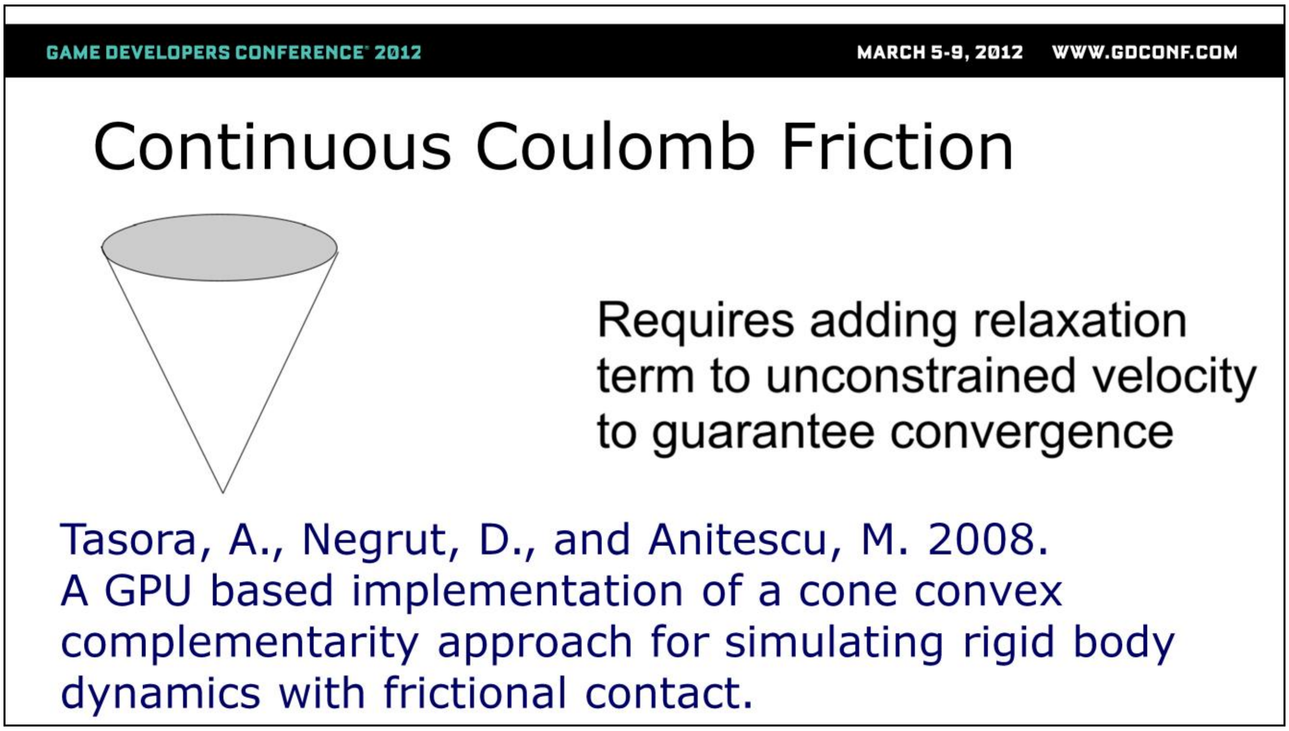
The collision detection tells us how much penetration has occurred, here we represent it with the letter Phi. We will decide to remove a certain proportion of this position error each frame. It is better to just remove a proportion of it each frame rather than all of it, because that will ensure that the correction happens smoothly and avoid one cause of jitter. PhysX is hard coded to remove 80% of the penetration each frame. Earlier we showed that Jv gives the realtive velocity that we want to zero (or allow to be positive). All we do is add 80% * Phi / h to this. Ok, that’s not exactly how we do it in PhysX, Erin’s previous talks cover other ways to do this.



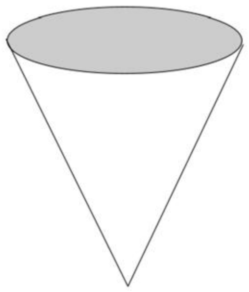
Section 3: Recent developments

That completes the description of the widely used PGS/SI algorithm for rigid body contact.

