

Iterative Rigid Body Solvers

Richard Tonge

Principal Software Engineer, NVIDIA



GAME DEVELOPERS CONFERENCE
SAN FRANCISCO, CA
MARCH 25-29, 2013
EXPO DATES: MARCH 27-29

2013

Arena destruction demo



Welcome to the rigid body solver presentation. First I'll show you a couple of demos so that you can see the results you can get with rigid body dynamics. This demo shows rigid body destruction of an arena made out of stone. As I fire the weapon, you'll notice the stone fracture, the debris fall, and form stable piles on the ground. The physics and graphics are all running on a single GPU.



Demo credits

Matthias Müller-Fischer
Nuttapong Chentanez
Tae-Yong Kim
Aron Zoellner
Kevin Newkirk

The fracture tech and demo were made by these researchers and artists at NVIDIA.

Hawken Demo

In the next demo you will see rigid body destruction implemented in a real game, Hawken by Adhesive games. Again the physics and graphics are running on the same GPU.

SIGGRAPH paper

Mass Splitting for Jitter-Free Parallel Rigid Body Simulation

Richard Tonge, Feodor Benevolenski, Andrey Voroshilov

Mass Splitting for Jitter-Free Parallel Rigid Body Simulation

Richard Tonge
NVIDIA
Feodor Benevolenski
NVIDIA
Andrey Voroshilov
NVIDIA



Figure 1. Left: Volcanoes with ability selection coming out of a rock cone straight out without jitter introduced on NVIDIA GEFORCE or intel HD GPU. Middle: Metal grating used as a road or floor pressure committee. The doors have irregular shapes and large mass moves. Right: The simulation is an animated stage with buildings, furniture, and objects.

Abstract

We present a parallel simulation rigid body solver that results from splitting the mass of each rigid body into smaller components. In large or real-time simulations, the jitter introduced by other numerical methods can become a significant problem. We propose a method to reduce this jitter by splitting the mass of each rigid body into smaller components. Projected Gauss-Hankel (PGH) distributes the resultant world rigid bodies into parallel threads. The challenge in order to parallel implementation is very challenging. In order, there are many constraints that must be considered to ensure convergence. We accomplish projected health by dividing each body into smaller components. We also use a local update approach for the body. We use the same contact types as the original PGH, and we use the same collision detection, providing well-timed and accurate collision detection. We also use the same physics engine as the original PGH, which is the same as the original PGH, which can simulate up to 1000 objects with no visible jitter or artifacts.

CC Categories: I.3.1 Computer Graphics/Computer Geometry and Object Modeling—Physical based modeling, I.3.5 [Mathematics and Statistics], Mathematics, Numerical methods, Numerical analysis.

Keywords: rigid bodies, non-smooth dynamics, contact, friction

Links: [DOI](#) | [PDF](#) | [A mail to: \[rgtonge@nvidia.com\]\(mailto:rgtonge@nvidia.com\)](#)

In previous years that we've done this talk, we've had a wide range of abilities in the audience.

I know that some of you write physics solvers for a living, and may be interested in the details, and I refer you to our SIGGRAPH paper.

Contents

- What do we need rigid bodies to do?
- Single core solver
- Parallel solver
- Results

The presentation you are about to see has four sections.

First, what do you need the rigid bodies in your games to do?

Second, I'll show you how you can write a solver to get these behaviors and avoid these problems. For beginners, I'll start from how to apply a force to a rigid body and go from there.

Third, for those of you that write games physics solvers for a living, I want to give you something too. So I'll also show you how to fix some of the problems unique to parallel solvers.

Finally, I'll show you the benefits of solving these problems, and the effects you can get by running large simulations on many threads.

I'll try and leave 10 minutes at the end for questions, so I ask that you wait until then to ask them.

Section 1

What are rigid bodies supposed to do?



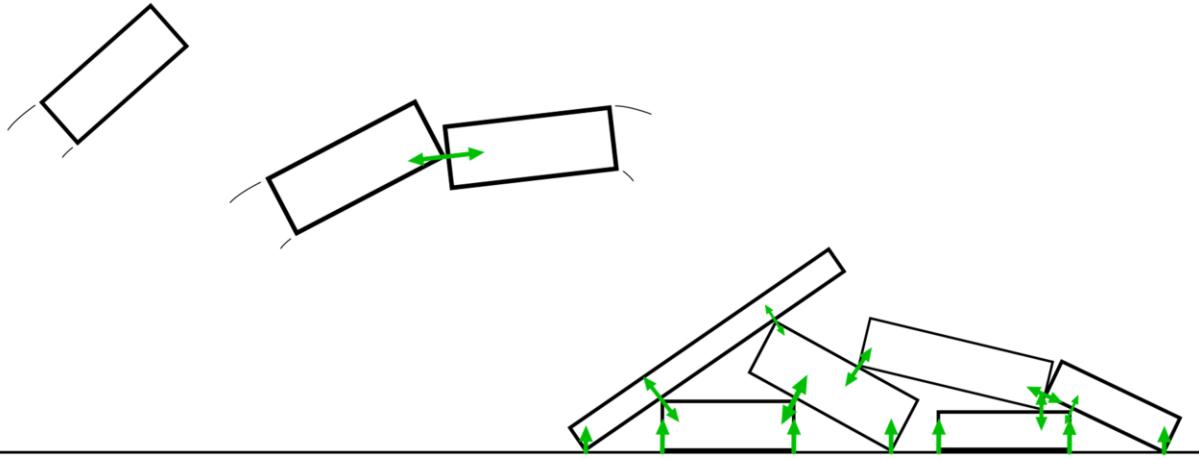
Section 1: Problem/Requirements

Games have lots of rigid objects, like the player, vehicles, non player characters, and the static environment.

What do you need rigid bodies to do?

- Accelerate and decelerate. Even early games had this.
- Appear to be solid, in other words, not go through each other.
- Fall under gravity.
- Slide down slopes.
- Stop sliding. We need to simulate surface properties like roughness, or friction. If we don't do this, then everything will look like it is sliding around on ice.
- Bounce like basketballs, or not bounce like concrete rubble.
- Form piles. This is just a consequence of objects not going through each other and falling under gravity, but it is surprisingly hard to get right, so it gets its own mention.

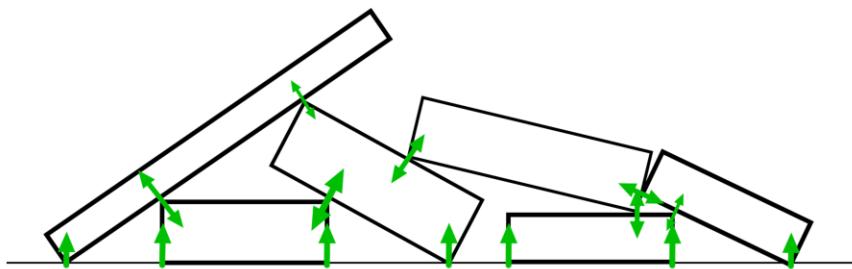
The picture on this slide shows a demo that we did a couple of years where you could wander around an art gallery making large piles of debris by blowing things up with a rocket launcher.



Cast your mind back to high school physics class. You probably learned how to calculate the motion of projectiles, like the one on the left. You probably also learned how to calculate and apply a collision impulse of two bodies in mid-air, like the diagram in the middle. The green arrows show the forces involved.

Cases like the diagram on the right are harder. This shows a pile of objects coming to rest on the ground. You can see that there are many forces involved, and that they all depend on one another.

Stable Piles = Balanced forces



The motion of each body is determined by the sum of all the forces acting on it.

If forces don't exactly sum to zero for each body, then the pile will never come to rest, and instead will jitter.

Contacts affect each other



Finding a set of forces that will eventually sum to zero is hard, because applying a force or impulse at one point on a body immediately affects the velocities at other points.

It is complicated further if you want to solve different forces on different threads.

This is why you might need a solver like the one I'm going to describe today.

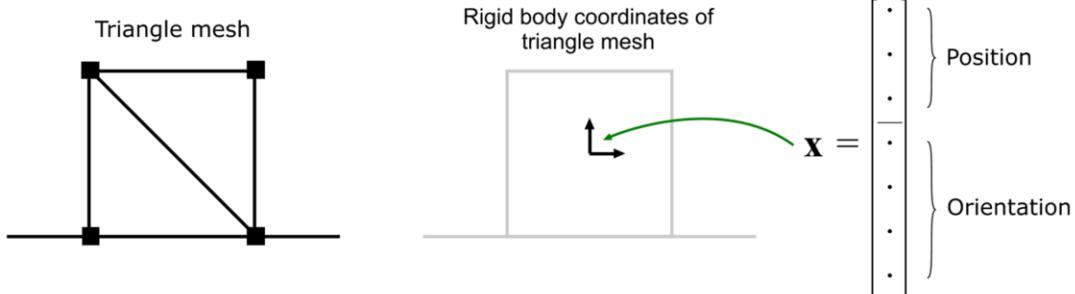
Section 2

How to make rigid bodies do the right thing
using a solver (single core)

This section is the introductory section, and will be familiar to those of you who were here last year.

Section three is where the new material starts.

