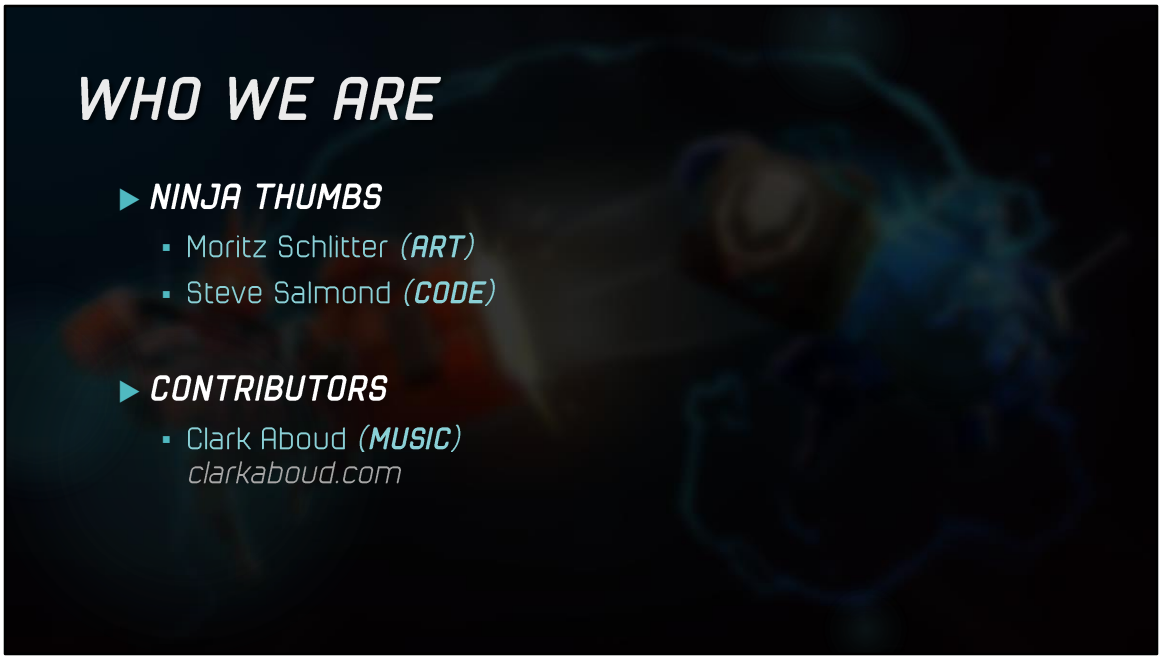

FINDING THE FUN
IN PHYSICS
building a physics-based game in Unity

Hi there, and welcome to 'Finding the Fun in Physics'.
I'll just start by giving you a quick overview of who we are, the game we're making, and what I mean by 'Finding the Fun in Physics'.

Ok, so who are we? We're 'Team Ninja Thumbs', a very small indie studio based here in Auckland.



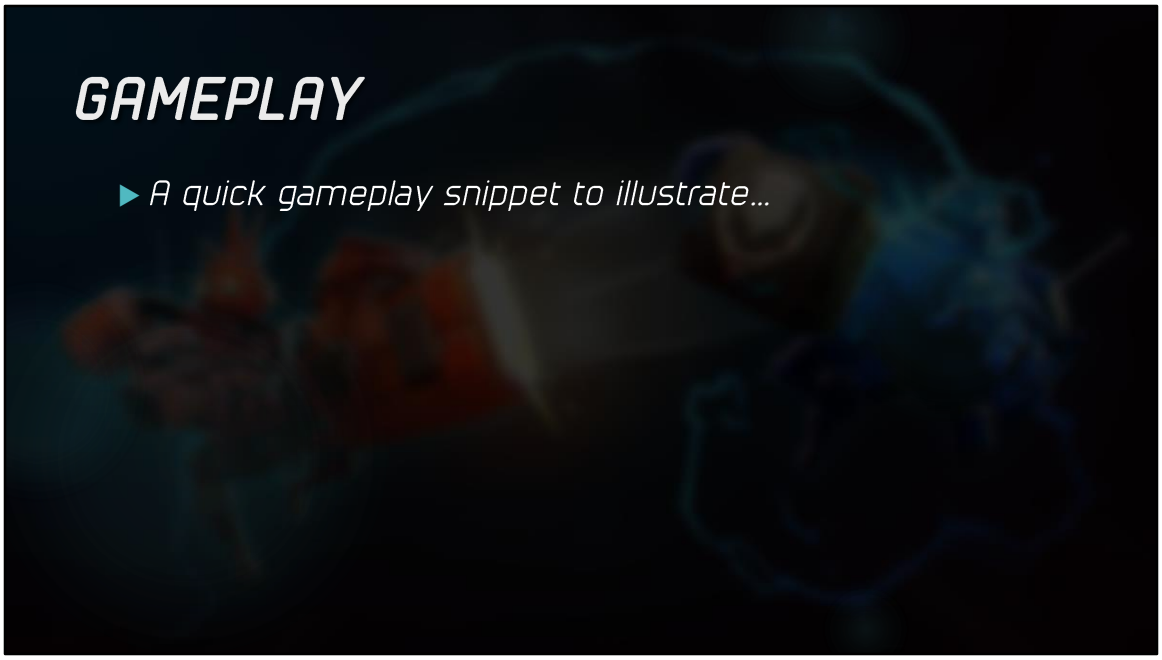
Ninja Thumbs consists of Moritz and myself (I'm Steve).
Moritz is a talented 3d artist and animator, and I'm more involved with the coding side of things.
We collaborate on game design!

For audio, we were lucky enough to meet Clark, who has produced an awesome set of music tracks for us and is really great to work with.

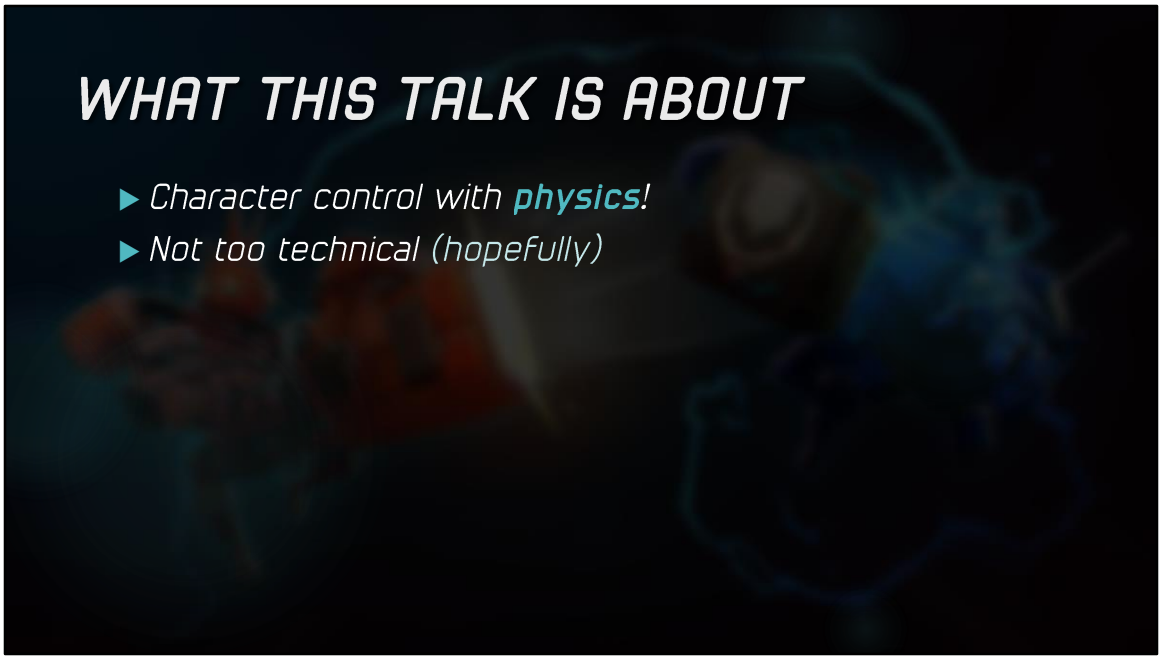
Right, so what are we making? This is our first project: GRABITY.

WHAT WE'RE MAKING

- **GRABITY**
  - A physics-based **arena brawler** for 2-4 players
  - Wield **grab guns** to grab and shoot objects
  - **2.5D** (3D world, most action on Z=0 plane)
  - Emphasis on **fluid movement**

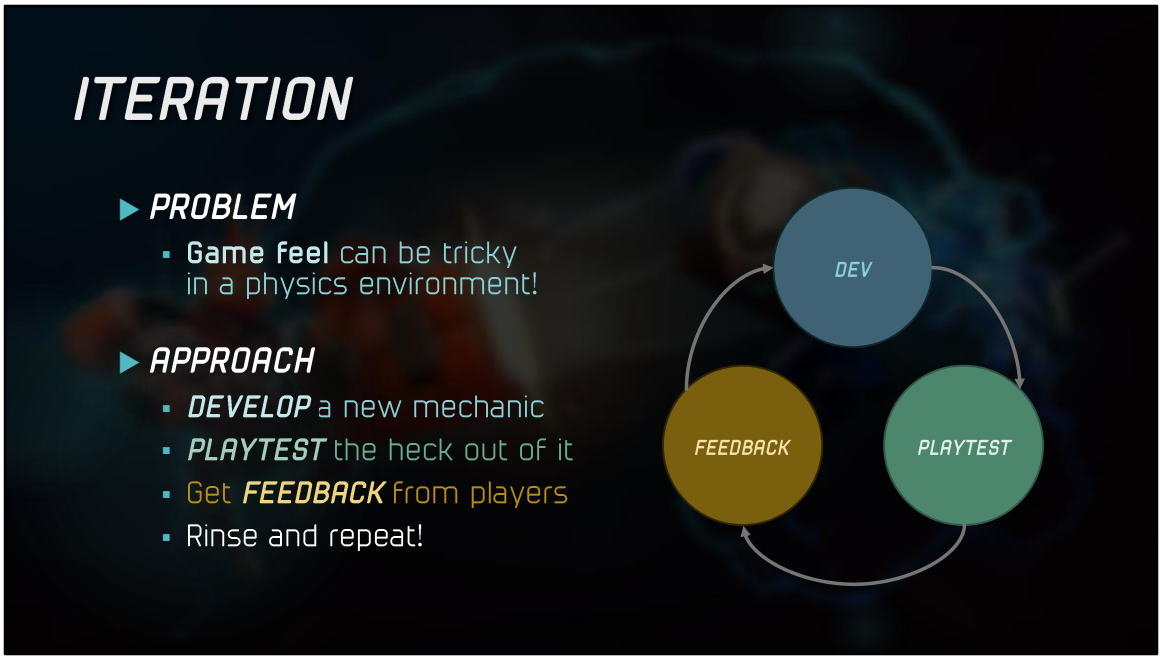

GAMEPLAY
A quick gameplay snippet to illustrate...