In the last section I'm going to share with you four extensions to it that I find interesting. I don't have much time left, but I've included a reference you can use to search the web for more information.



Continuous Coulomb Friction



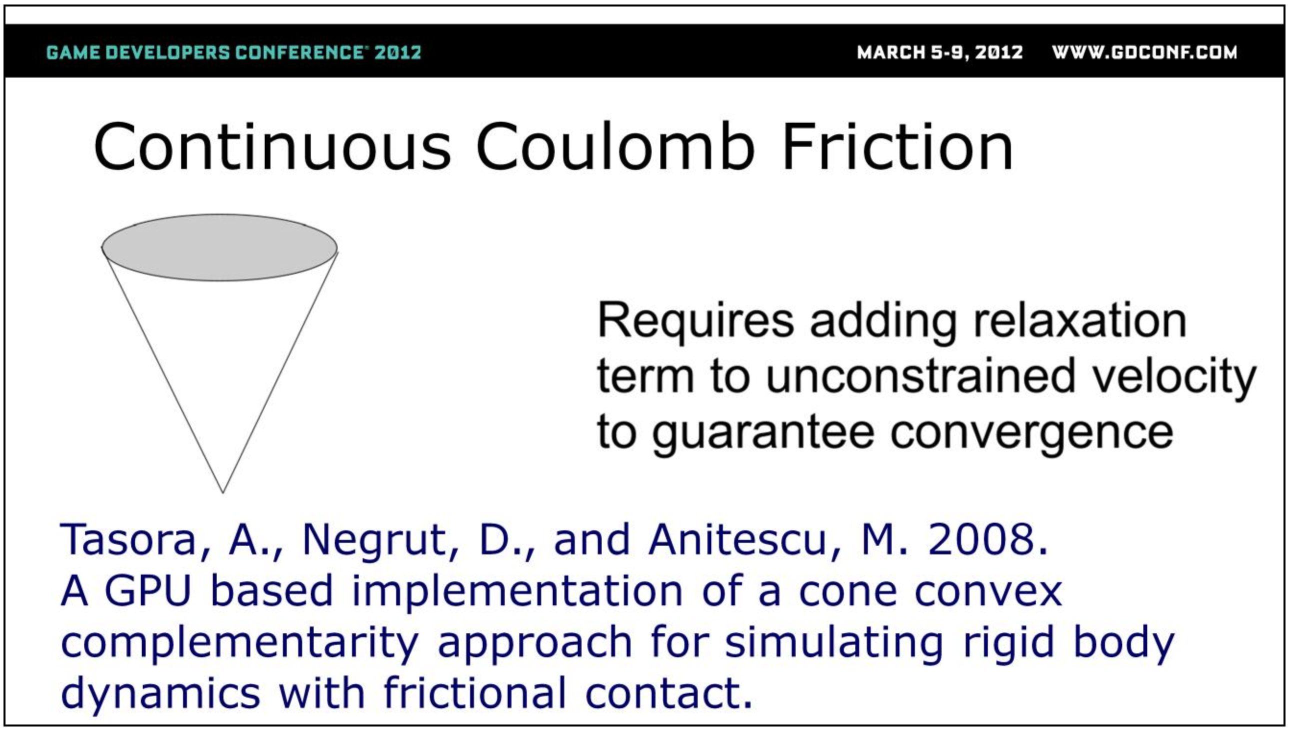
Requires adding relaxation term to unconstrained velocity to guarantee convergence

Tasora, A., Negrut, D., and Anitescu, M. 2008. A GPU based implementation of a cone convex complementarity approach for simulating rigid body dynamics with frictional contact.

I didn’t say anything about friction so far. A good friction model is that the more force an object is exerting on a surface, the harder you need to push it to get it moving across the surface. So heavy objects sitting on the ground are harder to push than light ones. Also, you push and push and the thing doesn’t move and you push a bit more and suddenly the thing starts moving. This is called the transition from sticking to sliding.