Rigid Body Basics



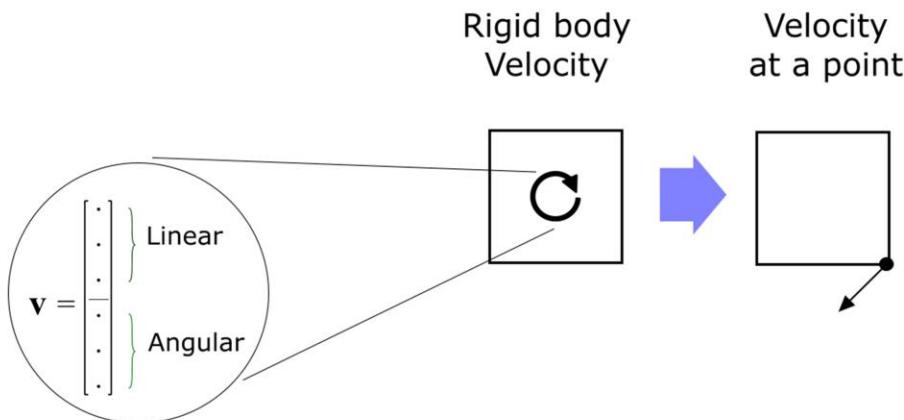
Definition of rigid body coordinates

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.

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.

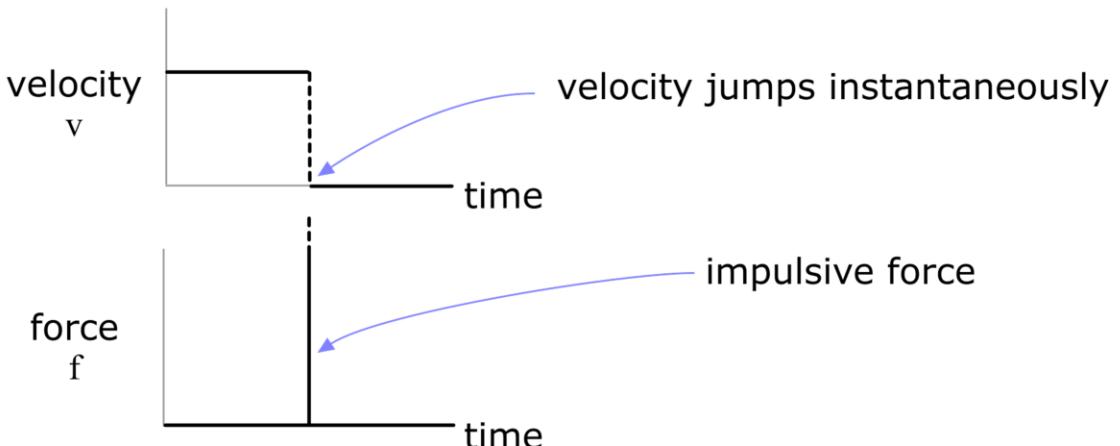
Use impulses and velocities



I recommend that you think of rigid body physics in terms of impulses and velocities, rather than forces and accelerations.

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

What is an impulse?



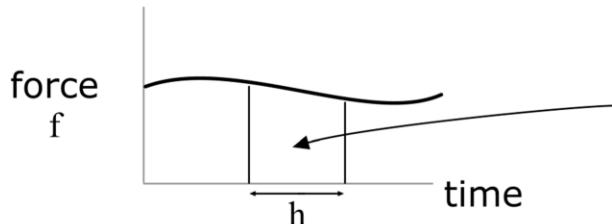
You'll hear the word impulse used in two ways, the first is in the term "impulsive force".

Imagine a car travelling at constant speed towards a concrete wall. Once the car hits the concrete wall its speed will go to zero very quickly. In the inelastic rigid body model this happens instantly. The graph at the top of the slide shows how the velocity of such a car changes over time.

Underneath that graph is the corresponding graph of the force between the wall and the car on the same time axis. You can see that the force is zero almost everywhere, except at this very short period of time where the wall is reducing the car's velocity, where it is very high.

Such forces that are applied over infinitesimally small time periods are called impulsive forces.

What is an impulse?



impulse is area under
force-time graph

(for constant force, $I = hf$)

The second way I'll use the term impulse is the area under a force-time graph between two points in time, the integral of force with respect to time.

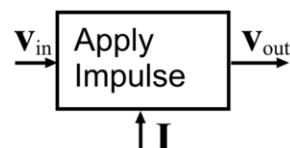
In your simulations you need to know the state of the system at regular intervals in time, the times in which you render a frame of graphics.

In the force-acceleration model, you'd calculate forces to apply only at these instants of time (assuming that you don't subdivide the time step).

In the impulse-velocity model, you instead solve to find the area under the force-time graph between frames. This allows you to calculate systems that have impulsive forces between frames, and treat resting contact and colliding contact in the same way. It also ensures that certain common frictional contact situations always have a solution. (see Baraff D. "Issues in computing contact forces for non-penetrating rigid bodies" for details of this common frictional contact situation).

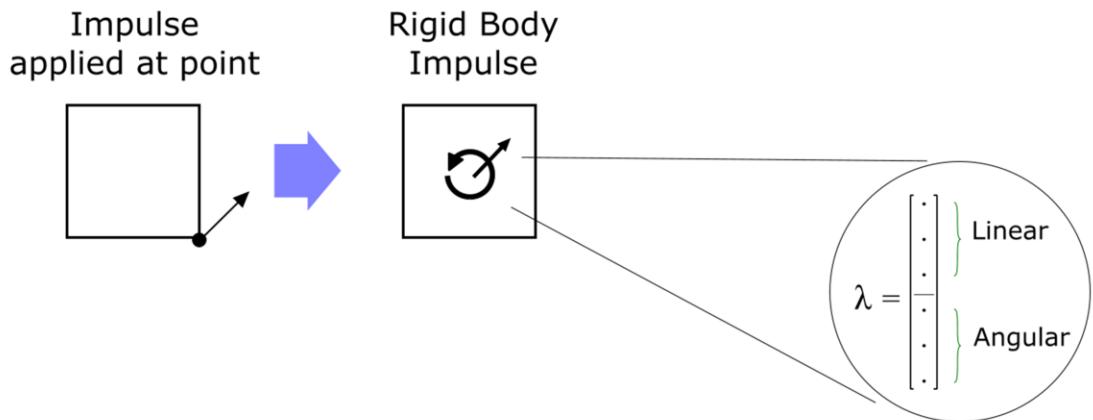
If you want to apply a constant force, you can easily convert it to an impulse by multiplying by the time step, h . Most of the rules about applying forces applies to impulses, for example, impulses occur in equal and opposite pairs, just like forces.

Impulse application



The boxes on this slide show that applying a force changes acceleration, and applying an impulse changes velocities.

Impulses



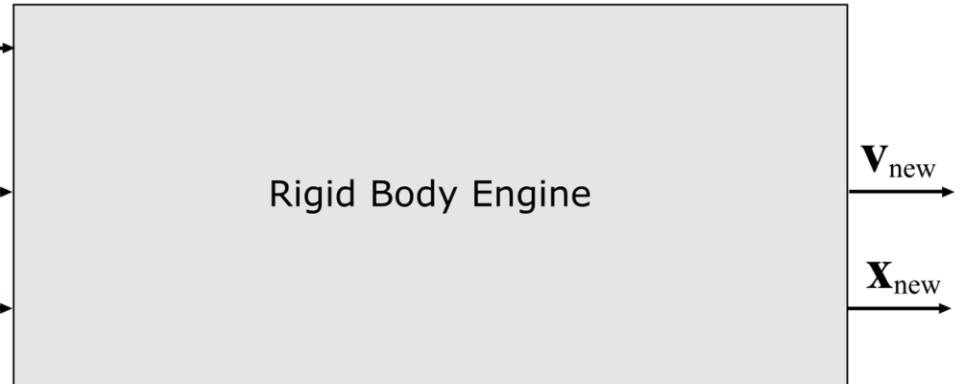
Earlier I showed you how you can track quantities like position and velocity at only one point, the center of mass.

If you want to apply an impulse to a vertex (or other point on the rigid body), you can calculate the equivalent rigid body impulse (a linear and angular impulse at the center of mass) and apply the impulse by changing the rigid body velocity.

