Real Time Strategy
At Light Speed

Julian Doherty
@madlep
juliandoherty.com

Hello

I am Julian Doherty and this is Real Time Strategy At Light Speed

I am a lead developer at Envato. We are Melbourne, Australia based company that provides online marketplaces for creatives to buy and sell digital assets.

I think we are hiring, so if moving to Australia appeals, have a chat with me after.

@ElixirMelbourne

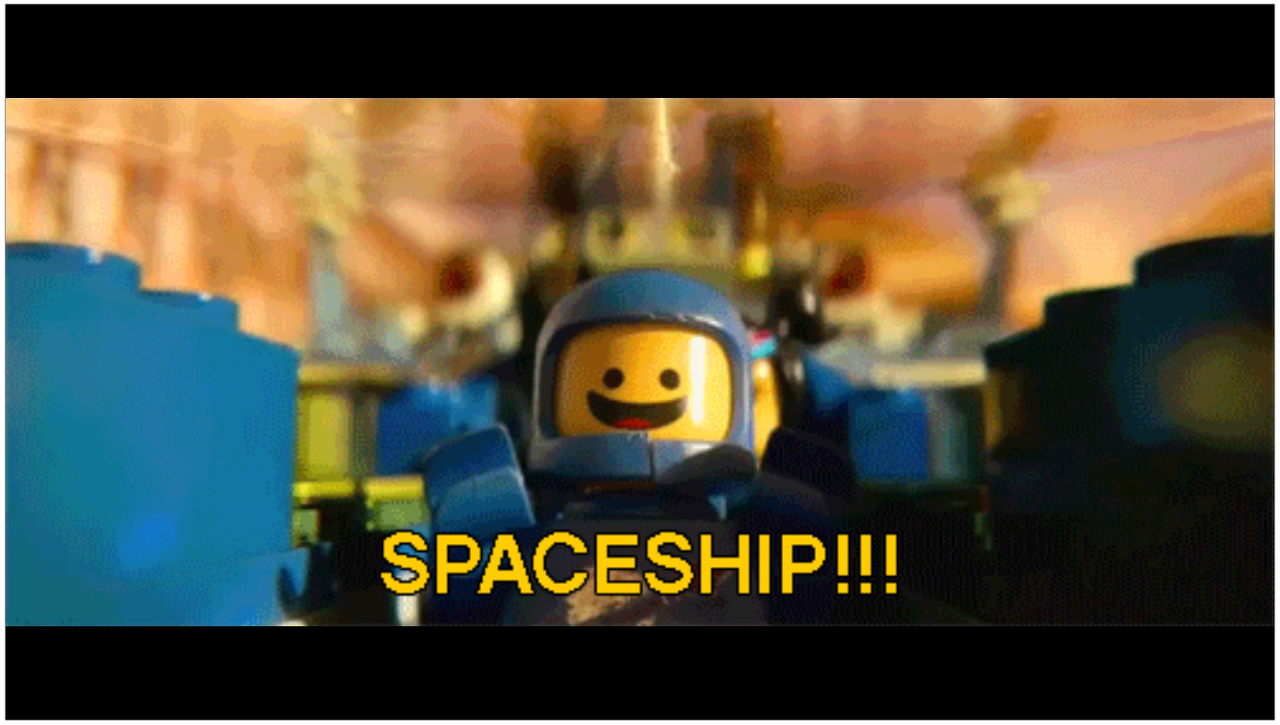
https://www.meetup.com/Elixir-Melbourne/

I also founded and run the Elixir Melbourne meetup.

If you’re in town we’d love you to drop in and maybe even do a talk

Let's talk about spaceships

So Let’s talk about spaceships



not that kind.

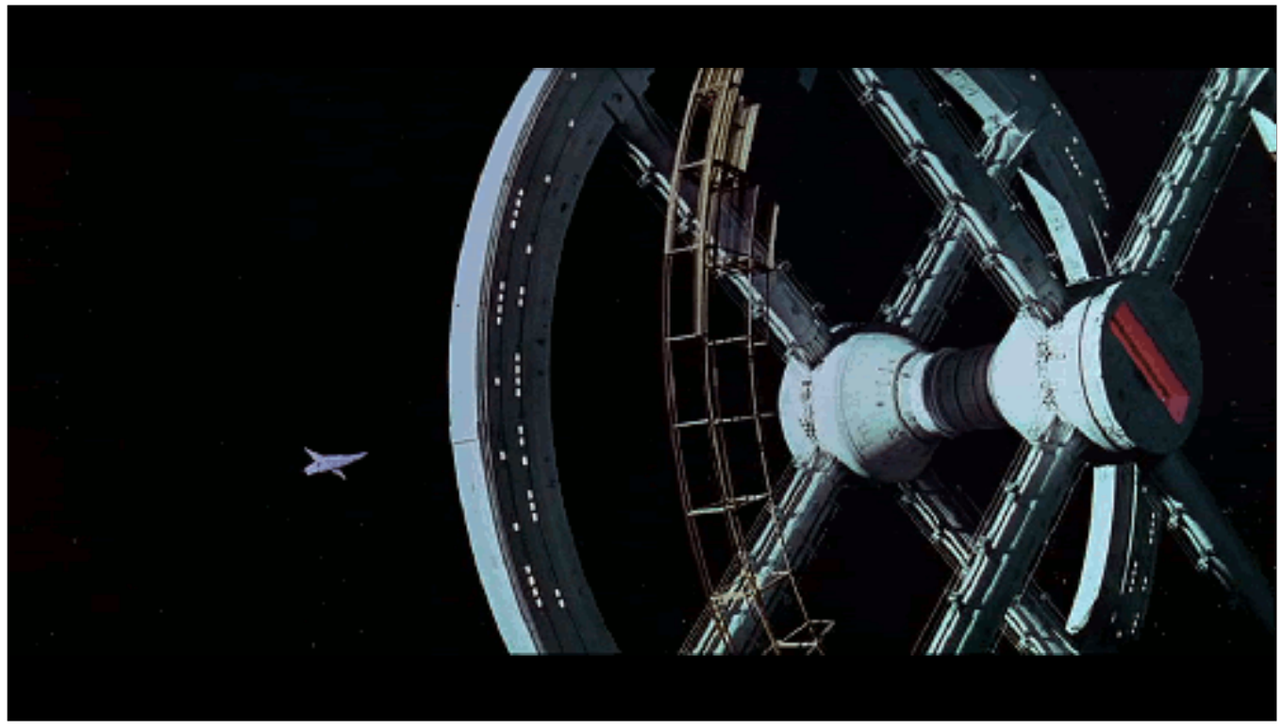
And Not that kind

Not that kind either.