WHAT THIS TALK IS ABOUT



So, today I'll be talking about our experiences trying to get player control feeling right, while playing nicely with the physics engine.

This is a short talk, so I'll try to keep it pretty high level – but feel free to ask questions at the end, or come see me afterwards if you'd like to chat.



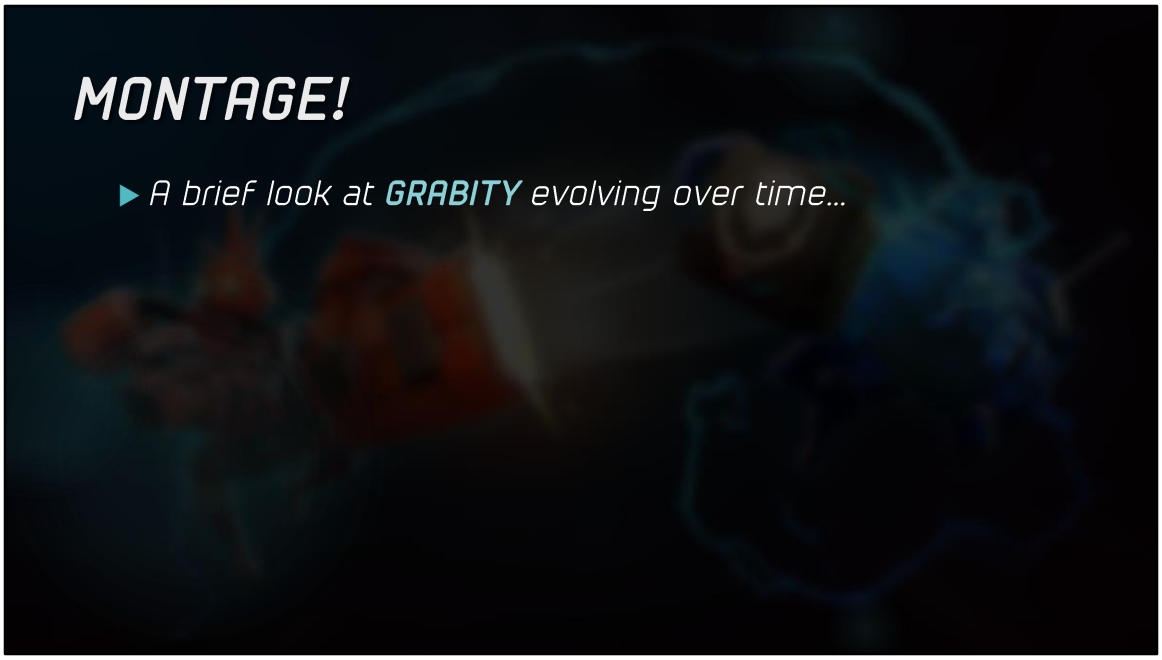
When we began making Grabity, I was concerned about **game feel** – that it would be hard to get controls to feel tight if we drove everything with the physics engine.

So, we started out very simple, **developing** some minimal control mechanics.

Once the game was running, we began **playtesting** with friends from work.

It wasn't pretty, but their **feedback** really helped drive and refine the design quite quickly.

This process continued until we had all the core mechanics in place.

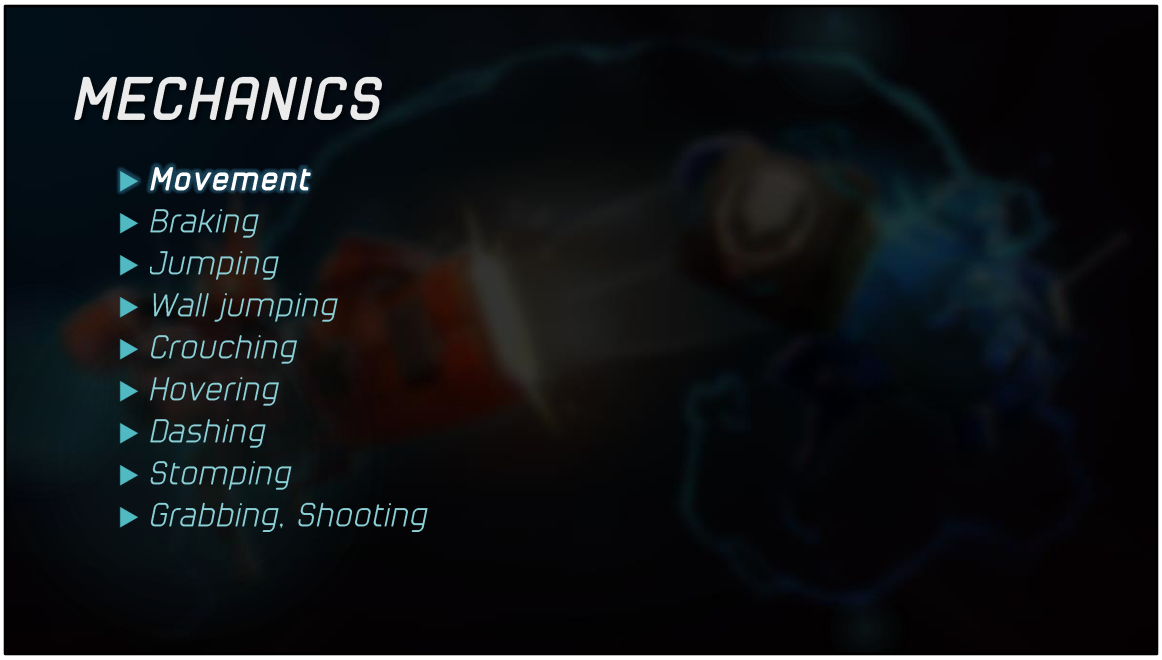


Because everyone loves a montage, here's a (highly condensed) retrospective of that iterative process.

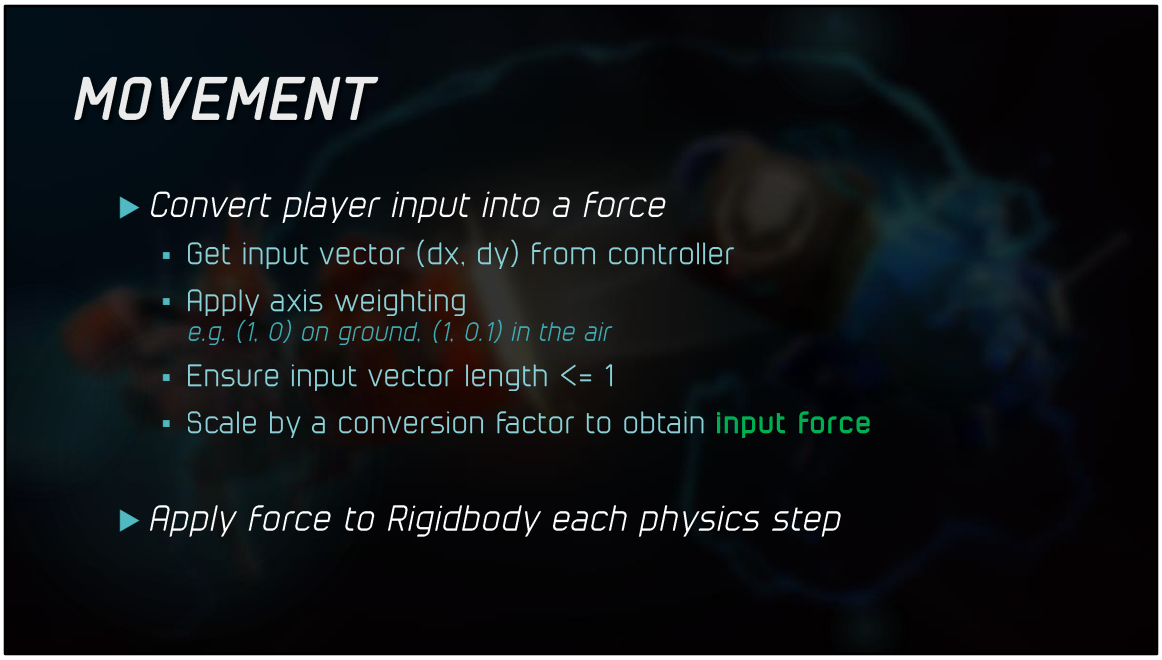

MONTAGE!
grabbing and shooting
playback rate 133%

Ok, that might have been a bit too quick to take in, but hopefully you get a sense of how things evolved over time. ☺

Let's break things down a bit, and go over each mechanic in roughly the order they were implemented.



We'll start with basic movement – walking and running.