Making a rigid body solver

Contacts
From Collision
Detection

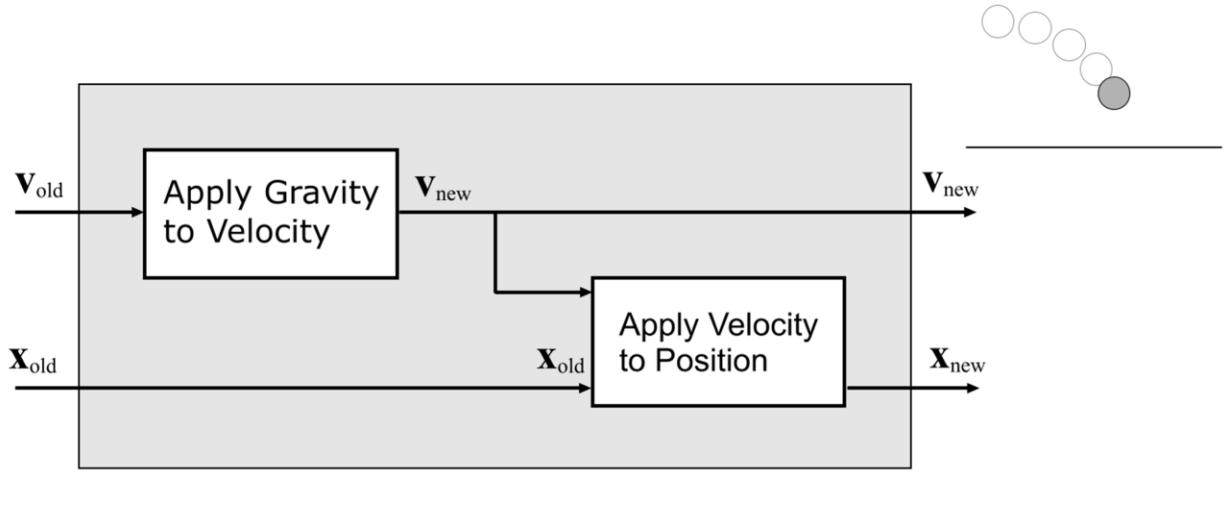
Rigid Body
Coordinates



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 you'll see the diagram become more detailed.

Moving a Body Without Collisions

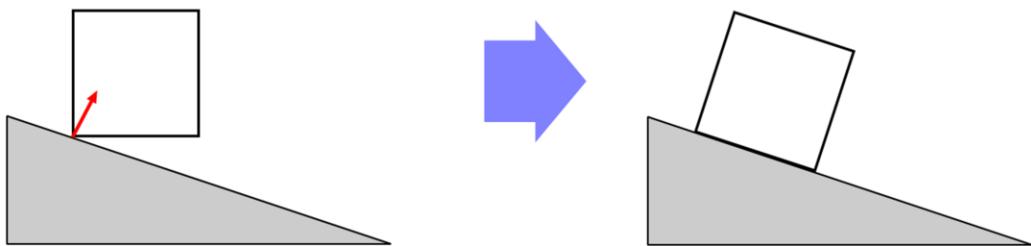


The simplest rigid body physics engine you could write would just move a single body through the air without collisions.

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

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

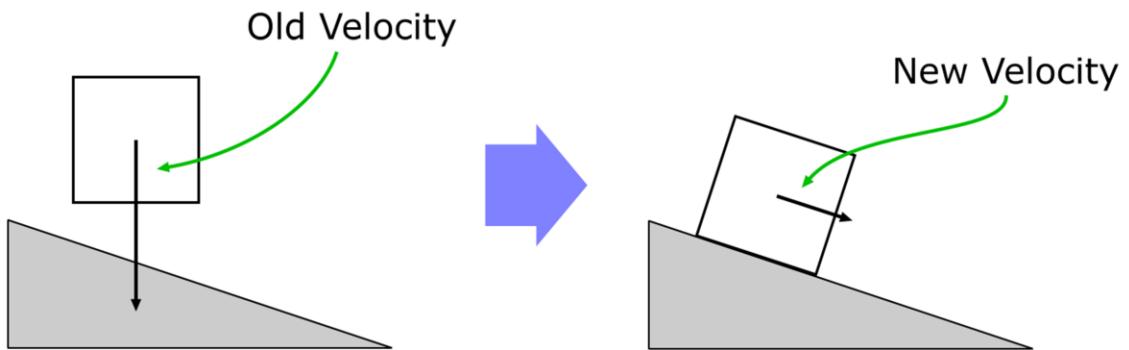
Adding a single contact



The next simplest simulation you could try 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 you want to happen. To keep things simple we're going to look at an inelastic contact, so imagine that the box and slope are so rigid that the box won't bounce when it hits the slope.

Contact at the velocity level

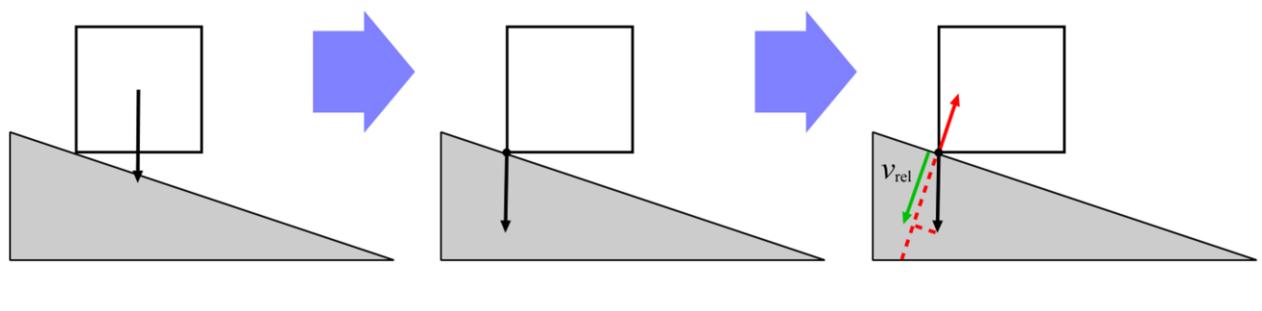


You can prevent bodies penetrating each other by applying impulses to change their velocities.

When the box hits the slope, you can 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.

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

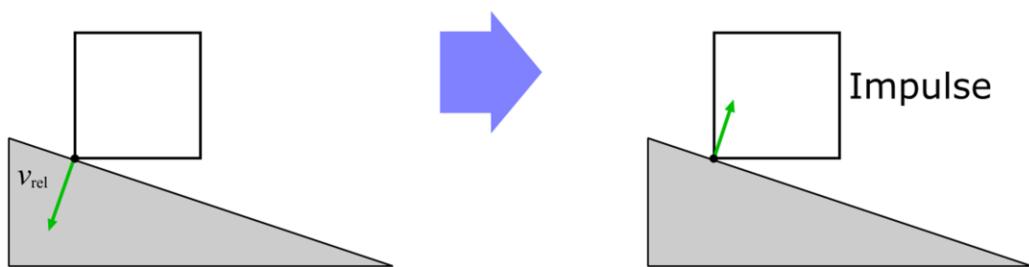
Velocity at the contact



You want the new velocity to 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. I'll show you 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.

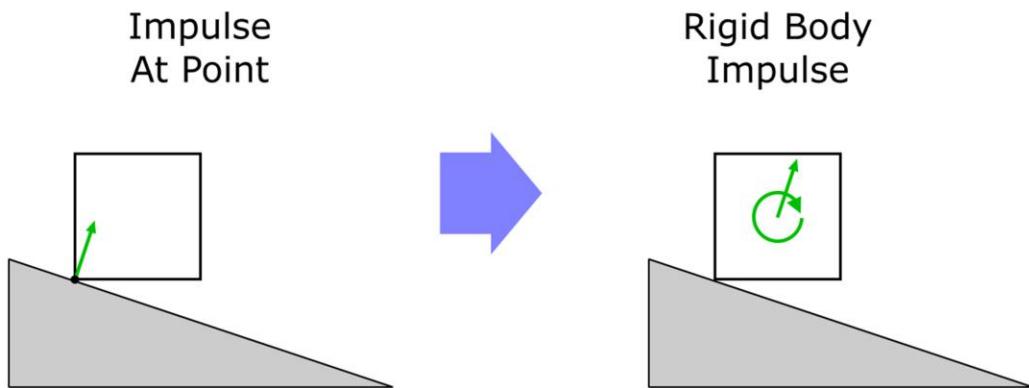
You want to eliminate the component of the velocity that is pulling the box into penetration, so first you need to calculate the magnitude of this velocity component.

Calculating the impulse



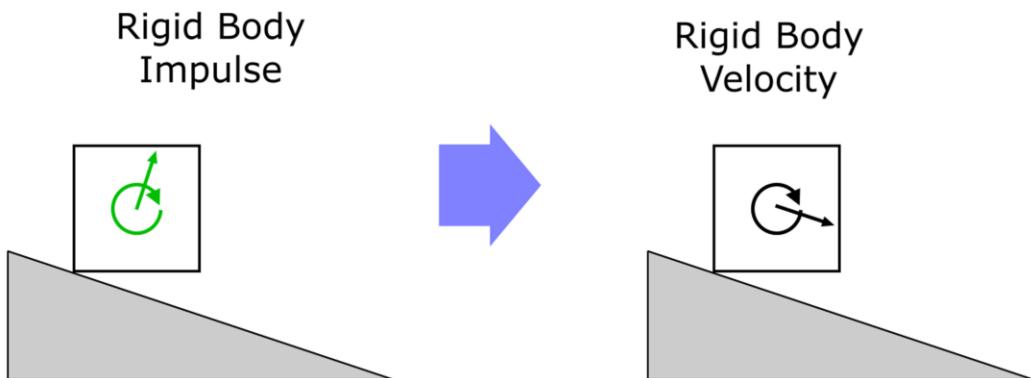
Once you know the direction and magnitude of the velocity component you want to eliminate, you can calculate the impulse required to eliminate it.