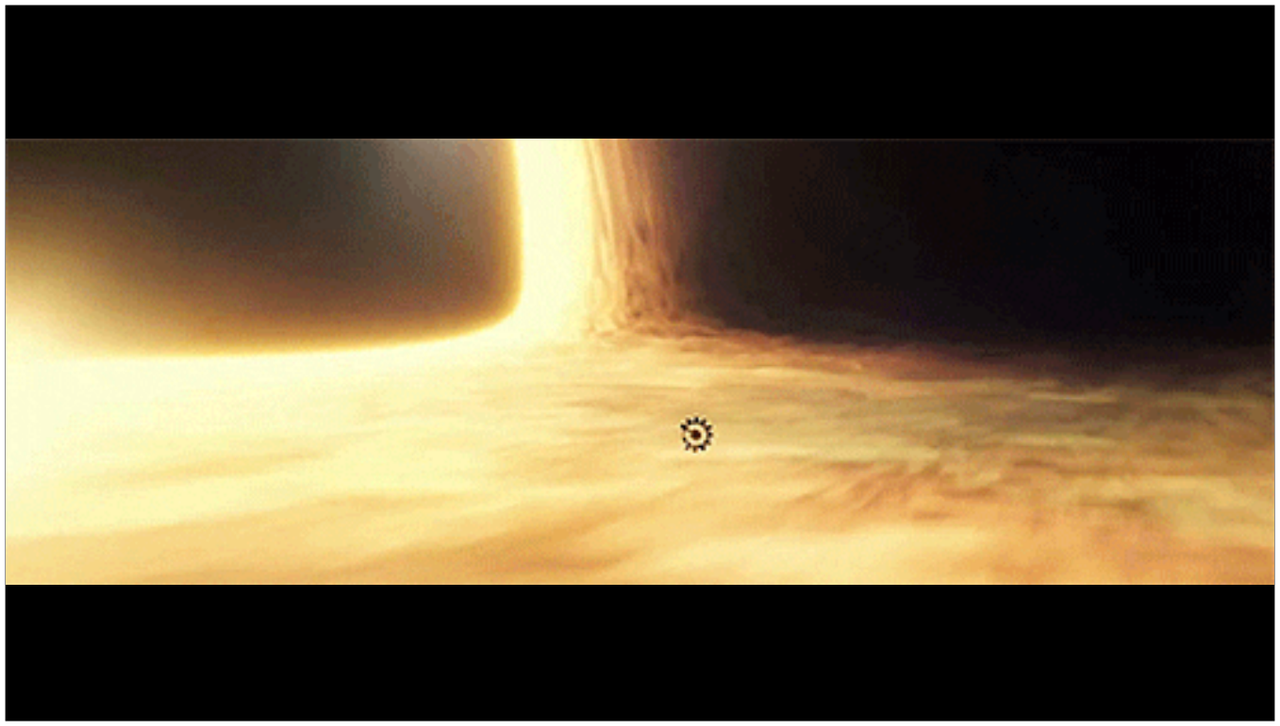
Spaceships more like...

We want spaceships that are…

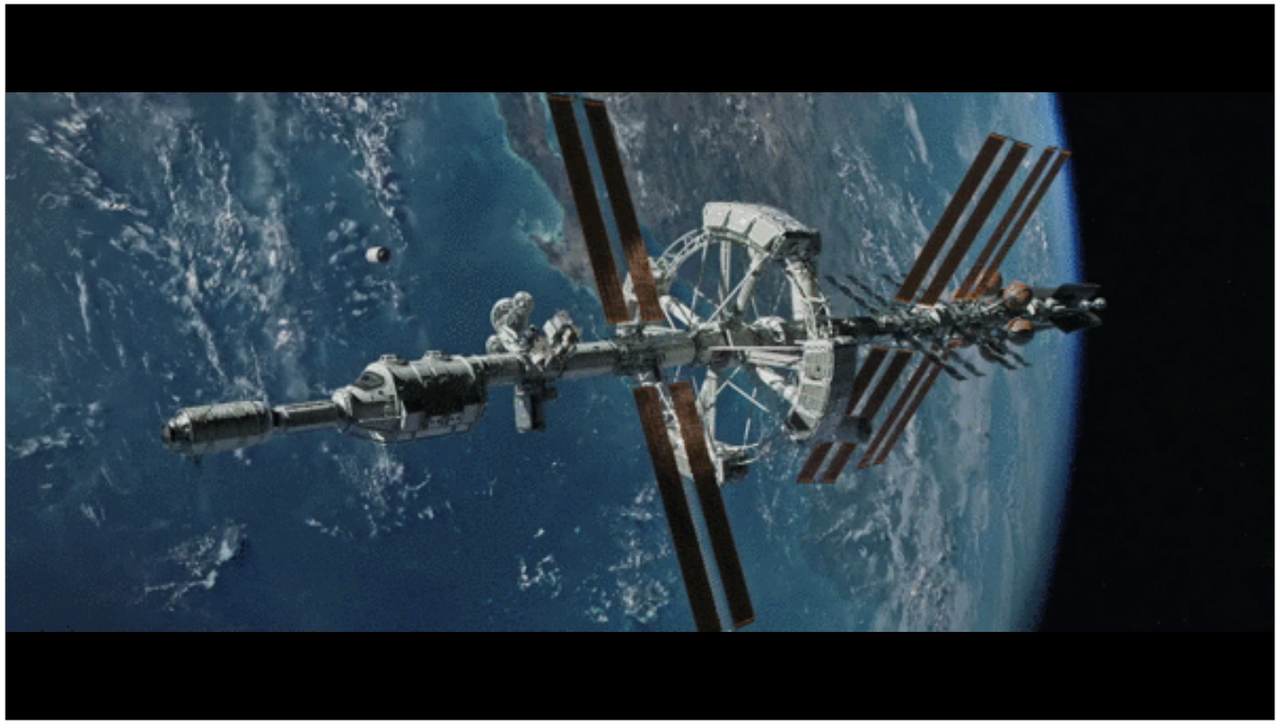
Sort of like this.