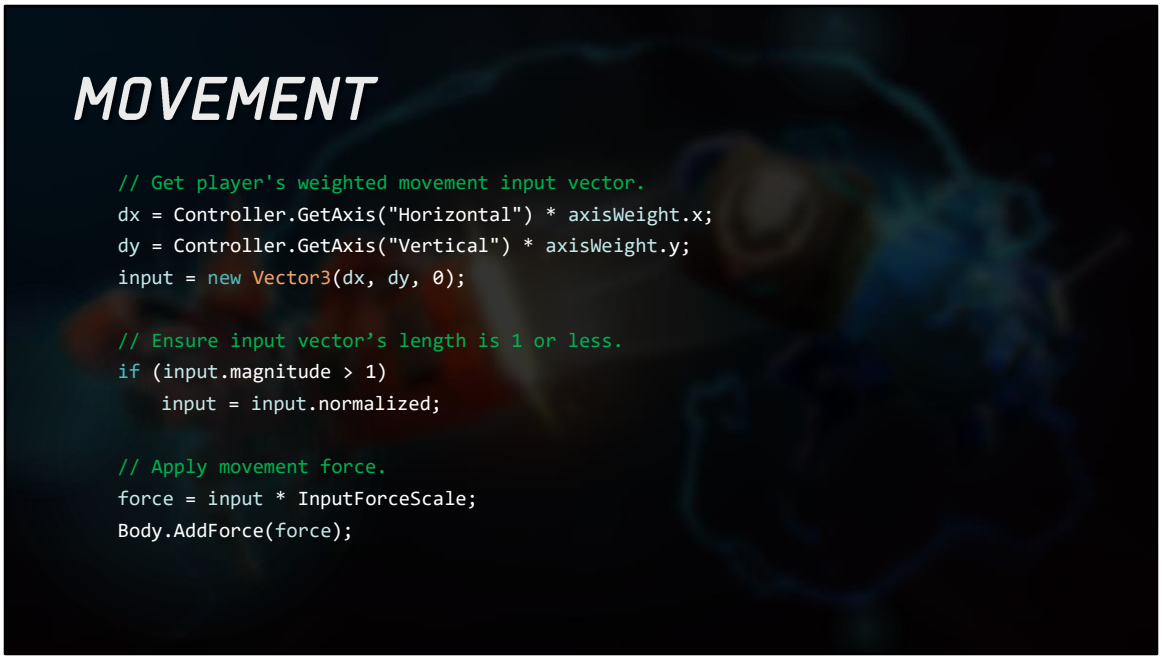


Right, so before you can run or do any other fancy stuff, you have to be able to walk!

Let's take input from the game controller, two numbers **dx** and **dy** ranging from -1 to +1 (representing how hard the player is pushing on the stick) and convert that into a force vector by scaling it.

Since we don't want the player being able to lift off the ground just by pushing up on the stick, we scale back the y component when they are on the ground.
(When you're in the air, we found it's nice to have a little bit of vertical control, so we bring back some of the y component in that case.)

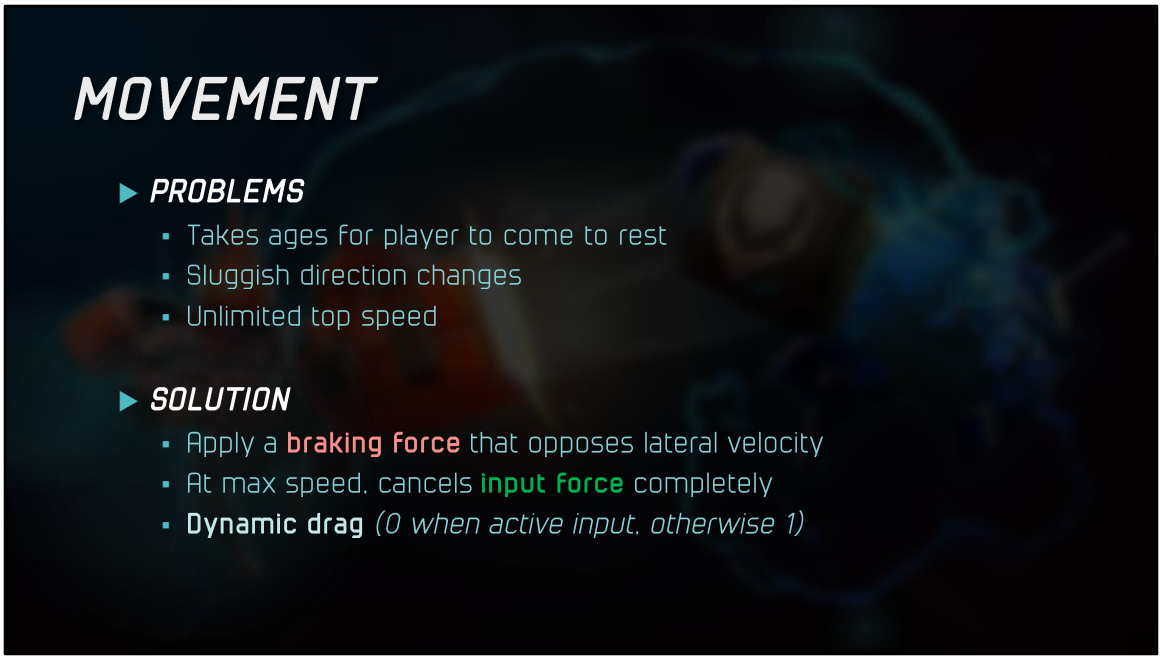
Once you have a force vector, just apply that to the player's Rigidbody and off you go, right?



This pseudocode glosses over details such as detecting whether the player is **grounded** or **airborne**, but hopefully gives a sense of how movement might work.

Let's see what this code looks like **in-game**..


MOVEMENT
sluggish direction changes!
v
input

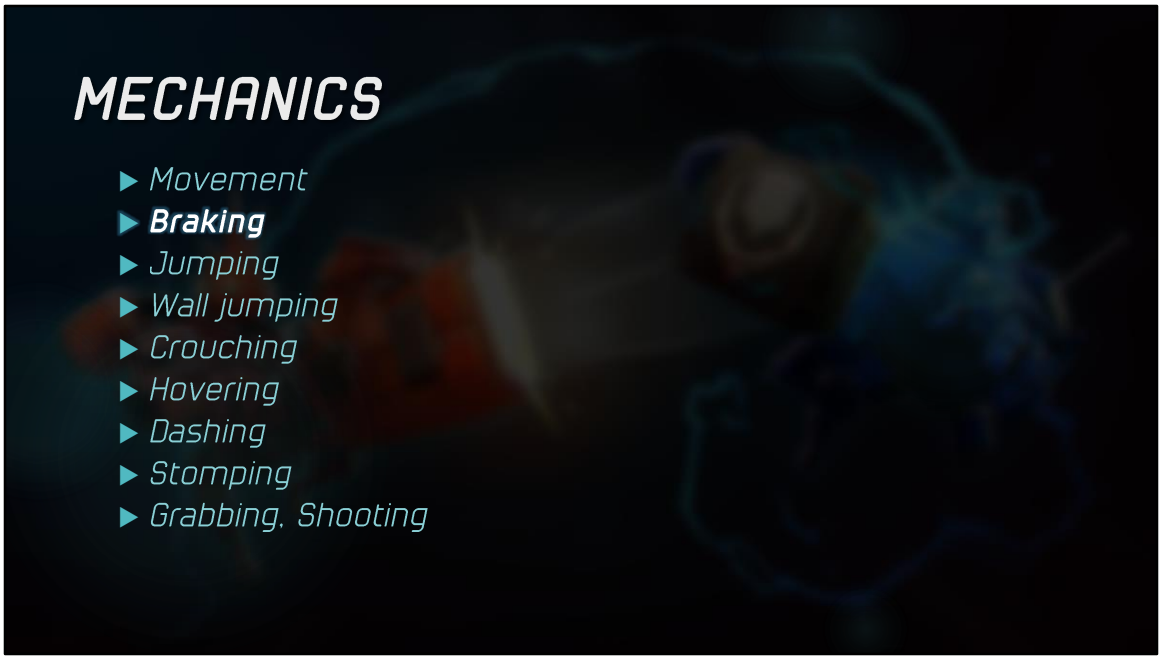


Ok, so not too great. There are a few problems with just chucking a force onto the player's Rigidbody.
It feels like iceskating – it takes ages to stop after you lift off the stick, it's hard to change directions, and you also go crazy fast if you keep pushing in a given direction.

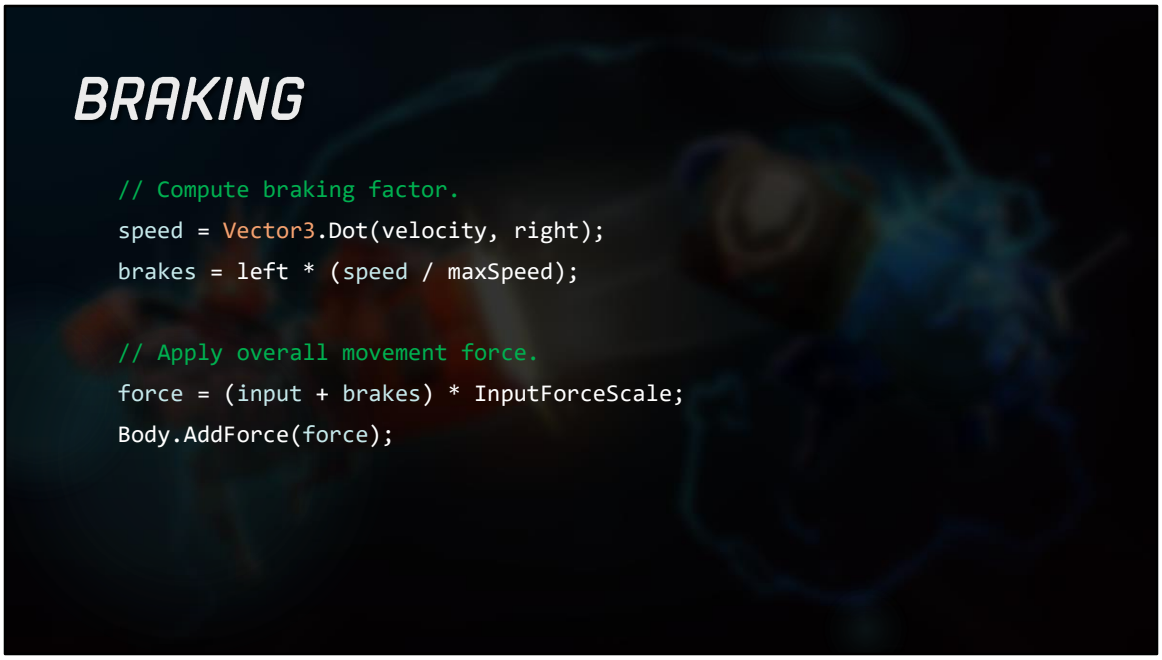
We did a couple of things to resolve this.

First, we added a **'braking' force** that ramps up as the player gets underway. This acts to oppose the player's current velocity, and balances out input once they hit a defined 'top speed'.

We also adjust the Rigidbody's **drag coefficient** depending on whether the player is actively trying to move or not.
This helps to prevent players from sliding off surfaces when they are just trying to sit still.