Converting impulse to RB coords



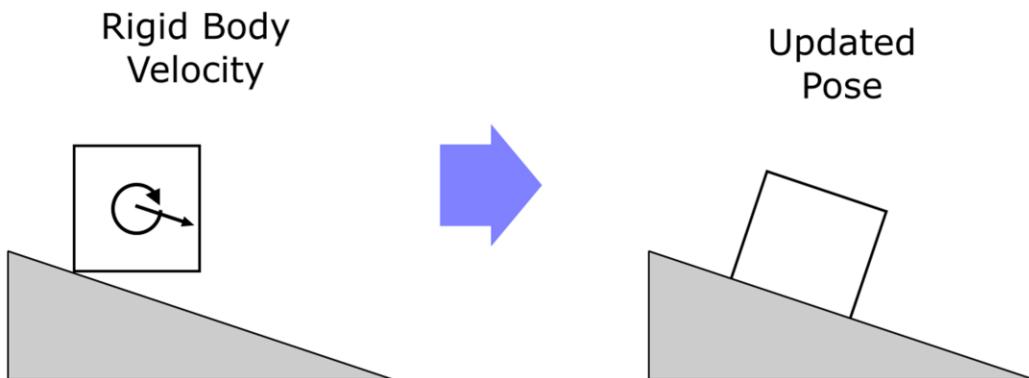
Recall from earlier that you can apply an impulse anywhere on a rigid body by calculating the equivalent impulse in rigid body coordinates and applying that. The rigid body impulse is shown in the right hand picture. Notice how applying the impulse off-center causes a rotation as well as a linear impulse.

Applying the impulse



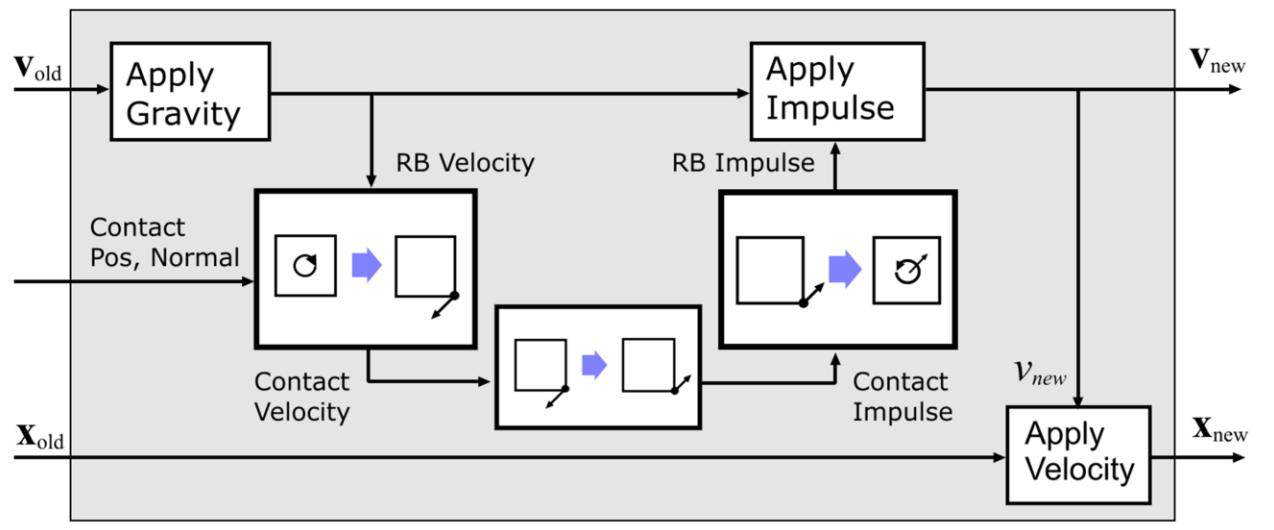
When you apply the impulse to the unconstrained velocity, the linear part of the new velocity aligns with the slope, just as we had foreseen.

Applying the velocity



Now all you 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 you are going to get more contacts from the collision detection to stop it rotating further through the slope. You'll hear about multiple contacts later.

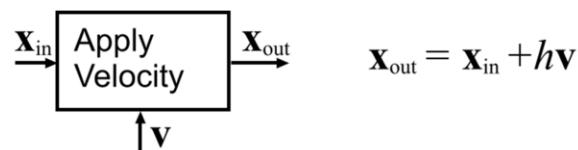
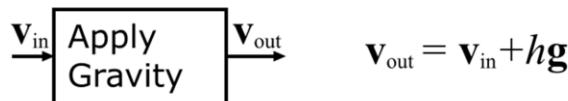
Putting it all together



Putting all the previous steps together, this is what you 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



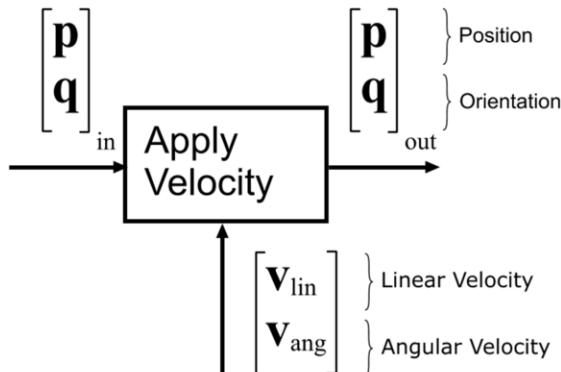
h is the timestep size, 1/60 seconds for 60Hz

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.

Rotation Complicates Things



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

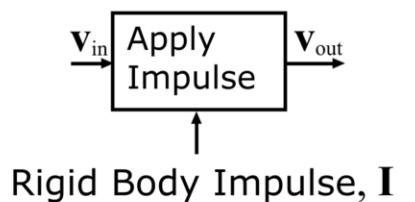
The velocity application in the last slide contains a slight problem. I wrote it the way I think about it, but it's not actually true.

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



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

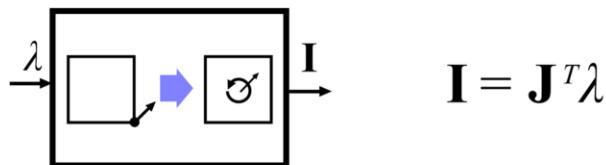
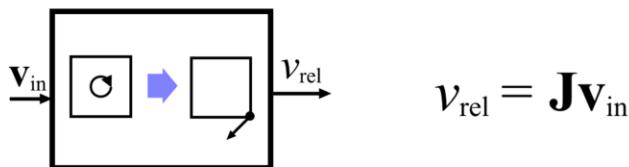
Mass Matrix

$$\mathbf{M} = \begin{bmatrix} m & & & \\ & m & & \\ & & m & \\ & & & \ddots & \ddots & \ddots \\ & & & \ddots & \ddots & \ddots \\ & & & & \ddots & \ddots \end{bmatrix}$$

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.

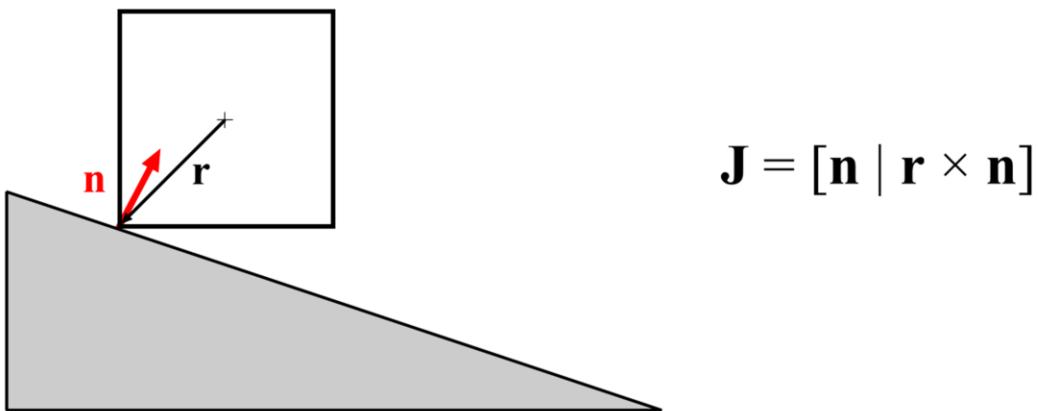
Summary of operations 2



Now you need to implement the boxes that give you the velocity at a point, given the rigid body velocity, and also the box that converts an impulse applied at a point to a rigid body impulse.

The first box is implemented by multiplying the rigid body velocity by a matrix J , and the second box is implemented by multiplying by the transpose of the same matrix J . In the next slide you'll see what J is.