The standard model of this is Coulomb friction, and it is represented by a cone. The up axis is the force exerted on the surface, and the two other axes are the forces in the tangent plane of the contact. So in the coulomb model you calculate how much tangent force would be needed to stop motion in the plane. If the force is within the friction cone, then you apply the force and the object doesn’t move. So as you push and push and object, the force the surface is using to resist you is getting closer and closer to the surface of the cone. As soon as it gets to the surface of the cone, it is clamped so it can’t resist all your force and the object suddenly starts moving.

In the Coulomb friction model, the amount that the surface is allowed to resist you is the same whatever direction you try to push it in the plane, this is because the cross sections of the cone are smooth circles. Some things don’t behave like this, for example at the grocery store you sometimes get a cart with a dodgy wheel that is happy to move in some directions but not others.



GAME DEVELOPERS CONFERENCE® 2012 MARCH 5-9, 2012 WWW.GDCONF.COM

Continuous Coulomb Friction

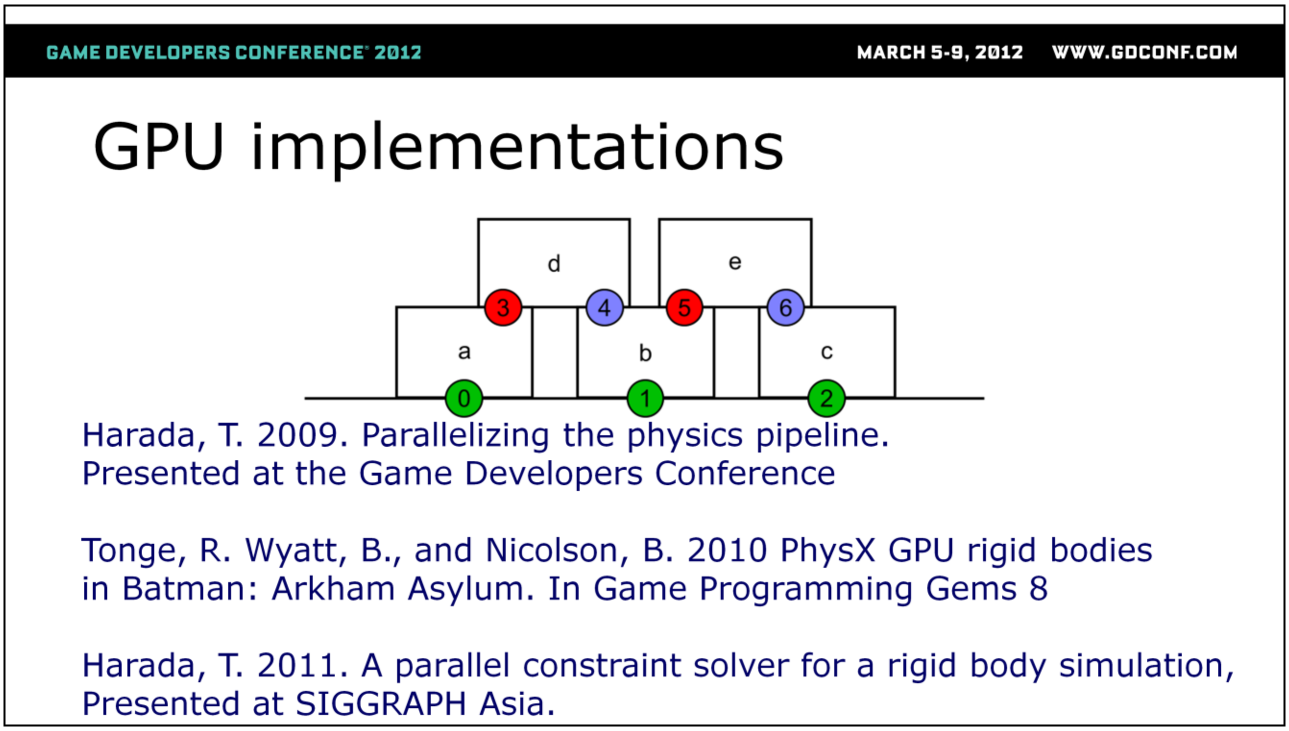
Requires adding relaxation term to unconstrained velocity to guarantee convergence

Tasora, A., Negrut, D., and Anitescu, M. 2008. A GPU based implementation of a cone convex complementarity approach for simulating rigid body dynamics with frictional contact.