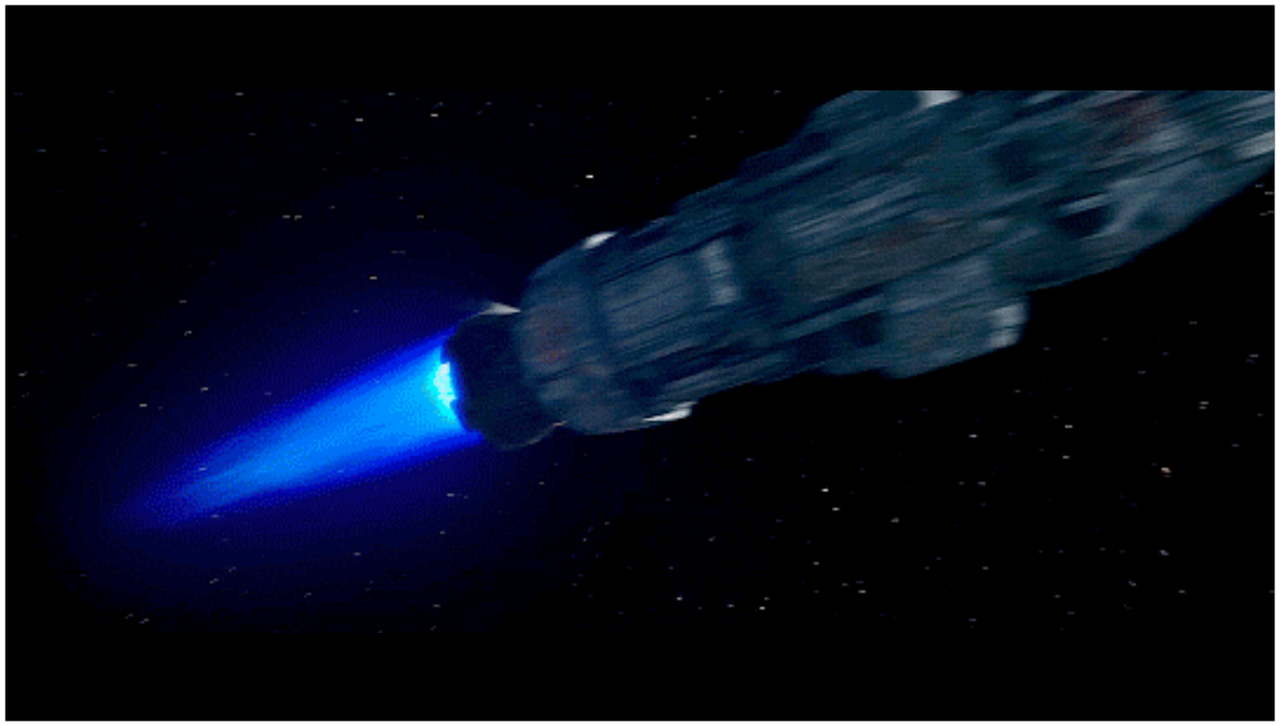
And more like this



And this



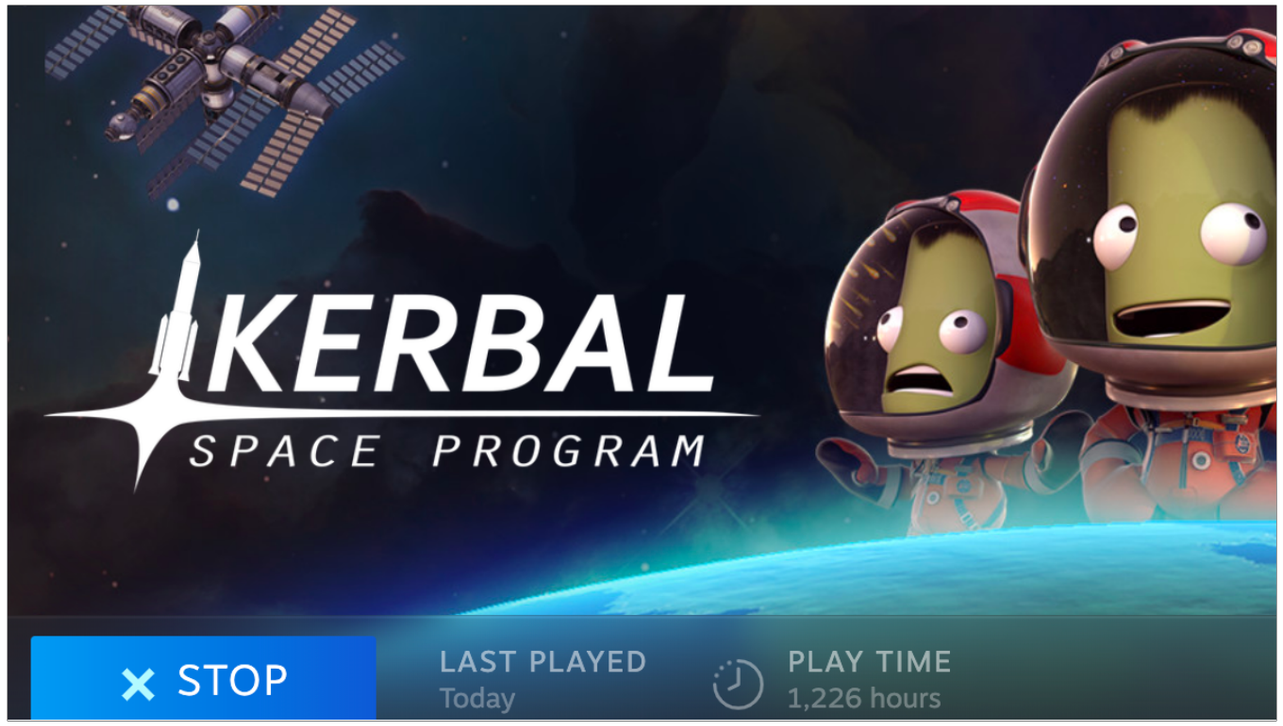
And even more like this.