Ok, so we're onto applying braking forces..



Here's a bit of pseudocode that gives a flavour of the braking force logic.

If the player's travelling right for example, the brakes will end up being a vector pointing left.
When they hit top speed, the braking vector will have a length of one.

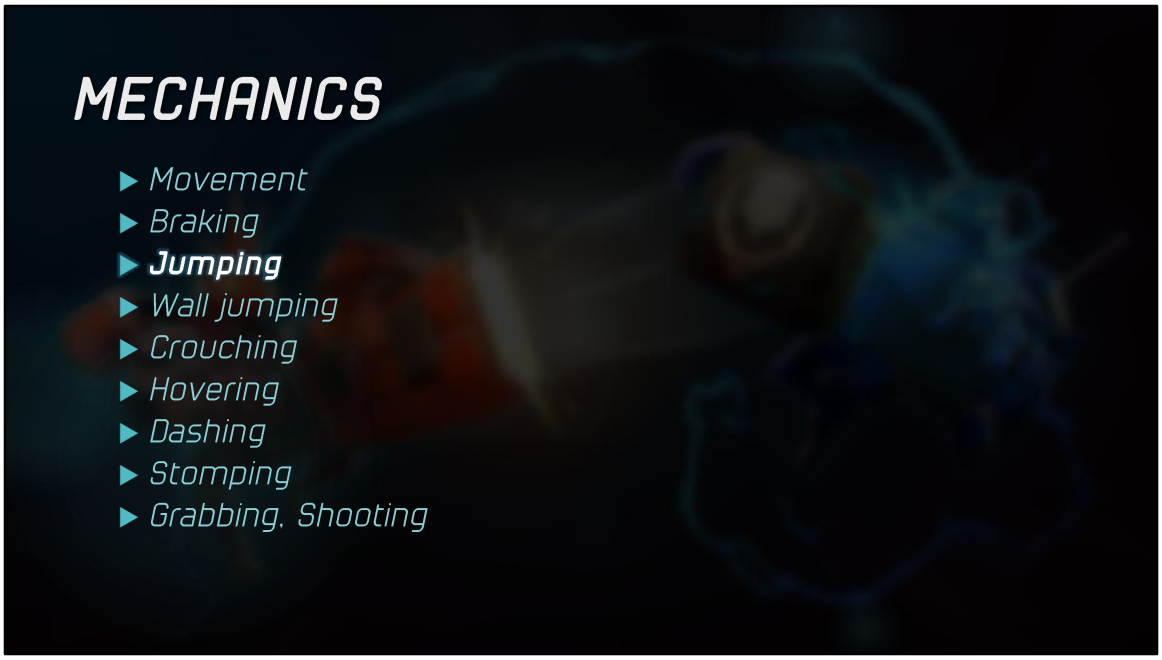
Since the input vector also has a length of one (or less), input and brakes should basically cancel.
If the net force ends up at zero, the player will stay at top speed as long as they keep pushing the stick in the direction of travel.

If the player lifts off the stick, the input vector will become zero, but the braking will continue until they come to rest.

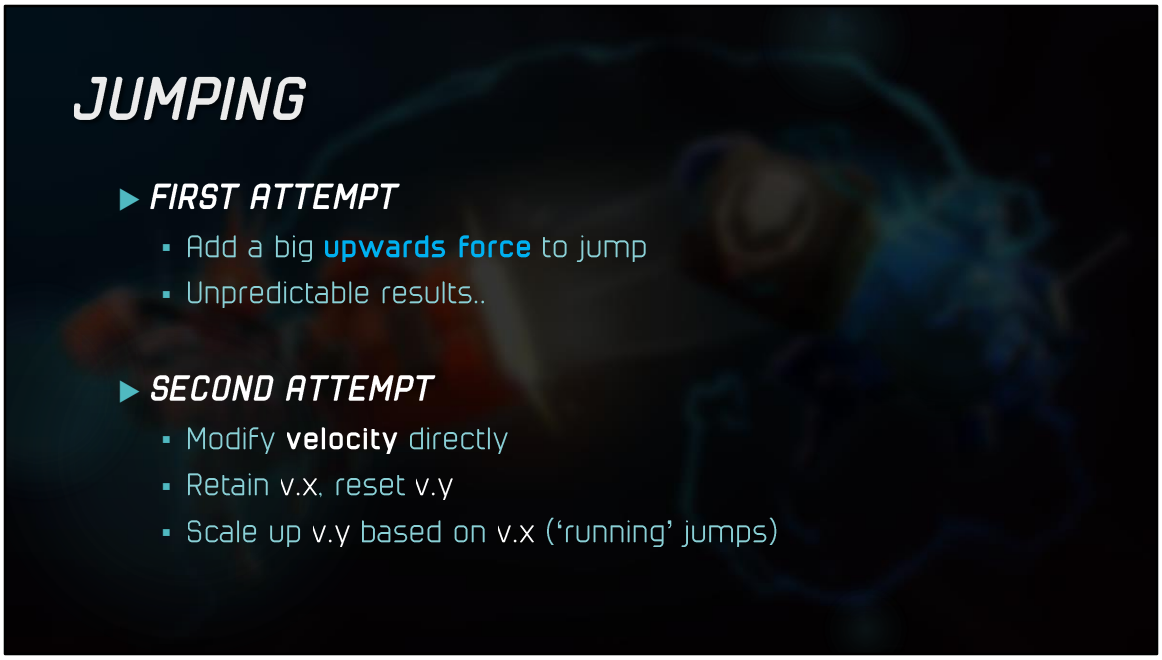
If the player wants to switch direction, the input and braking vectors will add together, resulting in a fast change.

Let’s see what that looks like in-game..

As you can see, the player comes to a halt much quicker, direction changes feel snappier, and top speed is now effectively capped. Better!



Ok, so now we can run, but there's no vertical component to gameplay. Let's figure out how to jump.



The first thing we tried was to use forces again – just add a big impulse force whenever the player is allowed to jump. This actually kinda works, but has some undesirable characteristics, as we'll see shortly.

The approach we ended up with though was to just manipulate the player's velocity directly on a jump event.

(As a bit of an aside, we found that it felt nice to give players a slightly higher/faster jump if they were travelling at speed).

JUMPING

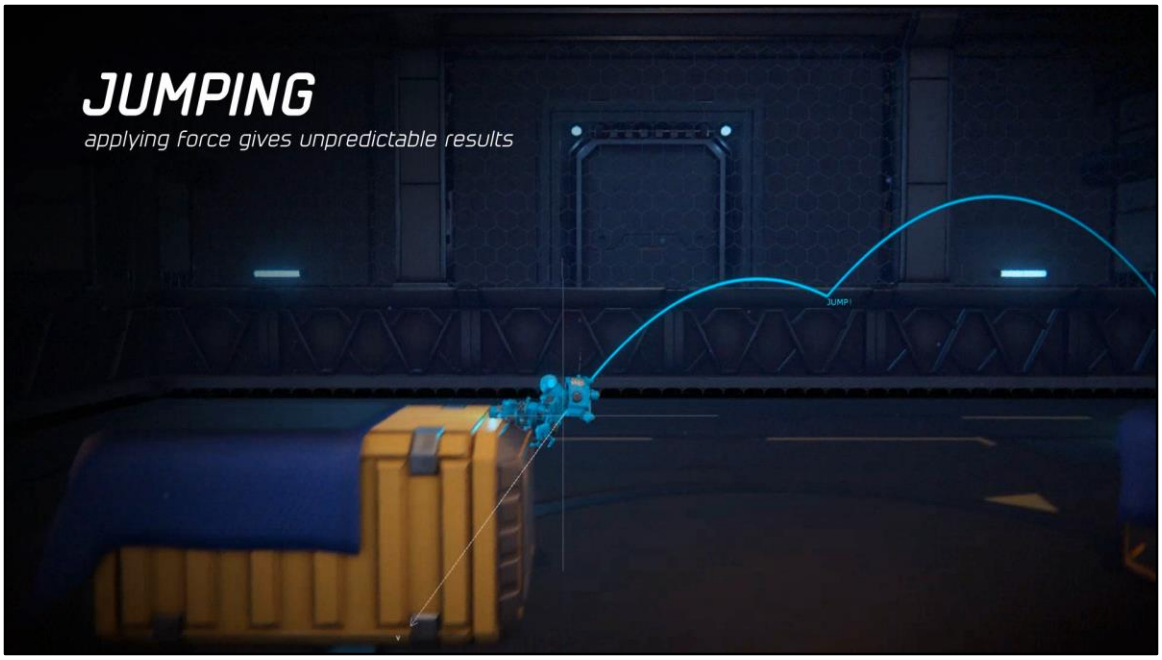
```
// Zero vertical component if airborne.
v = Body.velocity;
if (!grounded)
    v.y = 0;

// Apply jump (depending on lateral speed).
lateralSpeed = Abs(Vector3.Dot(right, v));
speed = JumpSpeed.Evaluate(lateralSpeed);
Body.velocity = v + (up * speed);
```

Here's roughly how the jump logic works. First, we zero out the y component of the current velocity (seems to feel better that way).

Then, we determine upwards jump speed based on how fast the player is travelling laterally, and apply that directly.