In the Coulomb friction model, the amount that the surface is allowed to resist you is the same whatever direction you try to push it in the plane, this is because the cross sections of the cone are smooth circles. Some things don't behave like this, for example at the grocery store you sometimes get a cart with a dodgy wheel that is happy to move in some directions but not others.

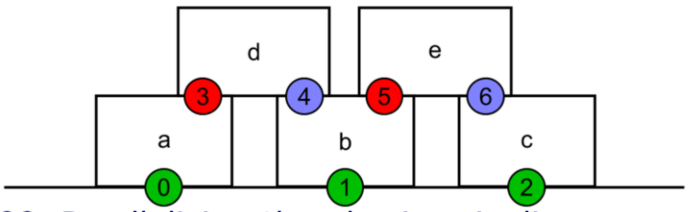
Anyway, although the smooth coulomb cone is desirable, until recently we had to discretize it into a polyhedral cone or a box to guarantee that the solver would converge. So video game solvers behaved a bit more like grocery store carts than we would like.

So the simplest way you might try to implement smooth Coulomb friction is to calculate the tangent force needed to zero the velocity in the plane each iteration and just clamp (project) it to the smooth cone. I tried this myself many years ago, and found situations where it caused the solver to not converge. In 2008, Anitescu and his collaborators worked out that you could make this simple method work just by adding a small relaxation term in the same place we added the position projection term. It does give a small unwanted vertical motion, but you can make this as small as you like by modifying the parameters.



GAME DEVELOPERS CONFERENCE® 2012 MARCH 5-9, 2012 WWW.GDCONF.COM

GPU implementations



Harada, T. 2009. Parallelizing the physics pipeline. Presented at the Game Developers Conference

Tonge, R. Wyatt, B., and Nicolson, B. 2010 PhysX GPU rigid bodies in Batman: Arkham Asylum. In Game Programming Gems 8

Harada, T. 2011. A parallel constraint solver for a rigid body simulation, Presented at SIGGRAPH Asia.

At the moment both NVIDIA and AMD are interested in making all this run on a GPU, and that's what I spend all my days doing. It is GPU computation that enabled the fracture demos presented at the start. Takahiro Harada from AMD is going to talk more about GPU solvers this afternoon.