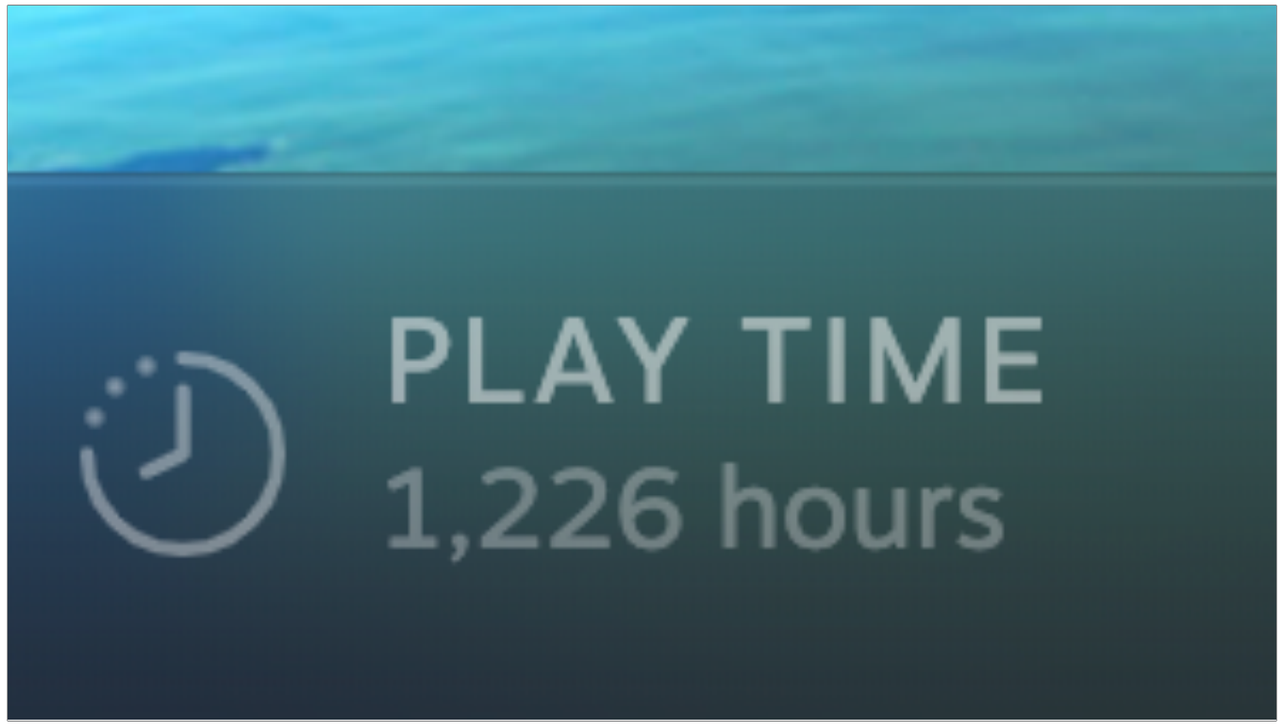
And something totally like this.
No, that is not playing in reverse.

Spaceships!