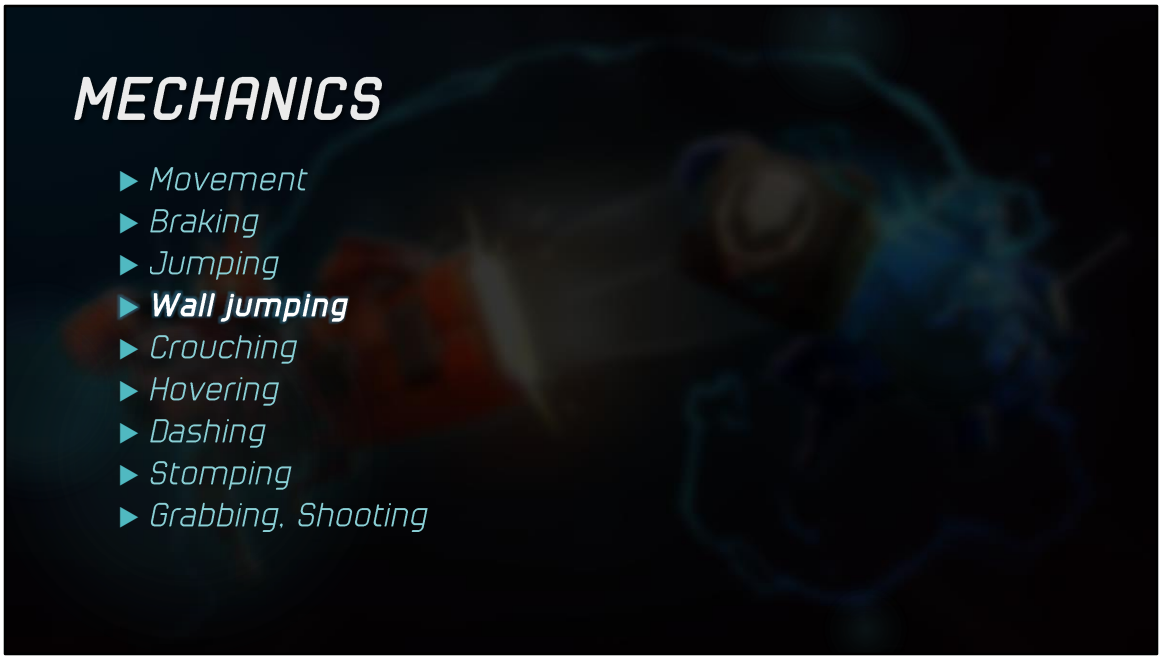
JumpSpeed.Evaluate() is a lookup into a animation curve (given this lateral speed, what should the upward speed be?). An easy way to visually configure things.



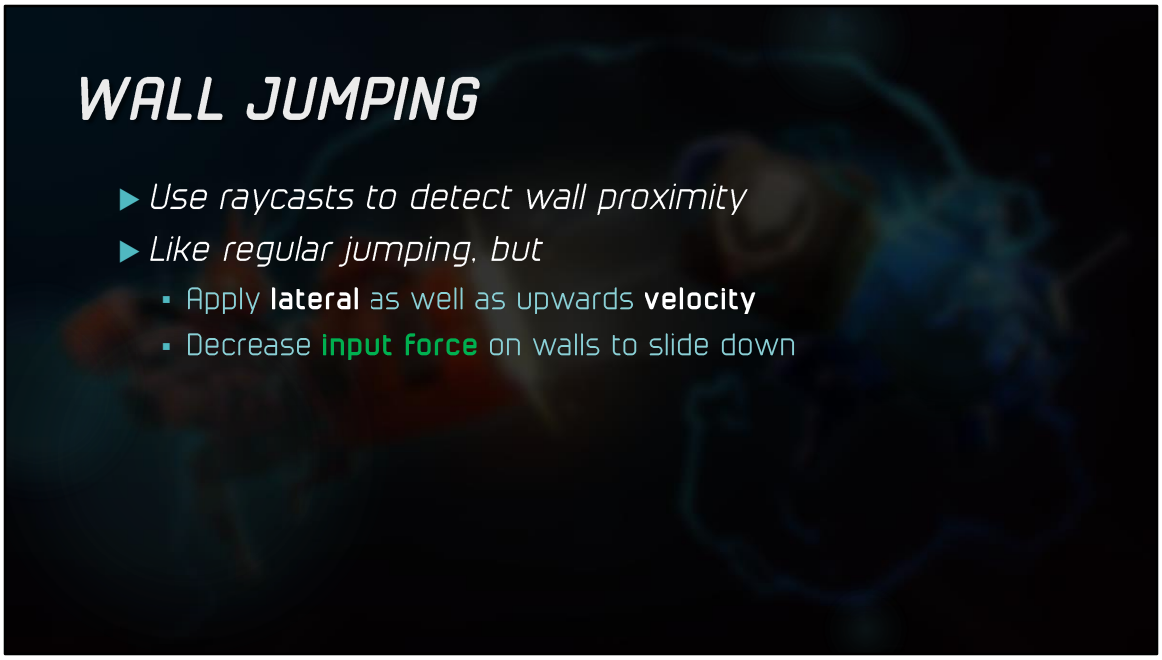
Here’s some footage of the difference between jumping with forces and velocity manipulation.

As you can see, the jump height is quite unpredictable using the former approach; it varies depending on the player’s velocity at the time of the jump.
If the player is already travelling up, they will go quite high, but if they are on the way down the jump is much less effective.

Directly manipulating velocity gives you more predictable results, which players seem to strongly prefer.



Next up is the logical extension of jumping – wall jumping.



Briefly, we detect when players are up against walls by using raycasting with layer masks.

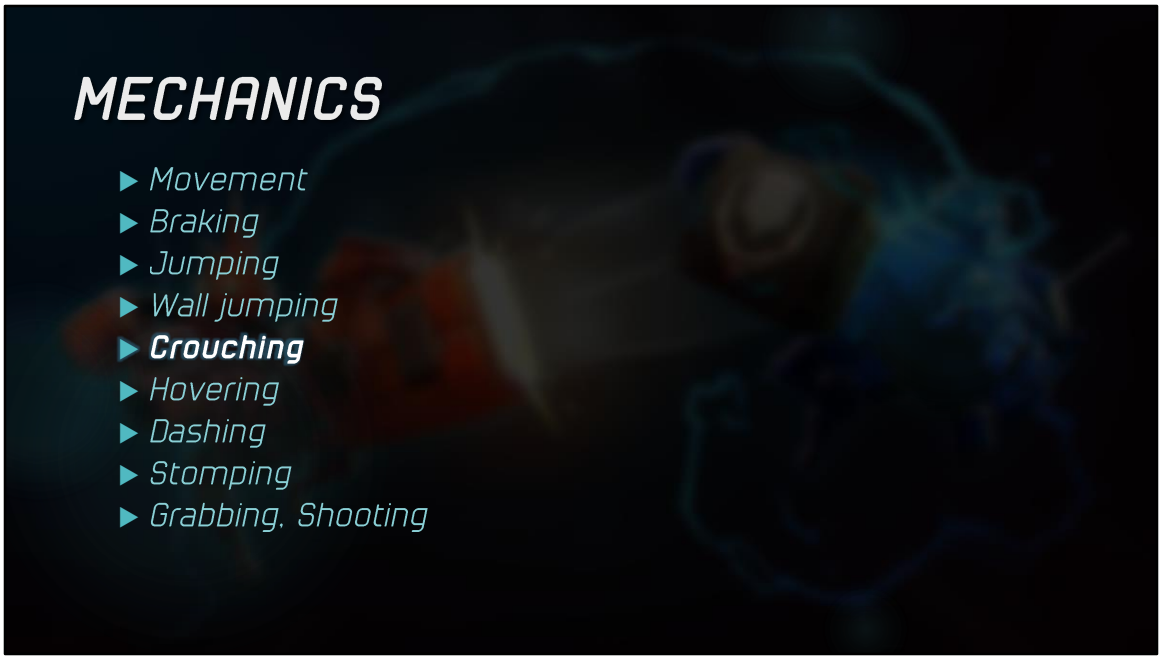
Then, if the player jumps and we know they are against a wall, we add in some lateral in addition to the regular upwards velocity.

Also, we scale down the player's input vector over time if they remain against a wall. This simulates them 'losing grip' over time and sliding down.

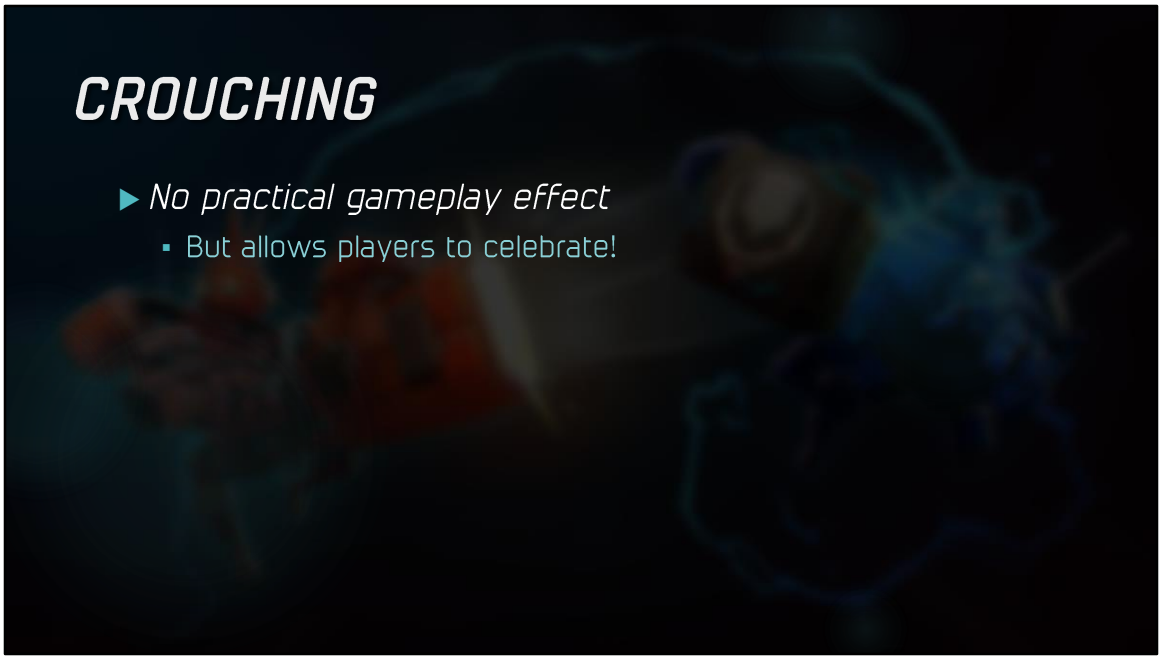
Here's how that might look in code – pretty straightforward stuff, just splice it into the existing jump logic.

Here's how wall jumping looks in-game. Notice the input vector's magnitude reducing over time, which leads to the player sliding down the wall if they don't jump.

You can see lateral velocity being applied on a wall jump, propelling the player away from the wall each time.