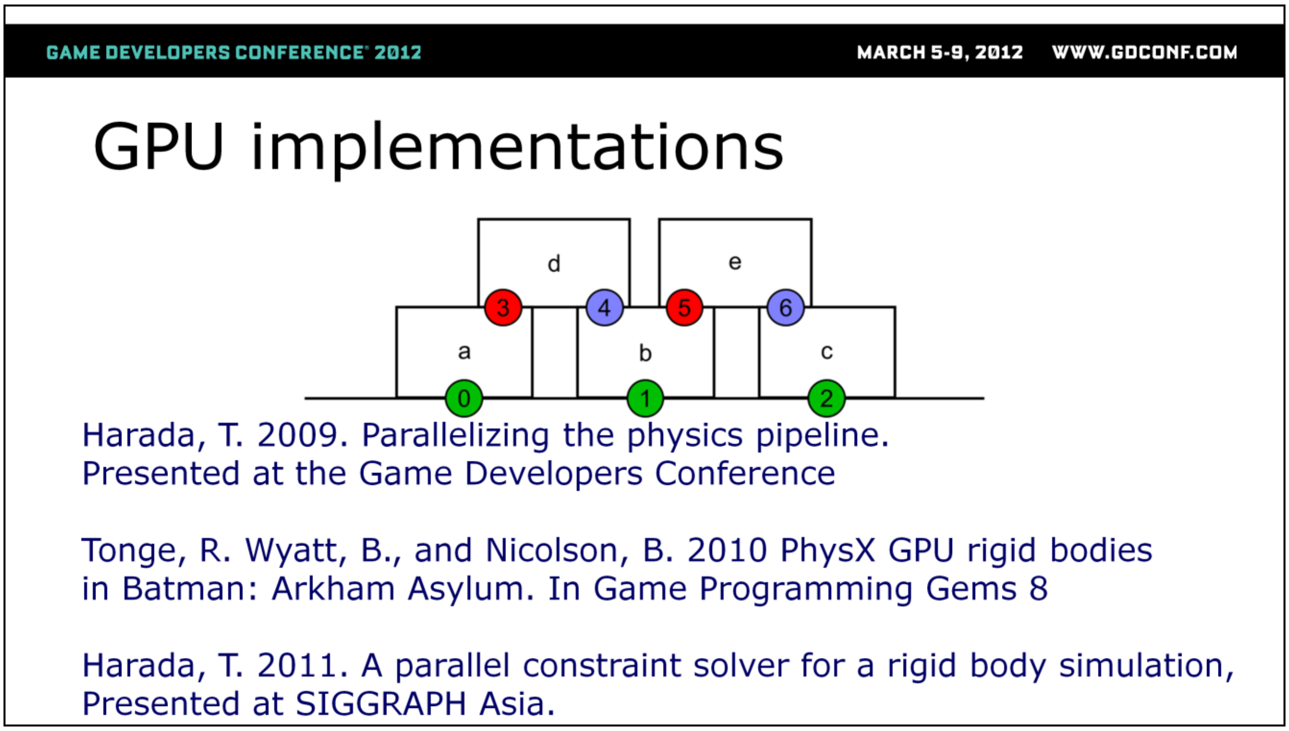
Here are some references from both companies that describe some of the work done.

The diagram here shows one of the fundamental problems of moving the algorithm described in this talk directly to the GPU.

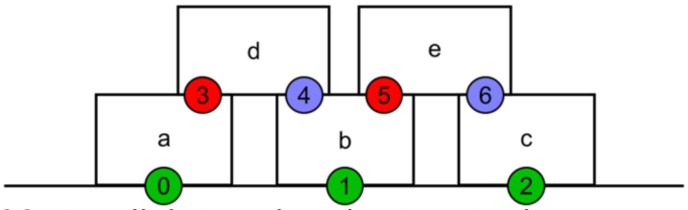
It shows 5 boxes sitting on the ground in a truncated pyramid kind of stack. There are 7 contacts, labelled with numbers.

Ideally we'd like to use 7 threads on the GPU to calculate the impulses in parallel. Unfortnately, if we do this then some velocity updates will get lost.

For example, thread (contact) three and thread four will both try to update body d, and only one of their writes can win. If updates are lost like this then the algorithm generally won't converge.



GPU implementations



Harada, T. 2009. Parallelizing the physics pipeline. Presented at the Game Developers Conference

Tonge, R. Wyatt, B., and Nicolson, B. 2010 PhysX GPU rigid bodies in Batman: Arkham Asylum. In Game Programming Gems 8