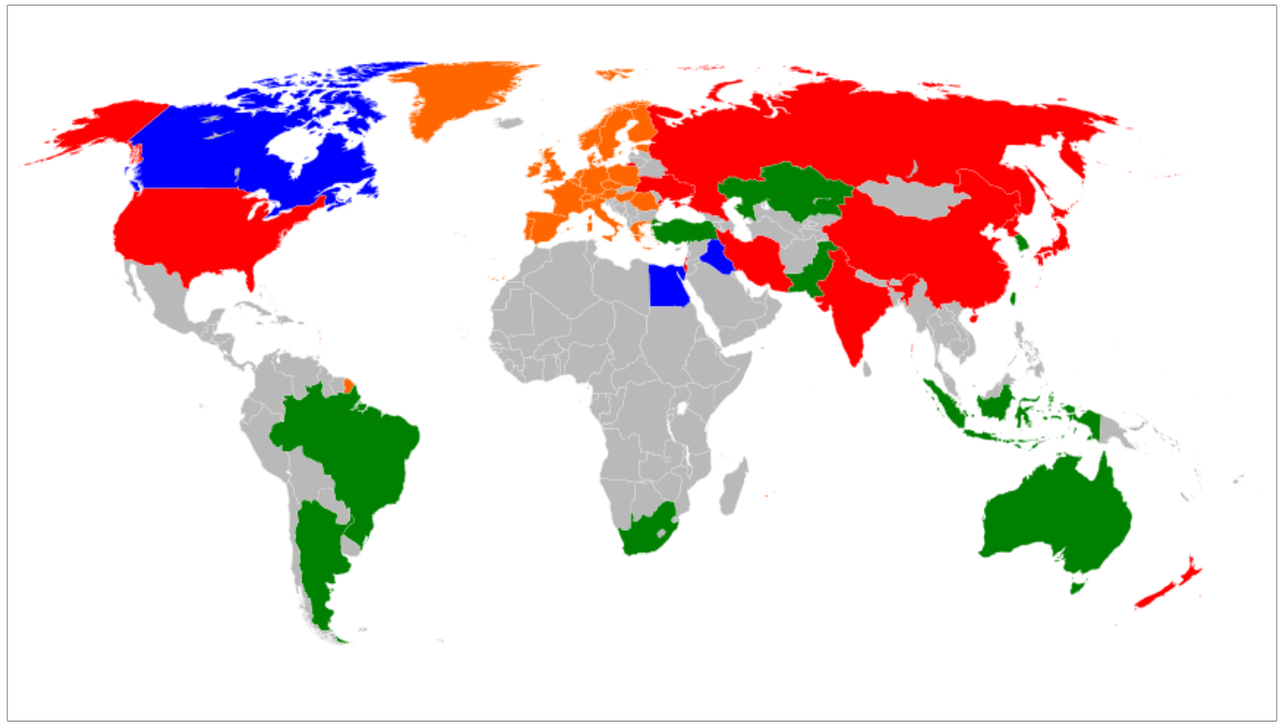
I am well qualified to talk about spaceships



I've a bit of Kerbal Space Program (pic of Steam hours played).

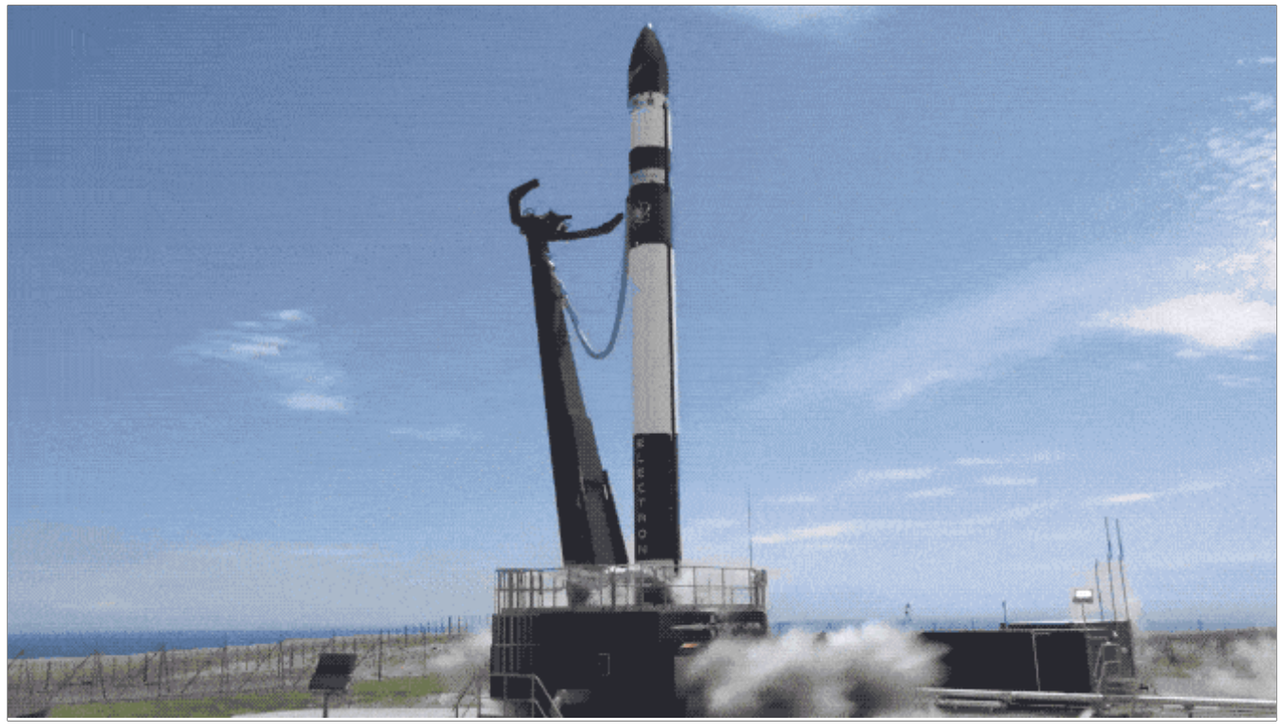


like… quite… a bit



And, I’m originally from New Zealand, one of only a handful of countries with current space launch capability. That’s the red and orange ones

That’s us in the bottom right. I’m Impressed we show up on this map. Usually they crop us out



This is Rocketlab, a New Zealand based launch company. They have grabbed about 6% of all launches worldwide since they started flying commercially a couple of years ago.

I can take some credit for some of that, as I bought a pair of commemorative "It's Business Time" socks to celebrate their first commercial launch, which was named in honour of one of New Zealand's other great success stories...