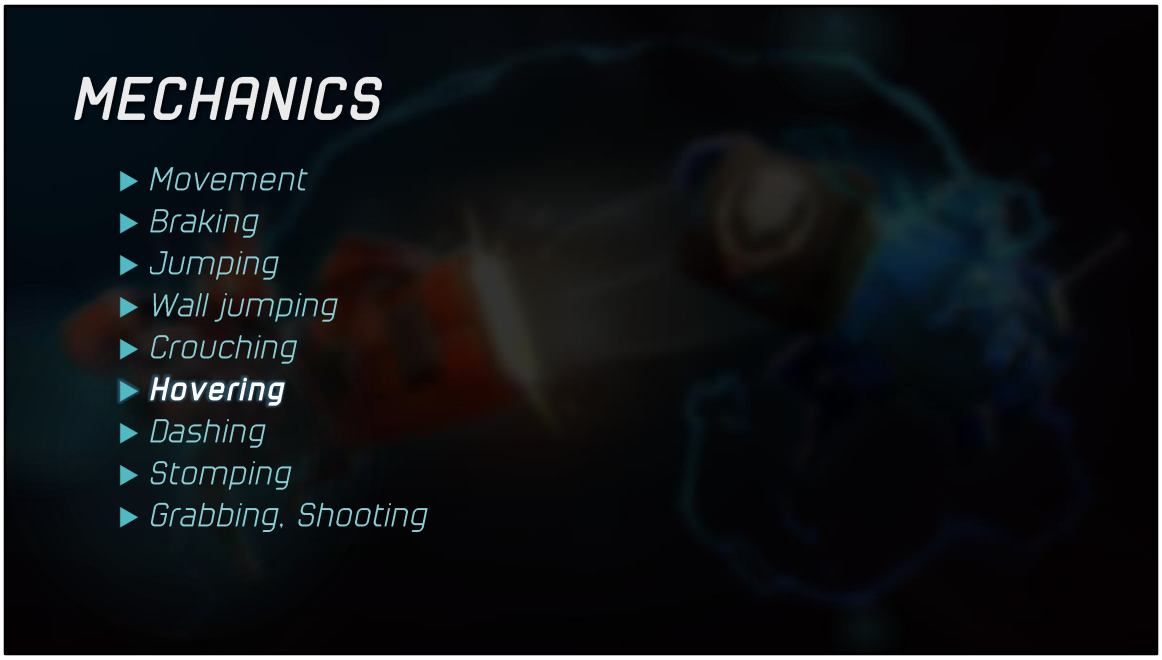
Ok, a quick interlude – crouching.



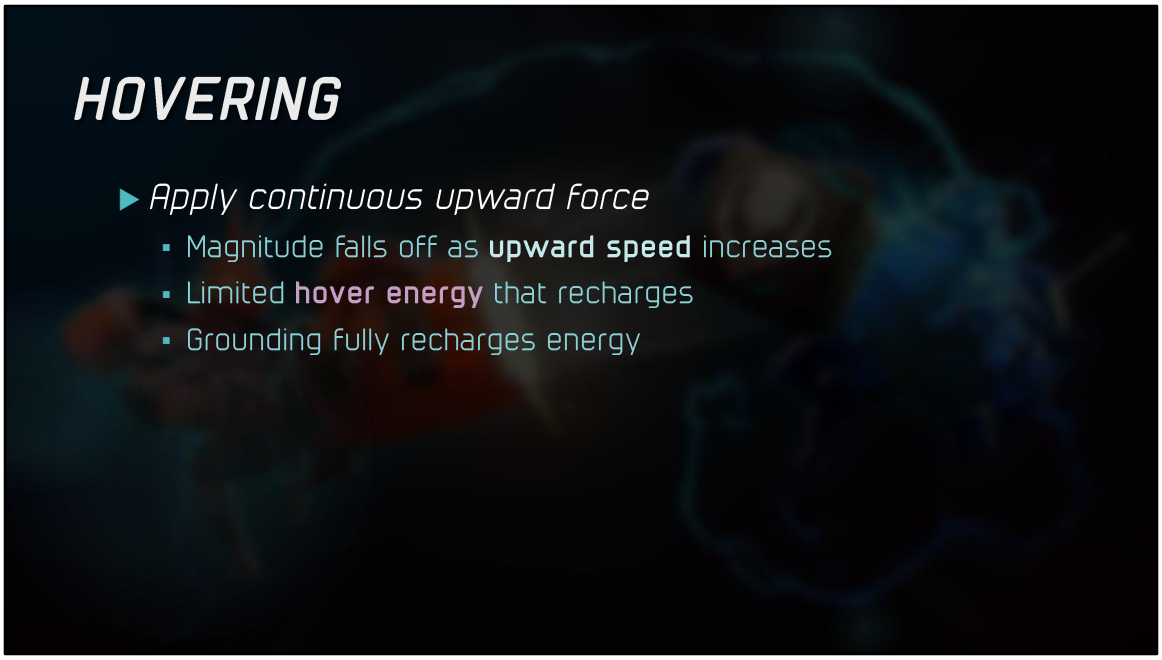
We found players really wanted to be able to celebrate when they made a good kill, or just to mess with each other.

So, we put in a crouch action – it has no practical effect on gameplay, but it **is** fun!

I mean, what fighting game is complete without teabagging?



Now, onto aerial maneuvers. First up was **hovering**, which gives players the ability to 'hang' in the air, waiting for the perfect moment to take the shot.



Hovering was achieved in a similar way to running - by adding in a force each physics step.

A few minor differences though – the hover force depends on vertical velocity. We wanted the player to hover, not propel themselves upwards really fast.

Also, there's a limited amount of 'hover energy' that is depleted while hovering, but recharges over time (or if the player touches down.)

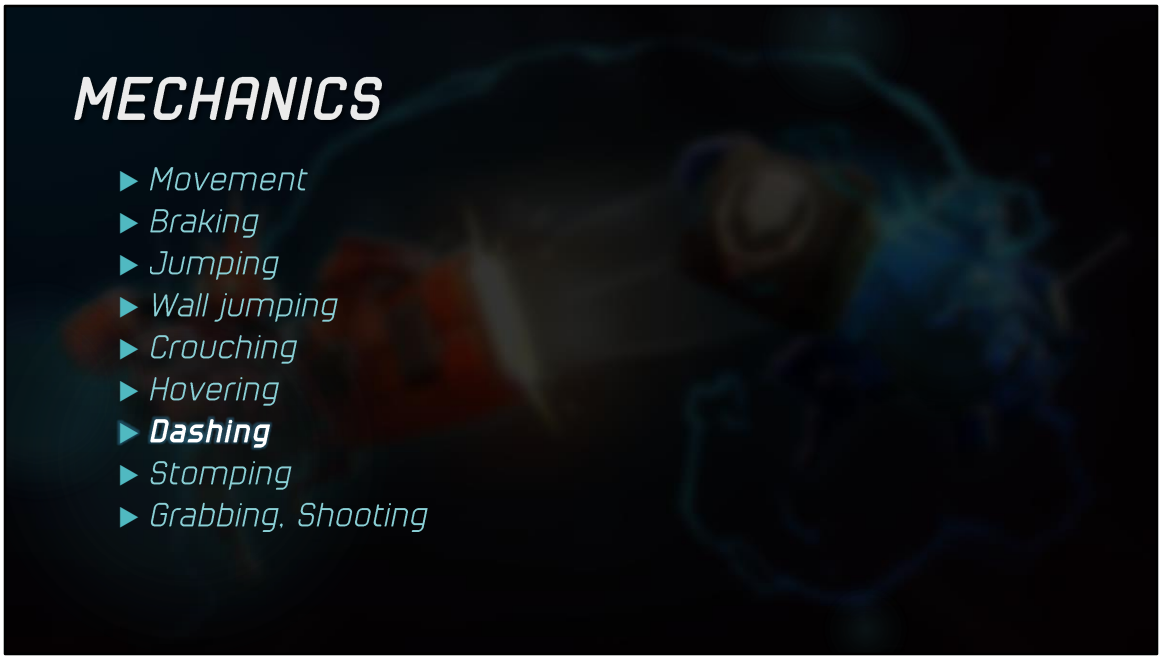
HOVERING

```
// Apply hover force according to upward speed.
upSpeed = Vector3.Dot(Body.velocity, up);
scale = UpwardSpeedFalloff.Evaluate(upSpeed);
force = up * HoverForceMax * scale;
Body.AddForce(force);
```

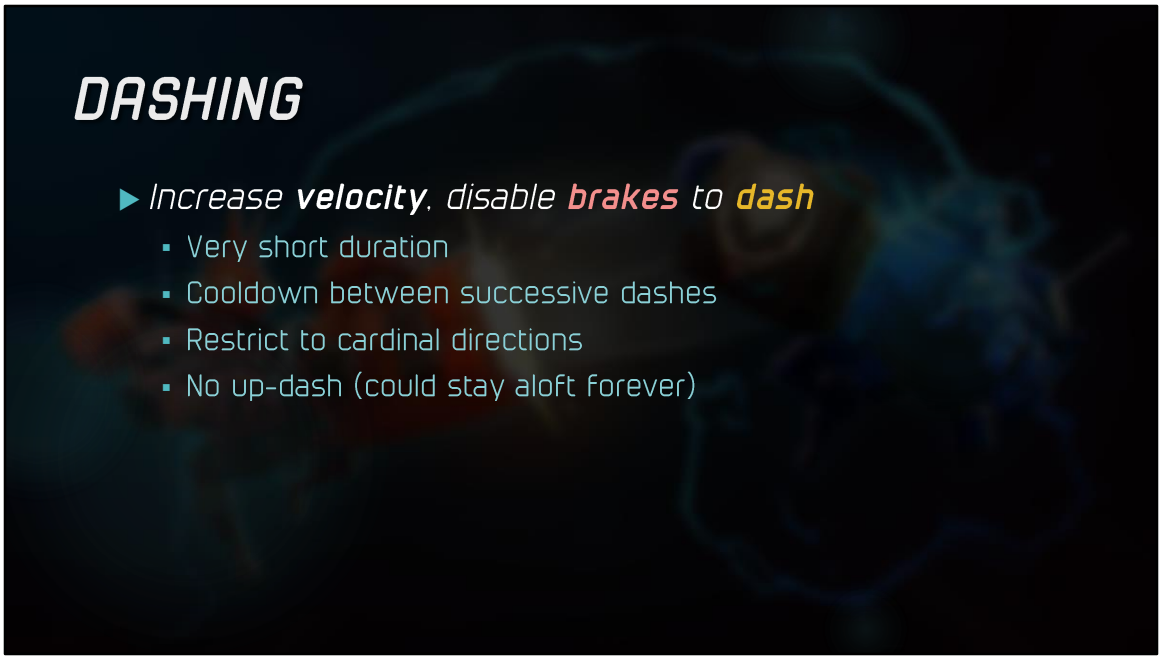
Here's the pseudocode for applying the hovering force.