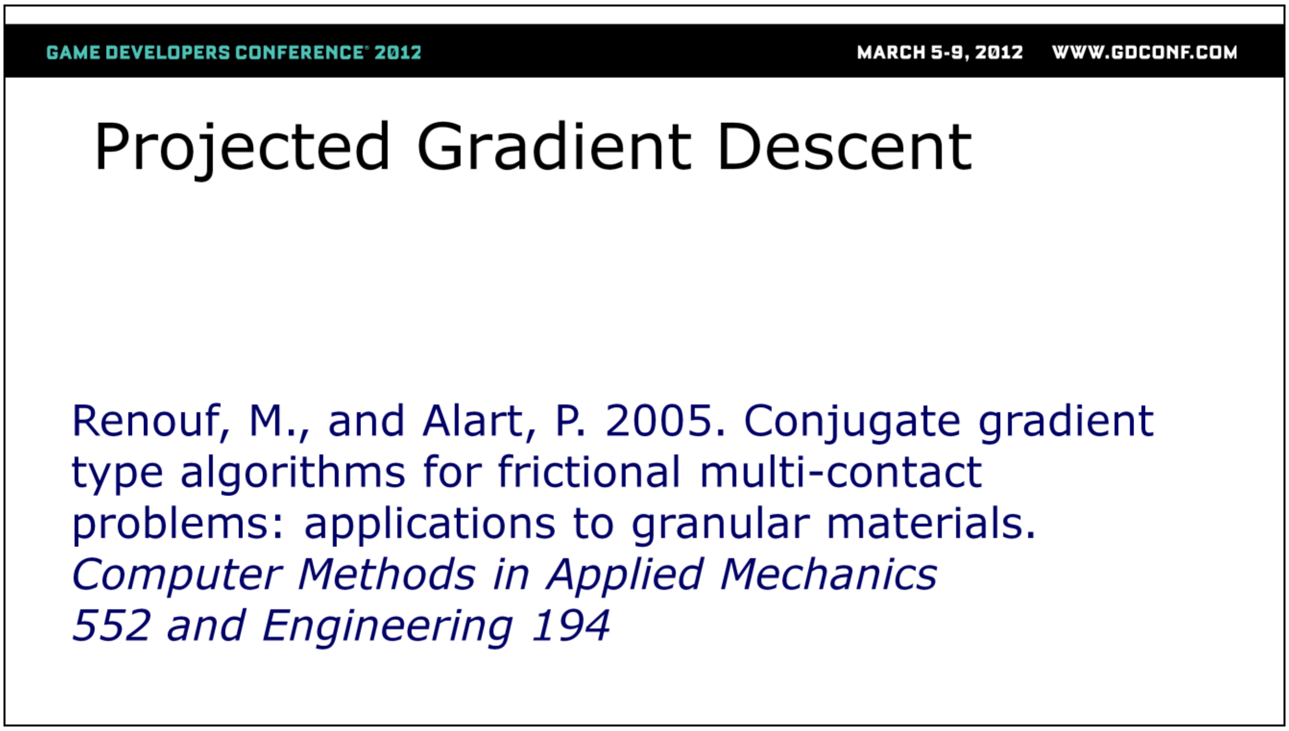
Harada, T. 2011. A parallel constraint solver for a rigid body simulation, Presented at SIGGRAPH Asia.

So what we have to do is to “color” the contacts into groups (colors) for which no body is affect by more than one contact in each color.

We then can process the colors sequentially, but process the contacts within each color in parallel.

The problem with that is that there is much less parallelism available. In this example we can use at most 3 threads in parallel, when we really want to use 7. The total time is now going to be multiplied by the number of colors. Also, in larger examples you find that the first few colors have quite a large proportion of the constraints, but after that there are lots of colors with hardly any constraints. This means that the GPU is mostly idle for almost all of the colors.

Next we’ll look at a couple of possible solutions to this.