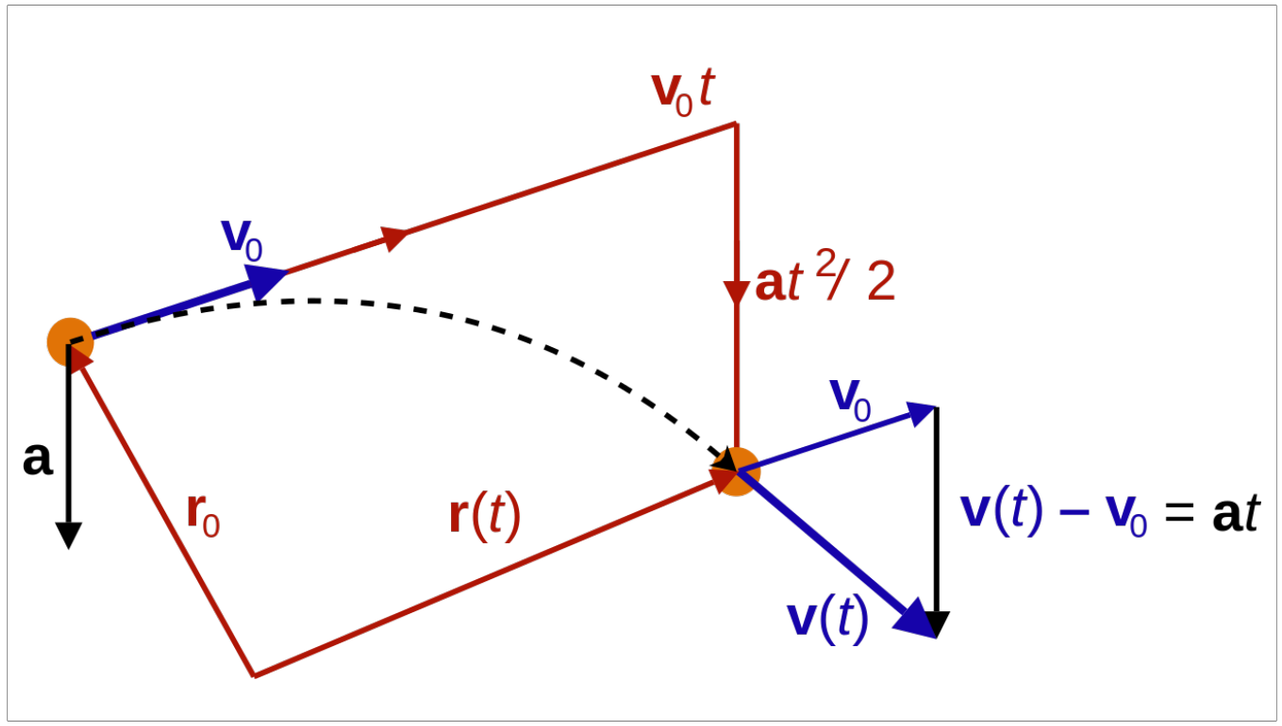
Flight Of The Conchords

I currently live in Australia though, which has no space program to speak of.

Although Australia did once launch a rocket into orbit for the UK government that looked like a giant tube of lipstick.

Hard Sci-Fi

So, I'm a big fan of hard sci-fi. That's the subgenre that tries to stay as close to reality and scientific accuracy as possible, but using more advanced tech than we have now.

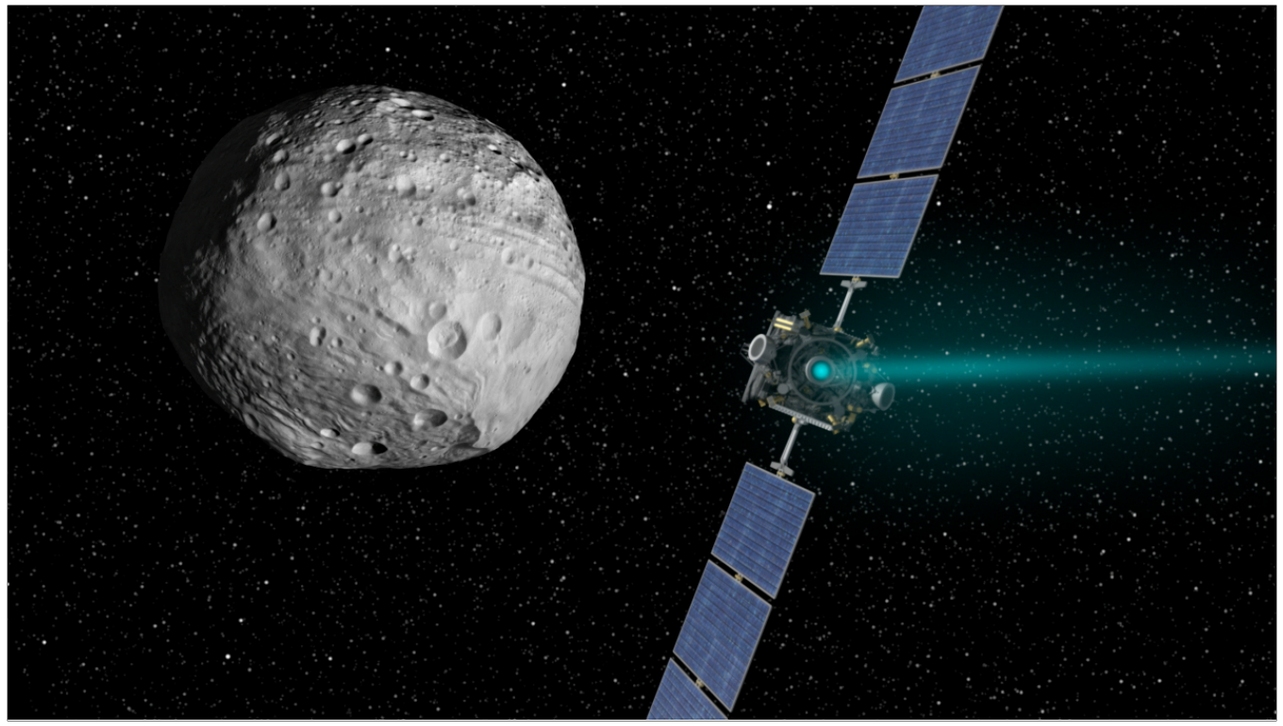


So that mostly means plain old Newtonian physics. You want to go somewhere? You need to somehow push yourself.