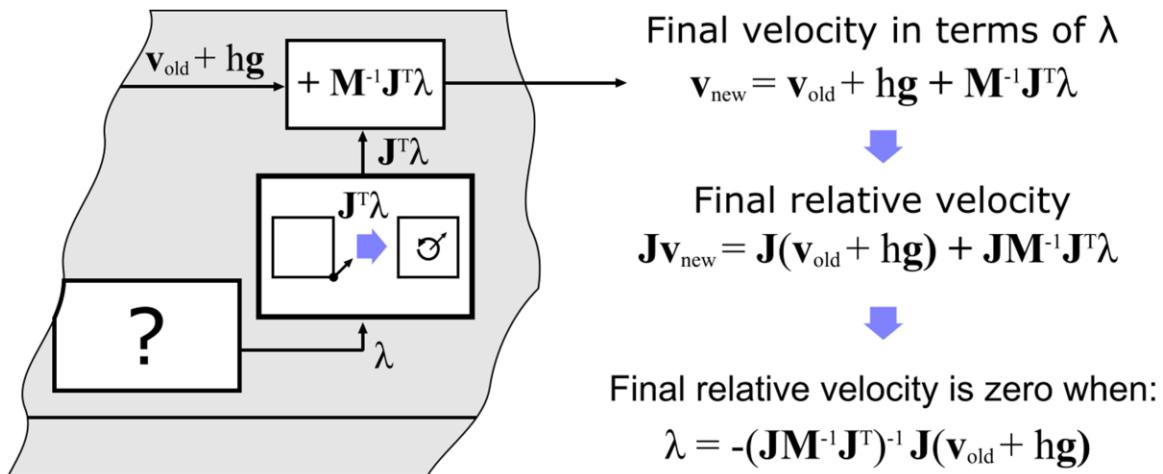
What is \mathbf{J} ?



\mathbf{J} is a 1×6 matrix, the first three elements are the linear part and the second three elements are the angular part. The linear part is the contact normal, and the angular part is the offset of the contact from the center of mass crossed with the contact normal.

Remember in high school that you learned that torque is force multiplied by perpendicular distance? The cross product does this same kind of operation, but in 3D.

Calculating the impulse (λ)



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 (λ) 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 λ

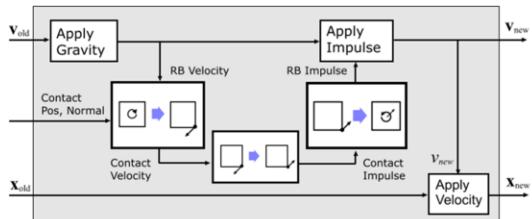
$$V_{\text{new}} = V_{\text{rel}} + M^{-1} J^T \lambda$$

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

$$\text{So we want } J v_{\text{new}} = 0$$

$$J v_{\text{new}} = J V_{\text{rel}} + J M^{-1} J^T \lambda = 0$$

Single Contact Algorithm

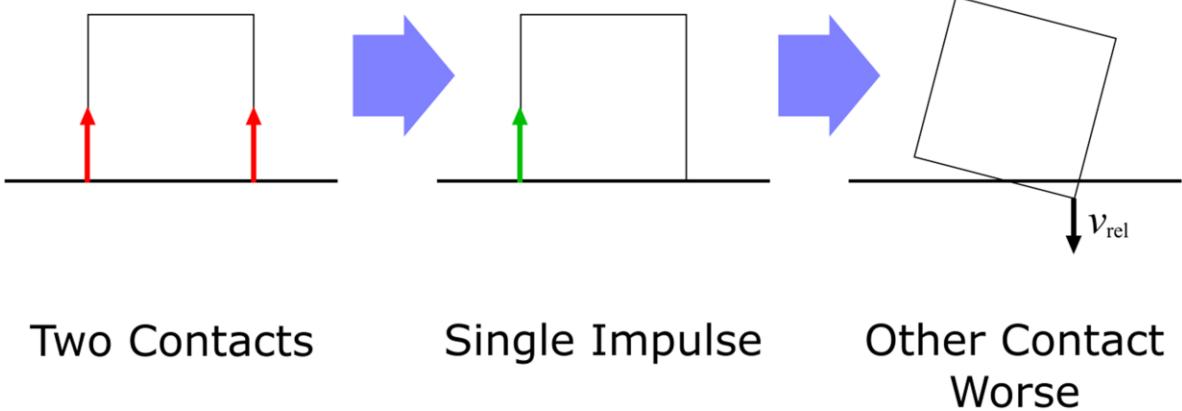


=

```
solveSingle(x, v, J, M, h, g)
{
    λ = (-(JM⁻¹JT)⁻¹ J(v + hg))⁺
    v = v + M⁻¹JTλ
    x = x + hv
}
```

Now you have all the information needed to implement the single contact solver diagram. On the right is the code.

Multiple Contacts



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

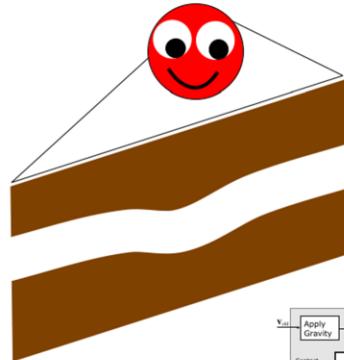
You need to find a set of impulses, one for each contact so that when they are applied simultaneously, the velocity constraints are satisfied simultaneously (taking into account all the coupling between the contacts)

Multiple contacts

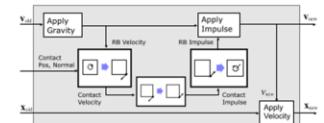


$$0 \leq \mathbf{v}_{\text{rel}} \perp 0 \leq \lambda$$

Model first



Algorithm second



I could just give you the multiple contact algorithm now, but I'm not going to.

First I'm going to show you the model that the multiple contact algorithm solves.

The value of knowing the model

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



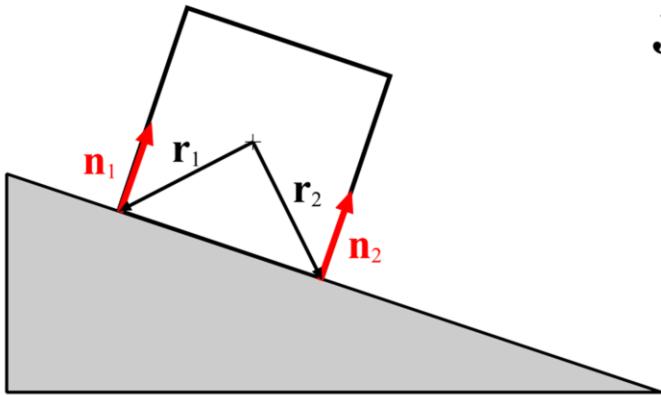
Why am I doing this to you? You just need to know the final algorithm so that you can code it, right?

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 \mathbf{J} ?

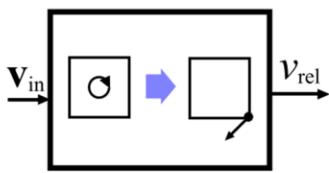


$$\mathbf{J} = \begin{bmatrix} \mathbf{n}_1 & \mathbf{r}_1 \times \mathbf{n}_1 \\ \mathbf{n}_2 & \mathbf{r}_2 \times \mathbf{n}_2 \end{bmatrix}$$
$$= \left[\begin{array}{ccc|cc} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot & \cdot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



In the multiple contact case you need to know what the relative velocities are at a set of contact points. You can do this by making a \mathbf{J} matrix with one row for each contact, and constructing each row in the same way as in the single contact case.

Operations still work with new **J**



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

$$\begin{aligned} v_{\text{rel}} \text{ at contact 1 } [\cdot] &= \left[\begin{array}{ccc|cc|c} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{array} \right] \begin{bmatrix} \cdot \\ \cdot \\ \vdots \\ \cdot \end{bmatrix} \\ v_{\text{rel}} \text{ at contact 2 } [\cdot] &= \left[\begin{array}{ccc|cc|c} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{array} \right] \begin{bmatrix} \cdot \\ \cdot \\ \vdots \\ \cdot \end{bmatrix} \end{aligned}$$



Applying **J** to the rigid body velocity now gives you a vector of relative velocities, one for each contact.

System is a matrix equation?



Given this, you may be thinking that the multiple contact problem is a matrix equation that you could solve using a standard linear system solver. Is that right?

No



$$\lambda = \text{LCP}(A, b)$$



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.

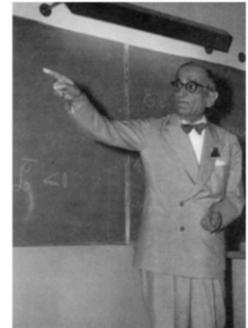
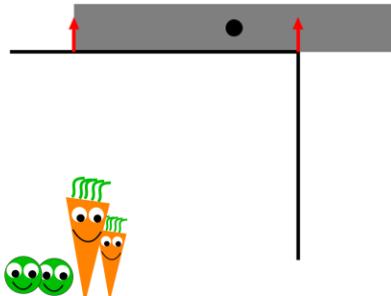
Why?

Contacts can break



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

When Should Contacts Break?



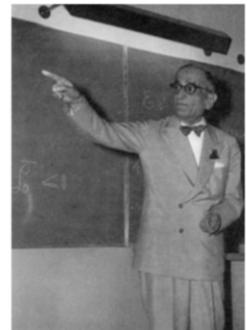
Antonio Signorini



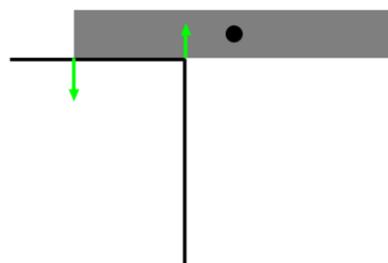
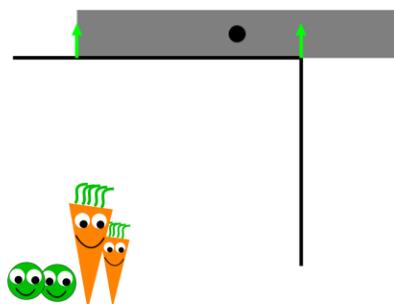
On this slide you can see two books sitting on the edge of a table. The circle represents the center of mass. The collision detection system has generated two contacts in each case, shown by the red arrows.