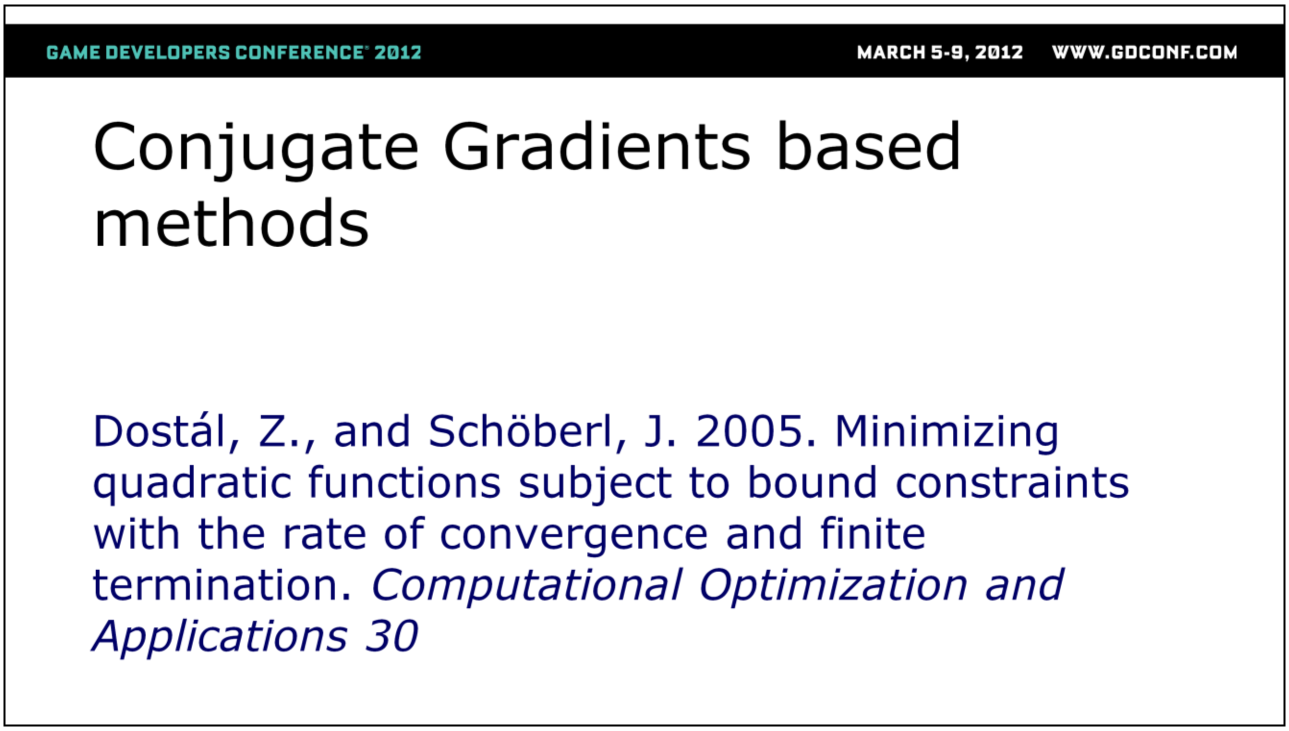
GAME DEVELOPERS CONFERENCE® 2012 MARCH 5-9, 2012 WWW.GDCONF.COM

Projected Gradient Descent

Renouf, M., and Alart, P. 2005. Conjugate gradient type algorithms for frictional multi-contact problems: applications to granular materials. *Computer Methods in Applied Mechanics 552 and Engineering 194*

We could apply impulse at all contacts at all iterations, using the maximum number of threads, just averaging contributions of impulses for shared bodies. If we try to apply the maximum relative velocity killing impulse at each contact (like we did in the CPU algorithm earlier) then the contacts will just fight each other and the algorithm will either converge slowly or not at all.

The paper referenced here contains an algorithm called Projected Gradient Descent (equation 10 in the paper) that shows how to calculate the maximum safe impulse to apply at each contact to ensure convergence. Convergence is still pretty slow though.