You want to stop? You need to point your engine backwards to push in the opposite direction

And that's how rockets we have today work.

we have engines with high thrust that can push really hard



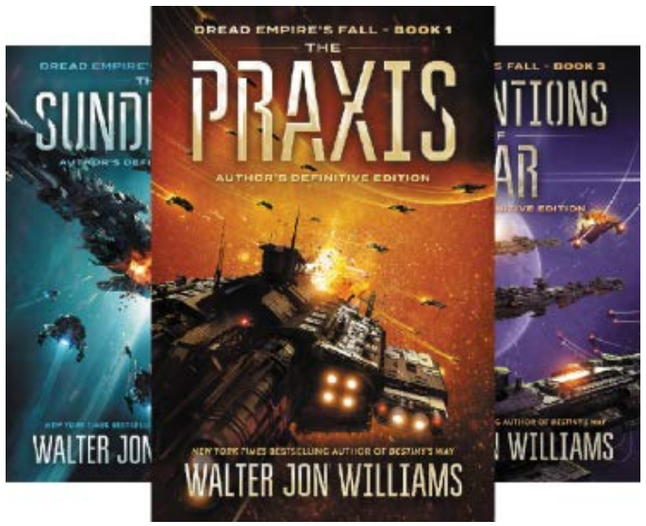
and we have engines with high efficiency that can push for a really long time

```
thrust? && efficiency?
# false
```

But we don't have engines that can do both. So high thrust, and high efficiency.

One of the hand-wavey things some hard sci-fi does do is solve this by having the concept of a "torchship". Something with infeasibly good performance. Where you can burn your engine at high thrust - like enough to crush your organs for weeks or months.

It’s a plot device that lets us have a plausible setting where the whole solar system is colonised and people can zoom around and have adventures where it doesn't take years or decades to get places.



(TL;DR - lots of spaceships hurling torpedos at each other)

A great example of this is the Dread Empire's Fall series by Walter Jon Williams.

TL;DR - lots of spaceships hurling torpedos at each other.

(TL;DR - lots of spaceships hurling torpedos at each other)

And, The Expanse book series by James S. A. Corey, which has also been made into a show on Amazon Prime Video.

(TL;DR - lots of spaceships hurling torpedos at each other)

In both these, the authors take pains to point out how big space is

The authors take pains to point out how big space is

Erlang. The Movie.

IN SPACE!

So you’ve all seen Erlang The Movie.

But what if we did it…

IN SPACE!

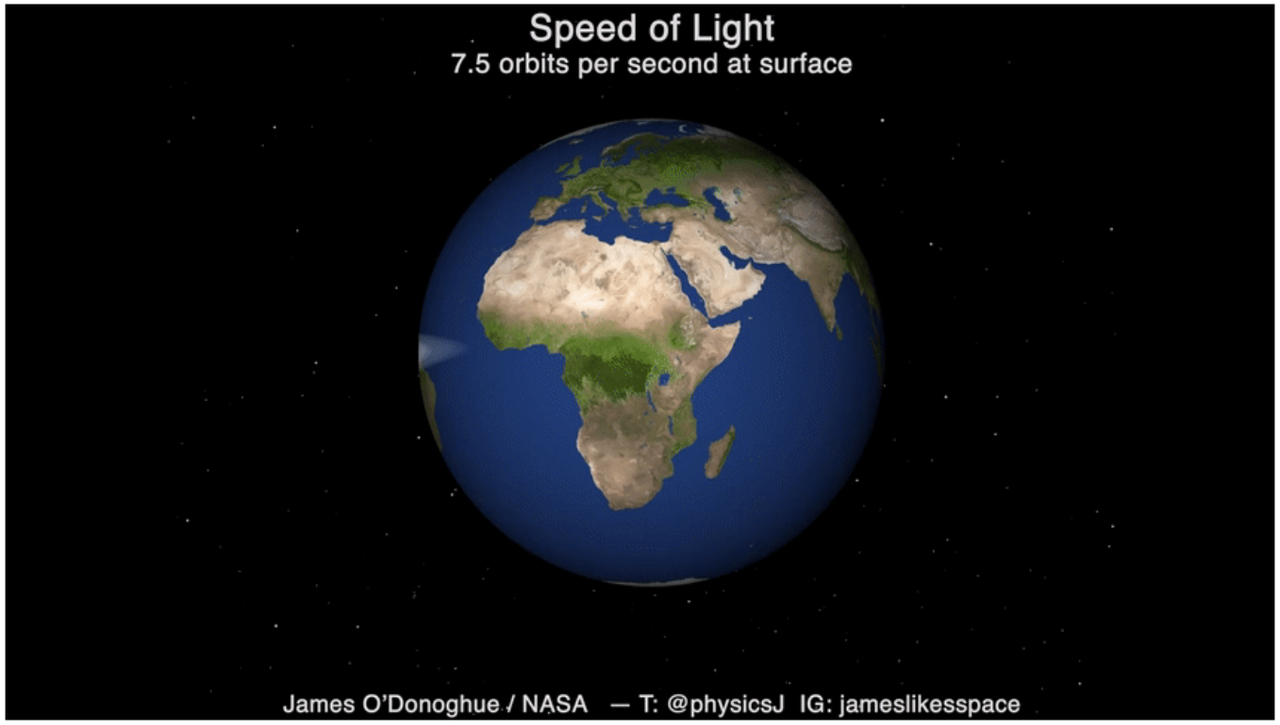
“Hello Mike!”