Intuitively, the book on the left should stay on the table, and the one on the right should fall off the table. As the book 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



Antonio Signorini



Bar won't fall

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.

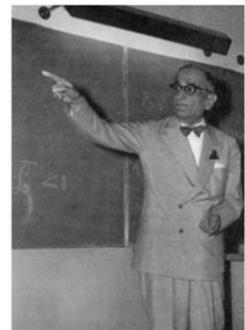
The problem is that the only way the solver can achieve this in the right hand picture is to apply an attractive force on the left contact. This is shown by the downward green arrow. The attractive force and zero relative velocity mean that the bar won't fall.

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

The Signorini Conditions:

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

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



Antonio Signorini

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



No impulse at separating contacts

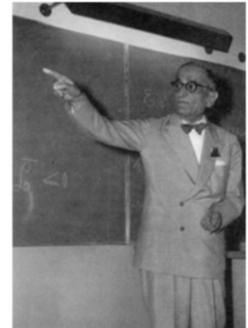
Here is how you can specify what should happen in terms of velocities and impulses:

You've seen that for contacts, you want impulses to be non attractive (non negative), and you 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 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 slide 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:



Antonio Signorini

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



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.

The Final Model

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

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

$$\mathbf{0} \leq \boldsymbol{\lambda} \perp \mathbf{0} \leq \mathbf{Jv}$$



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.

Discretized model

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

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

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

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

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

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

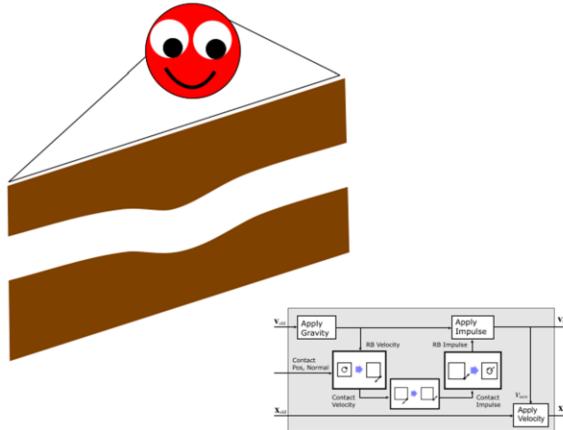
Now you have this idealized model which shows with infinite resolution how position and velocity vary over time, between collisions graphs of these things are perfectly smooth.

It is not possible to solve this model exactly in all interesting cases, and you 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 I've just replaced acceleration with $(\mathbf{v}_{\text{new}} - \mathbf{v}_{\text{old}})/h$ etc.

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

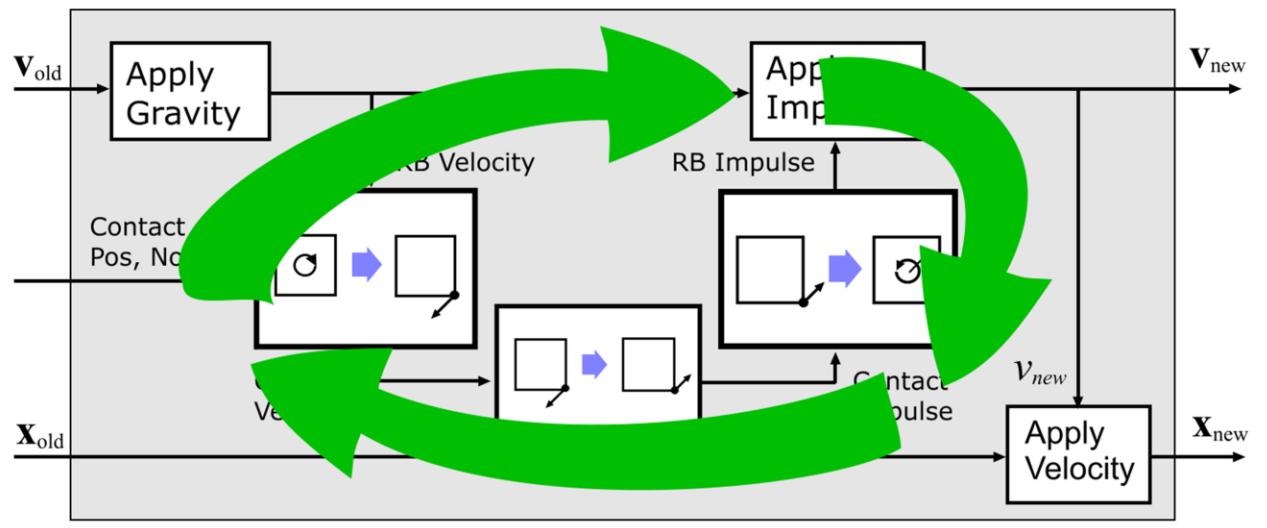
This unfortunately means that any existing linear system solvers that you might know about are not going to work.

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 you saw earlier.

Sequentially Applying Impulses



All you have to 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.

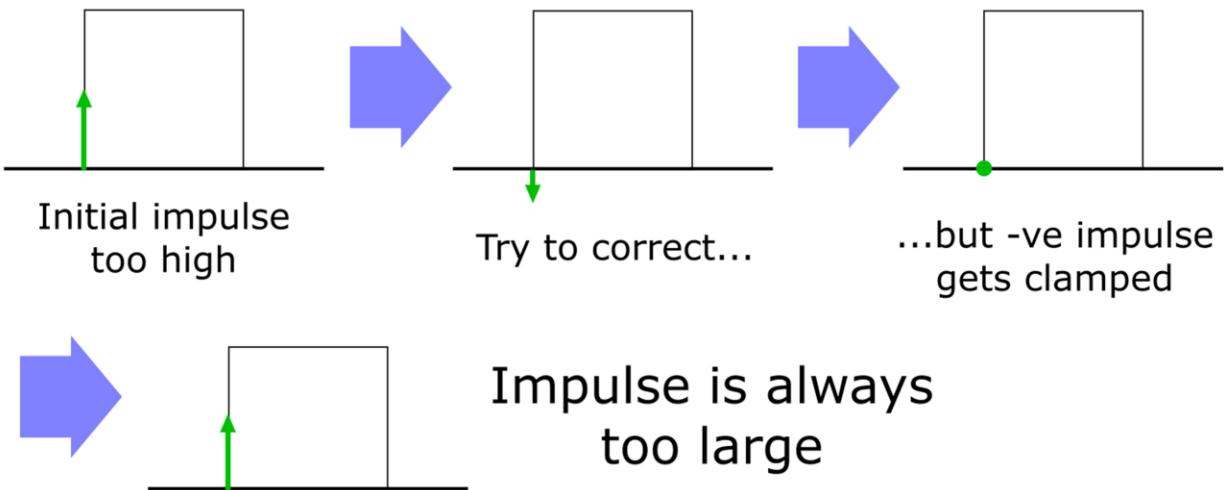
Initial idea to enforce Signorini

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

The question though is how you can ensure that the Signorini condition is met so make sure that your 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.

Potential problem



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

You will need to iterate over all the contacts many times to converge to the correct solution.

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.

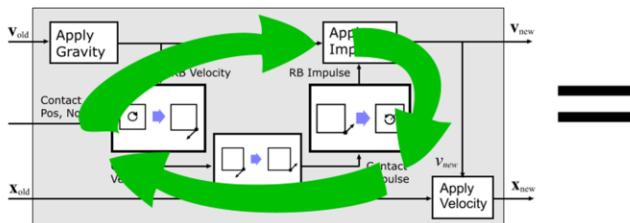
Suppose that on the first iteration you apply too much impulse at a contact. If you clamp the impulse applied on each iteration, then you would never be able a negative correction to reduce the impulse that was too large.

Solution: Clamp total impulse

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

Instead, you just need to keep accumulators that track how much impulse was applied to each contact this frame and clamp those.

Multiple Contact Algorithm



```

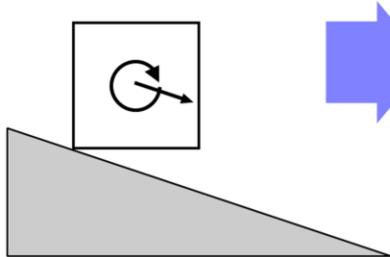
solveMultiple(x, v, J, M, h, g)
{
    for j=0; j<iterCnt; j++
    {
        for i=0; i<contactCnt; i++
        {
            t = λi
            m = JiM-1JiT
            λi = λi - (1/m)(Jv + b)
            λi = max(0, λi)
            v = v + M-1JT(λi - t)
        }
    }
    x = x + hv
}

```

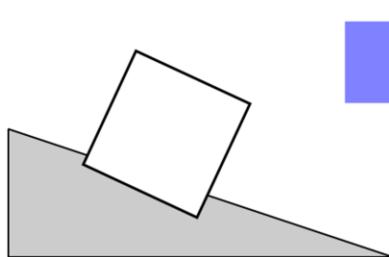
Here is the final algorithm in code

Position projection

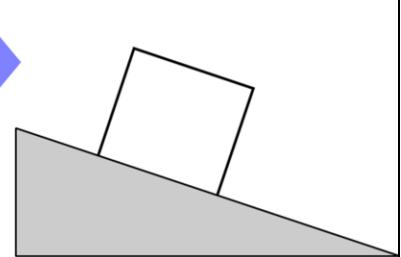
Rigid Body Velocity



Fixed timestep,
moved into penetration



After position
projection



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

As the timestep size is fixed you can't completely ignore position errors though.