Conjugate Gradients based methods

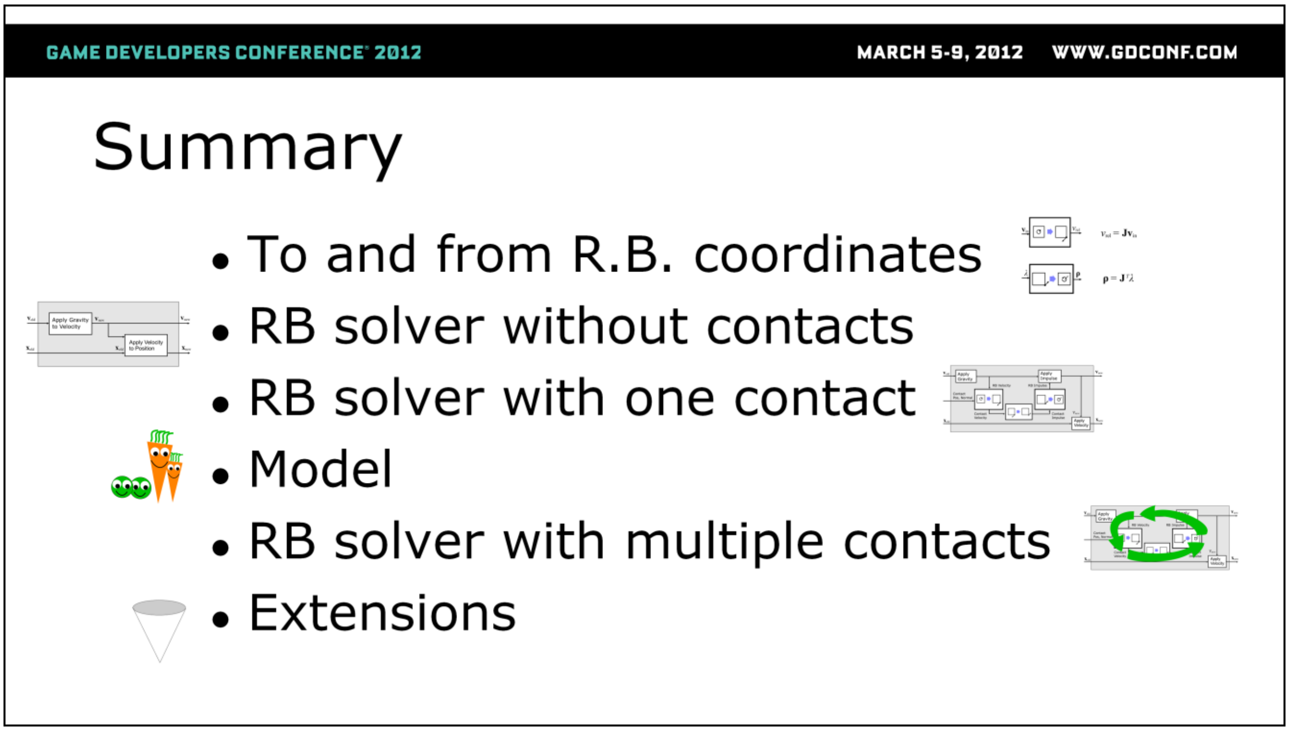
Dostál, Z., and Schöberl, J. 2005. Minimizing quadratic functions subject to bound constraints with the rate of convergence and finite termination. *Computational Optimization and Applications 30*

I described earlier the difference between a linear systems and an LCP.

You may have noticed that the CPU LCP solver described earlier is similar to a linear system solver called Gauss-Seidel. In fact the algorithm I presented is called Projected Gauss Seidel. So you might be wondering if we can take other popular linear system solvers (like conjugate gradients) and turn them into LCP solvers by adding projection (clamping) to them.

Unfortunately, with solvers like Conjugate Gradients it is not as easy to do as with Gauss-Seidel (while maintaining convergence). Ideally we want a method like Conjugate Gradients that solves our symmetric LCP that has a proof that it will always converge. The reason we need a proof is because in a game anything can happen, and we don’t want to keep adding hacks to our solver to make it converge as the artists add more things to the game or as the player finds new things to do.

There are a few projected Conjugate Gradient algorithms out there, but this one by Dostal is the first one that I’ve seen that has a convergence proof.



Summary

- To and from R.B. coordinates
- RB solver without contacts
- RB solver with one contact
- Model
- RB solver with multiple contacts
- Extensions

Ok, so what did I talk about today?

First I showed how to convert velocities and impulses to and from rigid body coordinates.

then I showed a simple RB solver for situations without contacts,

then I showed how to do one contact,

then I did a brief detour into what the model is,

then I showed how to do multiple contacts in a way that was almost the same as the method for one contact,

then I showed some extensions that I find interesting.

Thanks to Erin Catto, Erwin Coumans, the NVIDIA PhysX guys, the MathEngine guys and all the other people who worked on this stuff with (or, erm, alongside) us.

Questions?

rtonge@nvidia.com

Ta very much.