Same concept as in jumping – use an animation curve to govern force according to upward speed.

And a quick demo – notice how you can use hovering to stay in the air for quite a long time if you 'pulse' it.



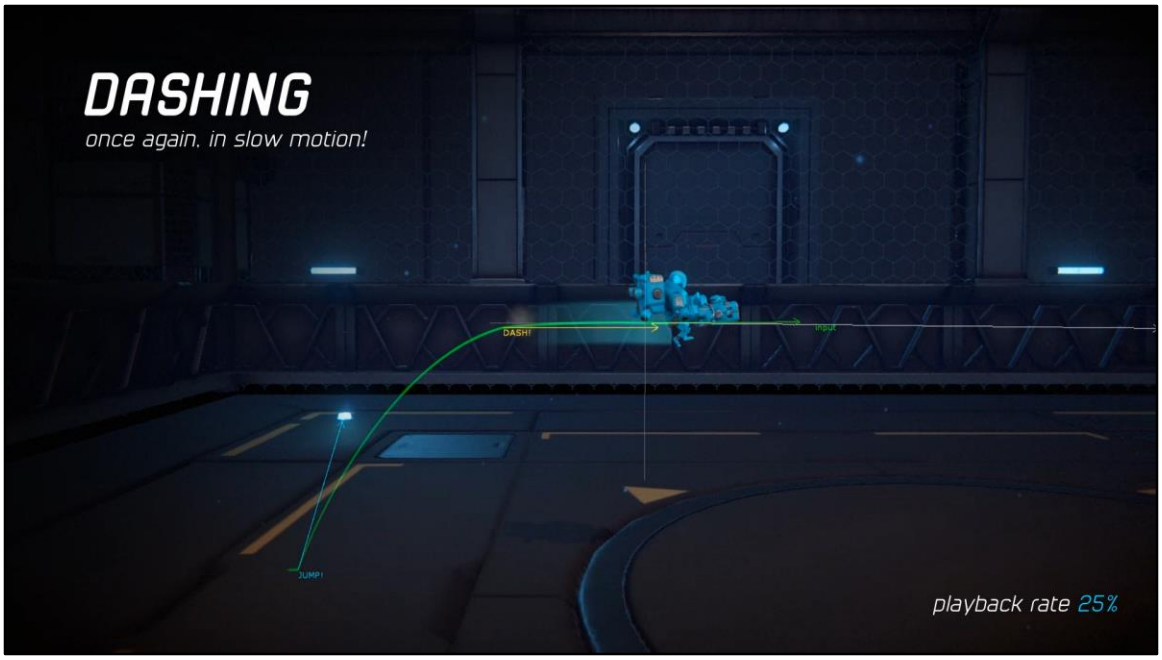
Dashing is an interesting mechanic, in that it can be applied both on the ground and in the air.



One of the main functions of dashing is dodging incoming shots, but it can also be used for maneuvering purposes.

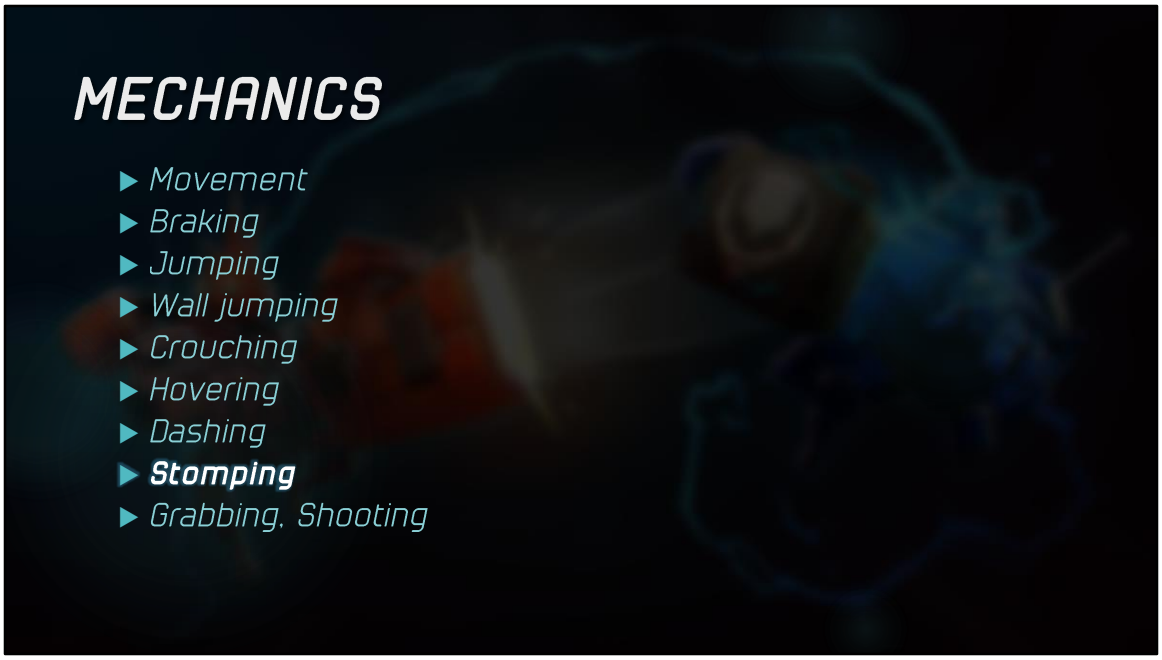
It works by bumping the player's velocity in the manner of a jump, but also briefly stops braking forces from being applied.

We experimented with allowing upward dashing, but it wasn't very fun – too similar to jumping, really.

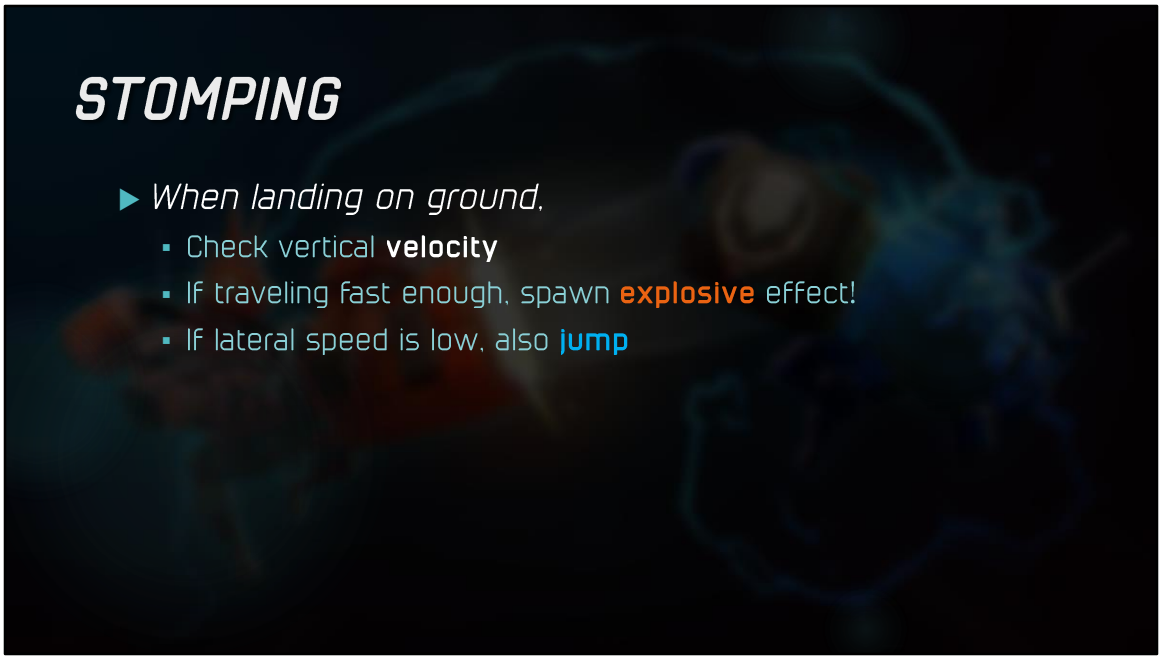


Here’s a demonstration of the dash mechanic.

It’s hard to spot at full speed, but brakes are briefly disabled during a dash. Should be easier to spot in slow motion.



Stomping is an evolution of the dash mechanic, that we stumbled upon mostly accidentally.



We found that players would try to dash down on the heads of opponents and damage them that way.

It was lots of fun, but very hard to pull off, so we added an **explosive** area effect to make stomping more satisfying.

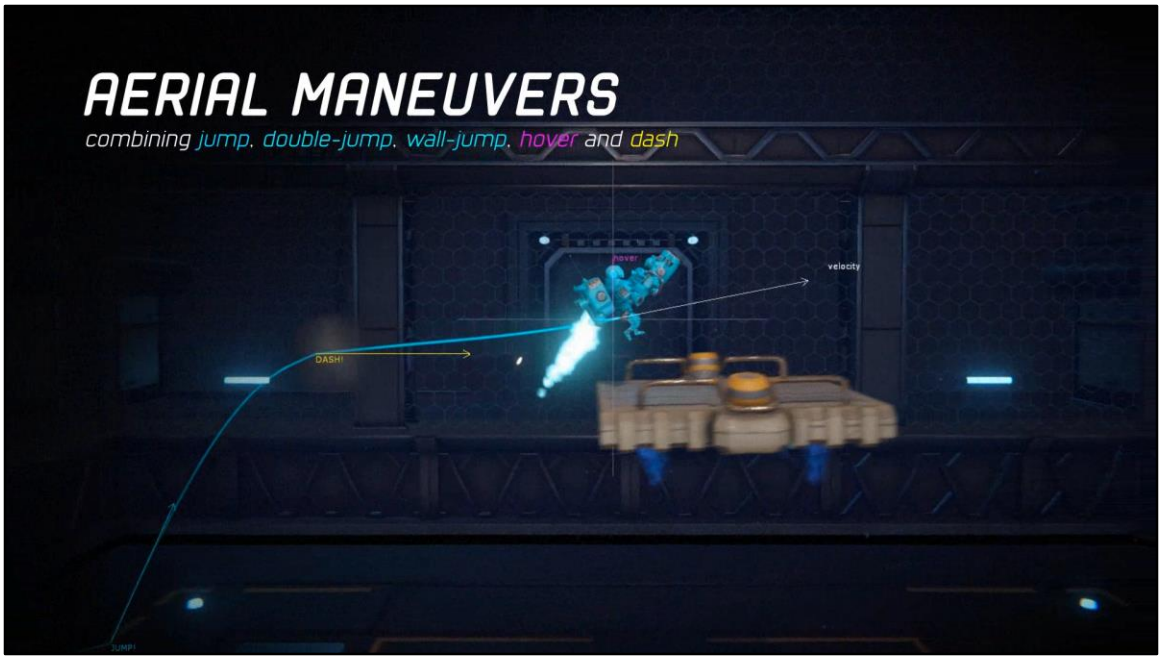
At one point I introduced a silly bug where explosion forces would apply to the dashing player as well as everything else. This made the player bounce instead of just touching down.