The middle diagram shows what might happen if you 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 you 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.

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 can tell you how much penetration has occurred, here I represent it with the letter Phi. It is better to just remove a proportion of the penetration 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 you saw that \mathbf{Jv} gives the relative velocity that you want to zero (or allow to be positive). All you need to do is add 80% * Phi / h to this.

Ok, that's not exactly how we do it in PhysX, Erin's previous talks cover some of the other ways to do this.

Section 3

How to make rigid bodies do the right thing using a solver (multicore)

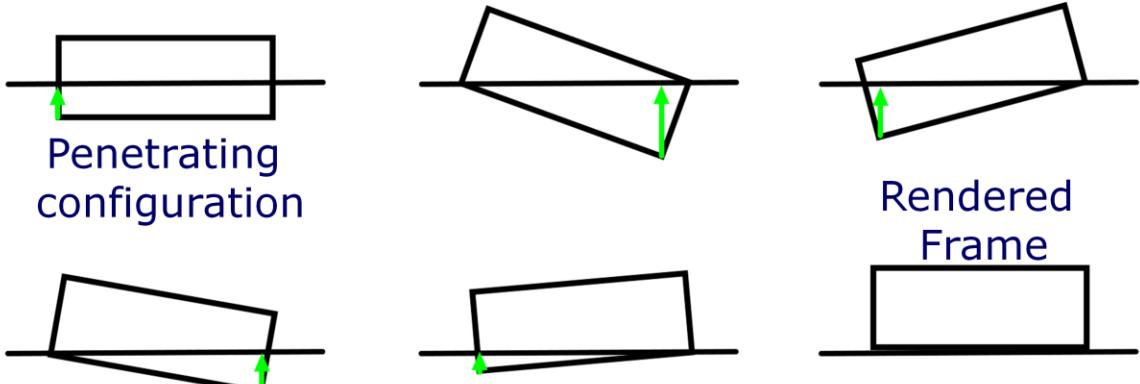
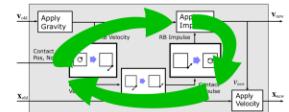
That completes the description of the widely used PGS/SI algorithm for rigid body contact. For single threaded implementations, it is a fine algorithm.

If you are a physics engine developer, then you are going to be asked at some point in your career to write a multithreaded solver. Why?

CPU clock speeds are stagnating, so scaling performance means using more cores. Also, at the recent announcement of the Playstation4, Sony demonstrated that the PS4 has some ability to run physics on the GPU, and GPUs need lots of threads to run efficiently.

Next you'll see how to write a rigid body solver that uses many threads.

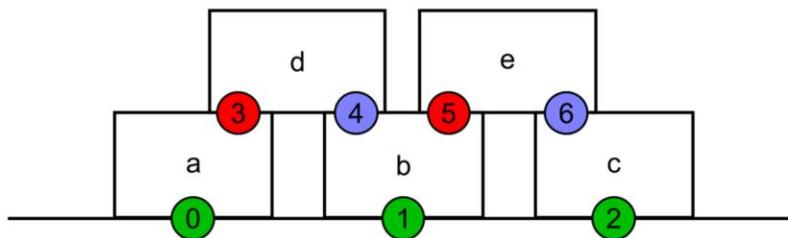
Existing solver method 1:



Suppose that you are simulating a box that has just collided with the ground. The algorithm I described earlier fixes one contact, causing the body to rotate one way, and then fixes the other contact, making it rotate the other way. You iterate between the two contacts, and the amount of rotation will decrease at each iteration, until you have something that you can render.

Notice that this is not parallelizable. To calculate the next impulse you need the velocity change from the current impulse.

PGS coloring



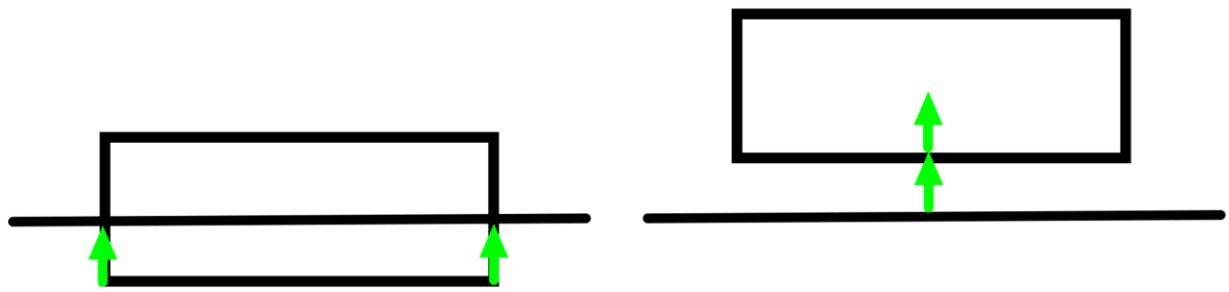
You may be thinking that this algorithm (PGS) has no parallelism.

With more than one body you can get some parallelism by coloring. What you do is assign colors to contacts such that in each color each body is referenced at most once. Then you can process the contacts in each color in parallel, and do the colors sequentially.

For example, in this diagram, contacts 3 and 5 can be done in parallel, and then 4 and 6 can be done in parallel.

This is not great though, because the number of contacts that can be done in parallel is small compared to the number of bodies.

Existing solver method 2:



A different thing you could try is assigning one thread to each contact, and have each thread calculate its contact without considering what is going on at the other contacts.

In this diagram you can see that this leads to each contact applying too much impulse.

- Method 1 (Projected Gauss Seidel, PGS)

- Provably convergent ✓
- Limited parallelism ✗
- Jitters ✗
- Widely used

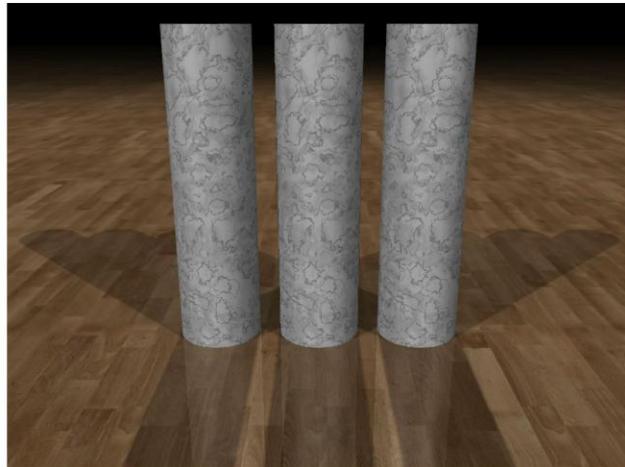
- Method 2 (Projected Jacobi)

- Maximally parallel ✓
- Jitter free ✓
- Non convergent in many cases ✗
- Converges slowly ✗
- Unusable in games

Method 1 (PGS) is provably convergent, but has limited parallelism, as you saw on the coloring slide. It also suffers from a problem called jitter, which I will talk more about in the next slide. The method is widely used though.

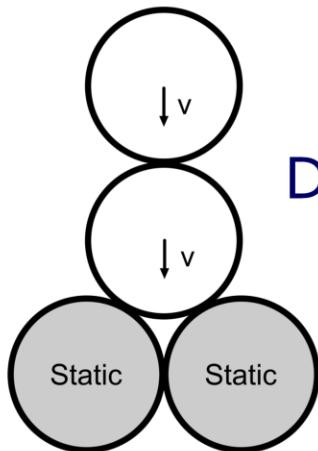
Method 2 (Projected Jacobi) was maximally parallel so it seems like it would be a good choice for a multithreaded solver, and doesn't suffer from this jitter problem, but doesn't converge in some simple cases and converges too slowly, so is not used in practice.

Parallel PGS Jitter



This video shows the jitter problem. You will see three concrete columns be destructed. The debris will fall to the floor, and you'd expect the debris to come to rest in piles. With PGS with coloring, you don't see this, instead the pieces continue to move, and this is called jitter.

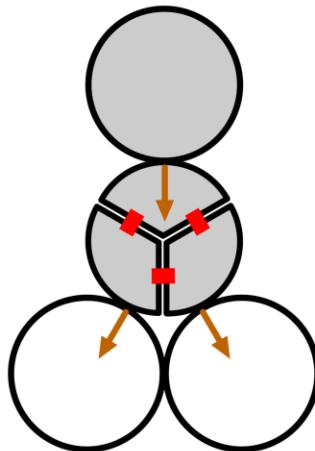
Example



Doesn't converge with Jacobi

Jacobi seemed like a good idea, but it doesn't converge in some simple cases. Here is an example.