Wrong, but fun! So we kept it.

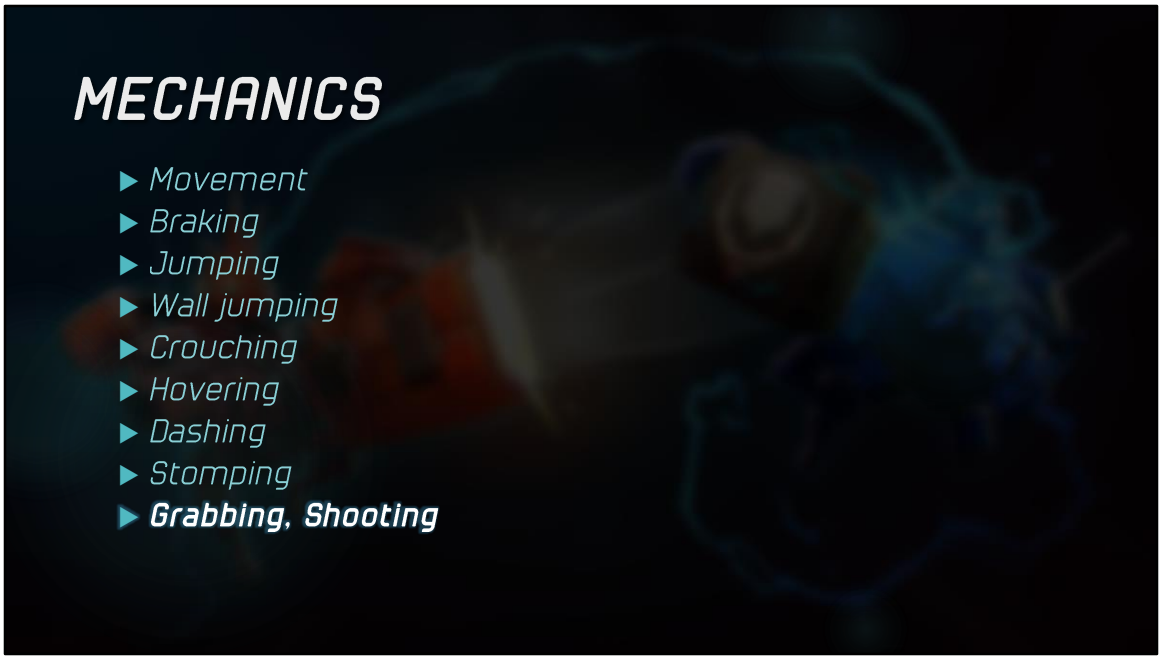
This clip shows the explosive stomp effect in action.

Can be used for denying ammo to your opponents if you're particularly mean-spirited.

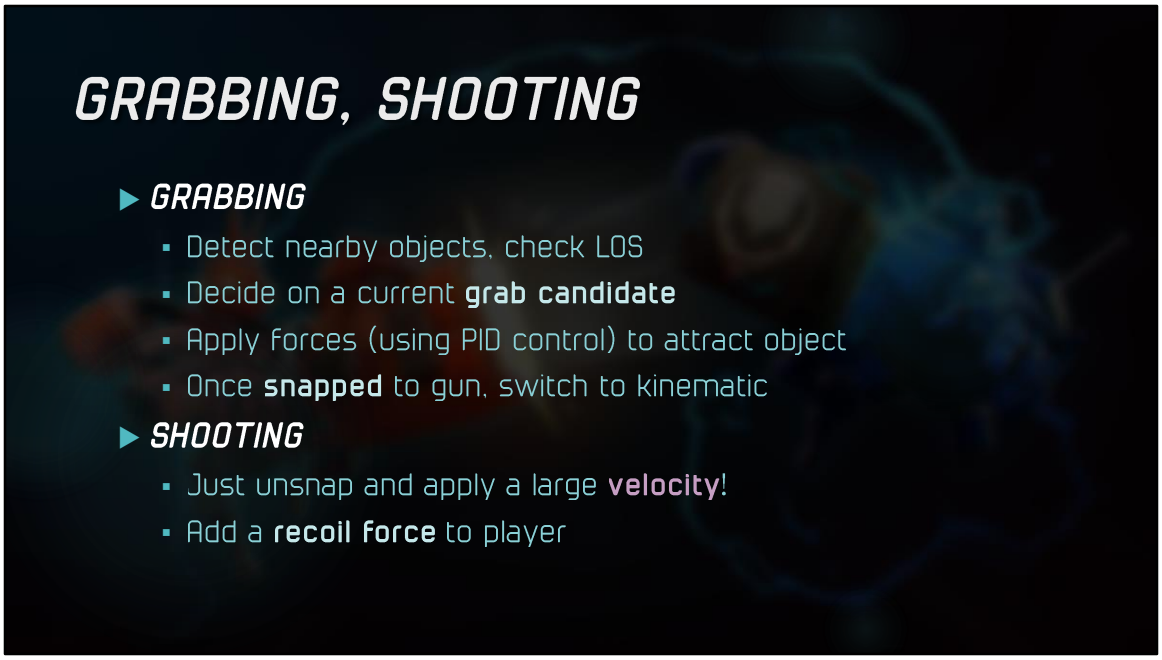
Chaining together aerial maneuvers gives players a lot of flexibility in how they traverse the environment, and gives a nice sense of **mastery**.



You can see how skilled players are able to stay aloft for quite a long time!



Last, but not least, is **grabbing** and **shooting** objects. This is really Grabity's core mechanic.



There probably isn't time to go over everything here, but the basic gist is as follows:

Grabbing
- Look for objects that are near the player, and decide which one is the 'best' one to grab at present.
- When the player tries to grab it, apply forces to **attract** the object towards their gun.
- If the object gets sufficiently close, **snap** it to the end of the gun using kinematics (directly control the object's position and orientation).

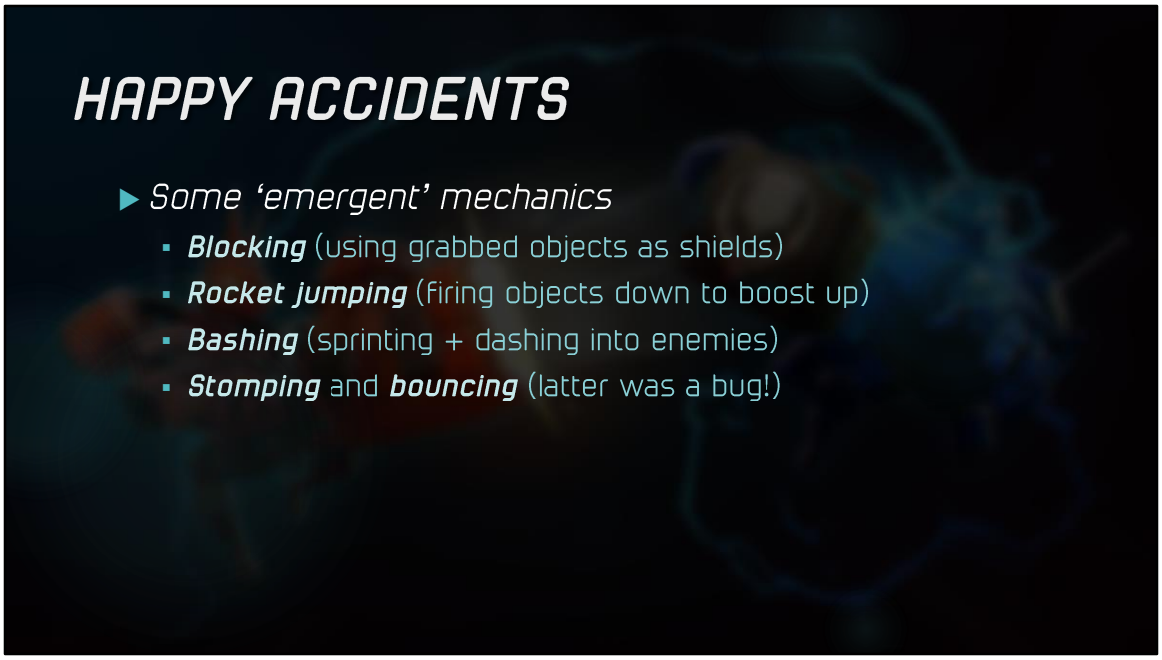
Shooting
- **Unsnap** the object (stop controlling it kinematically)
- Apply a large **velocity** away from the player's gun.
- Add a certain amount of **recoil force** to the player as well.

Here's a quick demo showing those concepts. You should be able to see the recoil in action.

Recoil is also good for '**rocket jumping**', as it turns out.

So, that pretty much covers it, really.

Putting all those mechanics together gives us character control that seems to work well in most situations, and is hopefully ‘easy to learn, but tough to master’.



Oh, a couple of things that we didn't really plan for (but turned out to be fun!)


01:46.5
TRAILER
P1
P3
P4
P2
P1
P2
P3
P4
000
000
213
025

Thanks very much for your time. We'll be at Bean Bag End all of tomorrow if you want to come say hi, or to give the game a try!

Any questions?