First idea: Spatial splitting

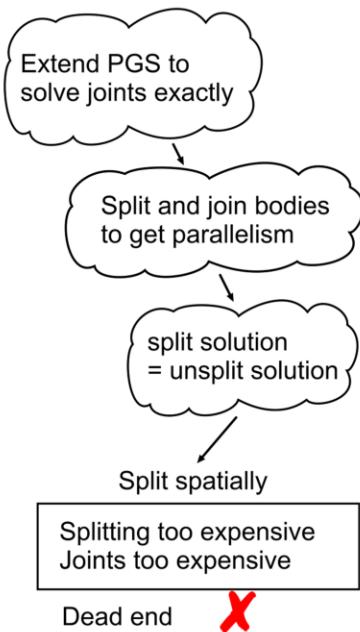


I'll show you how we made an algorithm that is similar to Jacobi, but does converge, and converges fast enough to be used.

The problem with the previous example is that the three impulses on the middle body are calculated without knowledge of each other, and they just get bigger and bigger, causing divergence.

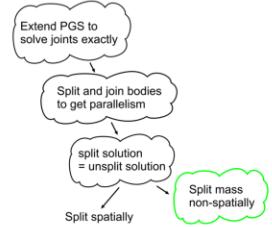
The first idea we had was to split the middle body spatially, so that each force has a separate sub-body, and then join them back together with fixed joints.

This wasn't a good idea, splitting a triangle mesh spatially and recalculating its inertia is expensive, and takes lots of memory. Also you need extra time and memory for the fixed joints.

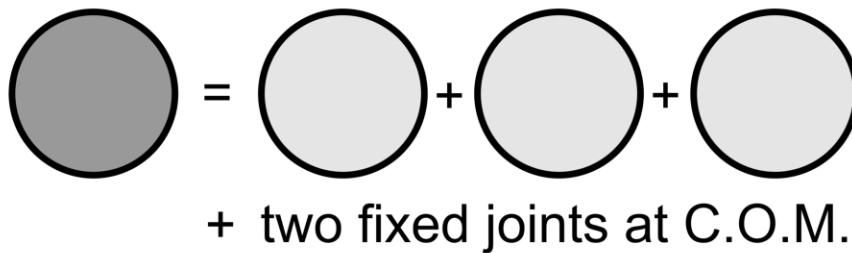


This slide shows the progression of ideas that lead to this dead-end.

New idea: Mass splitting

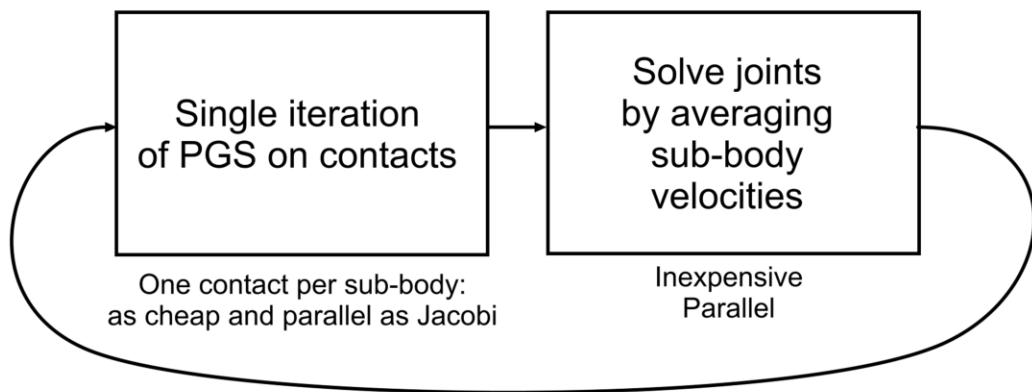


Make sub-bodies have the same spatial extent



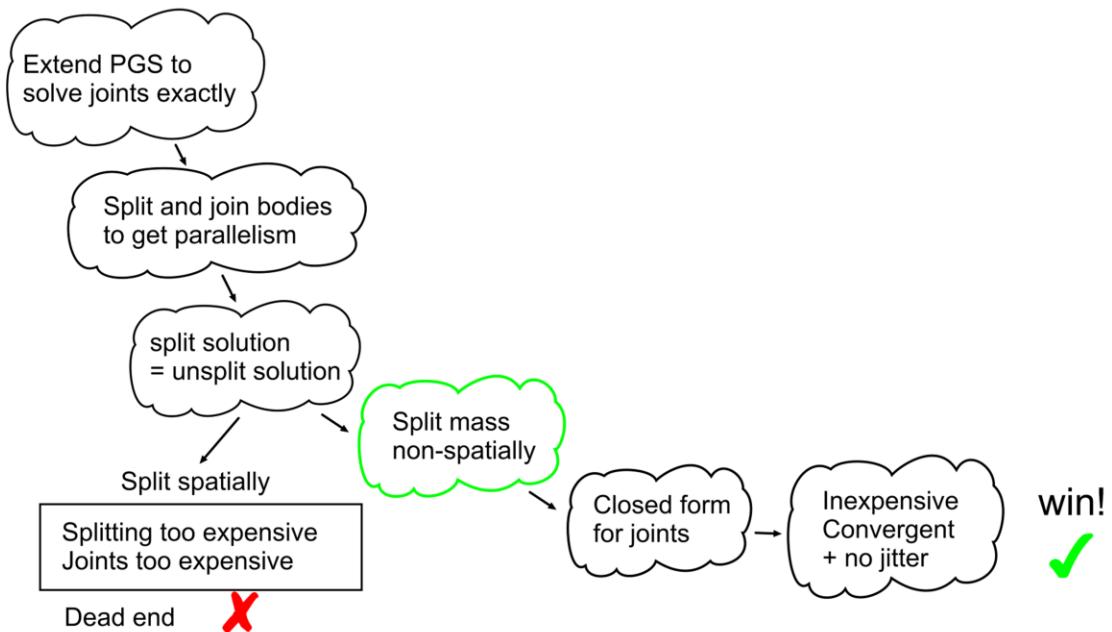
The next idea we had was that you could split the body non-spatially. In other words you could take the center of mass, and split it into 3 pieces that have the same position and spatial extent, each with 1/3 of the mass. You could then fix them together using simpler joints at the center of mass. Splitting the mass is just scalar division, which is much cheaper than splitting and storing geometry.

PGS with exact joints



You still have the joints though. You can solve a system of contacts and joints in a provably convergent way by interleaving PGS iterations with matrix solves for the joints. Solving a matrix for the fixed joints would be too expensive.

We realized that you don't have to solve a matrix to enforce the fixed joints, there is a closed form solution, the average of the sub-body velocities. Averaging a few velocities is very inexpensive.



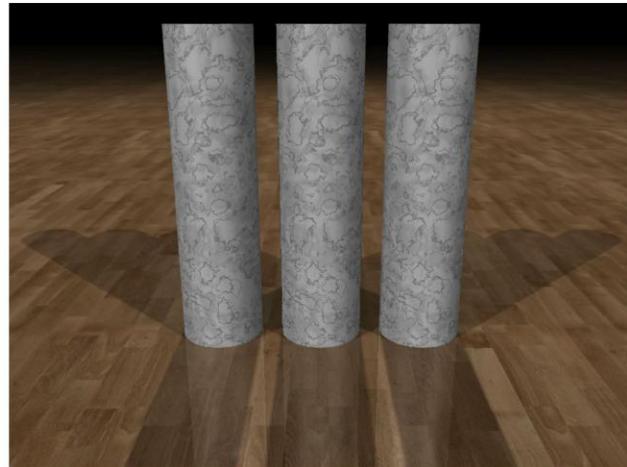
Here is the slide showing the sequence of ideas again.

We had the idea of splitting the mass non-spatially, then we realized that there is a cheap closed form solution for the joints, and we ended up with a method that is parallel, inexpensive, provably convergent and doesn't jitter.

Section 4

Results

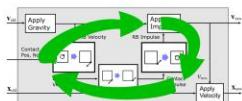
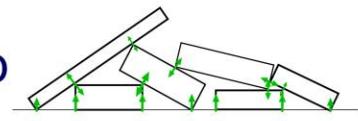
Mass splitting



Here is the same system that you saw earlier using the mass splitting algorithm instead of PGS. As you can see, the jitter is gone. The computation was implemented using thousands of threads on a GPU.

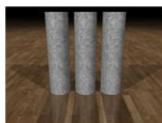
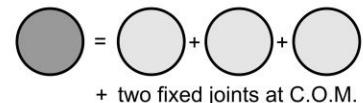
Summary

What we need rigid bodies to do



Single core solver

Parallel solver



Jitter-free

Here is what you just saw:

- To pile objects you need a consistent set of forces/impulses, and that you can calculate such a set using a solver.
- You can solve such systems by applying impulses sequentially, an called algorithm called PGS. This is good for single threaded solvers.
- If you need to use lots of threads, you could use mass splitting, which is jitter-free.

Thanks very much.

Acknowledgments

GPU rigid body technology

Feodor Benevolenski

Andrey Voroshilov

Richard Tonge

Fracture technology and demo

Matthias Müller-Fischer

Nuttapong Chentanez

Tae-Yong Kim

Aron Zoellner

Thanks also to the PhysX SDK team

NVIDIA and NVIDIA PhysX are trademarks and/or registered trademarks of NVIDIA Corporation in the United States and other countries.