...
...
... waiting
... still waiting

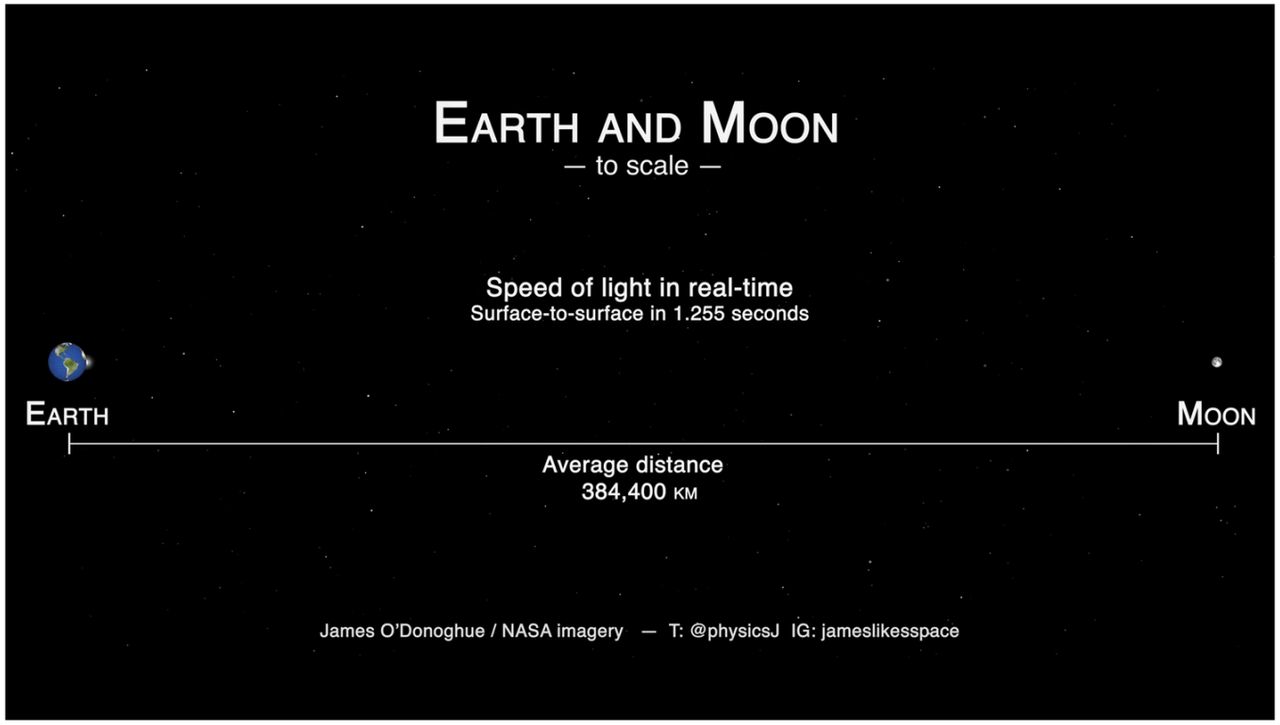
“Hello Joe!”

It would go something like this:
- phone rings
- “Hello Mike”
- …
- …
- Waiting
- Still waiting
- AND THEN 45 minutes later, Hello Joe!

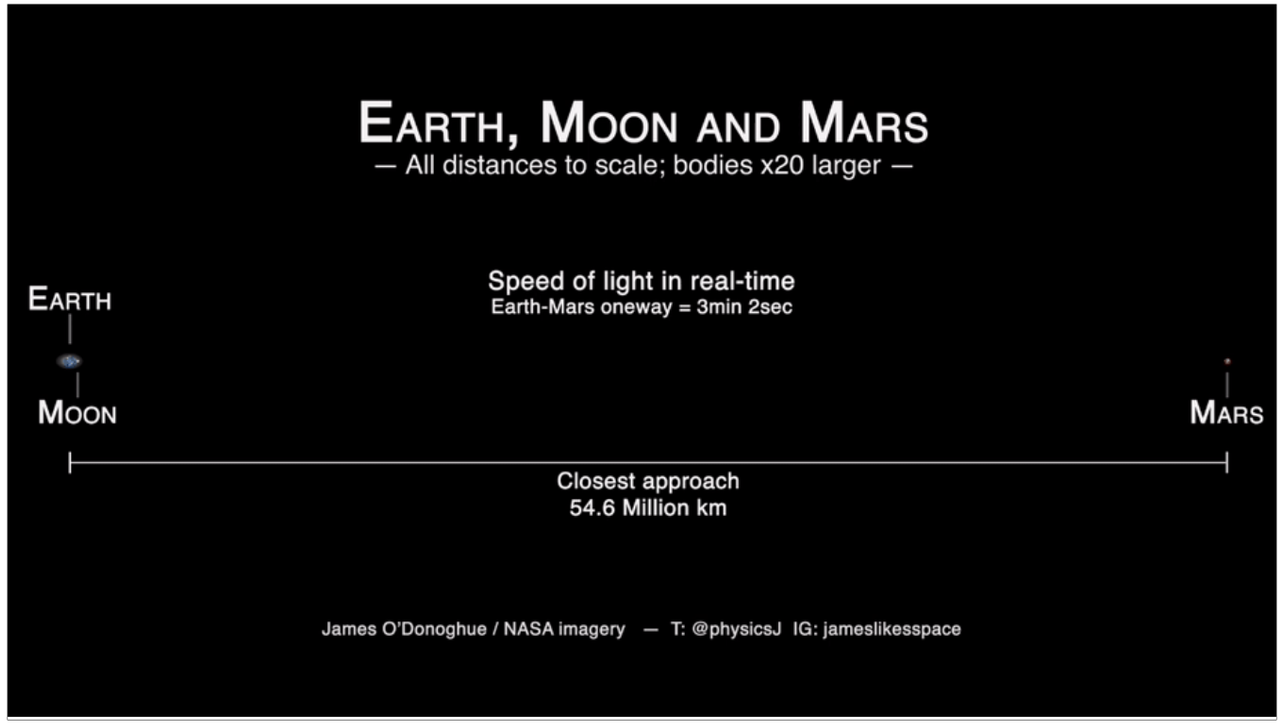
The reason for that is the speed of light



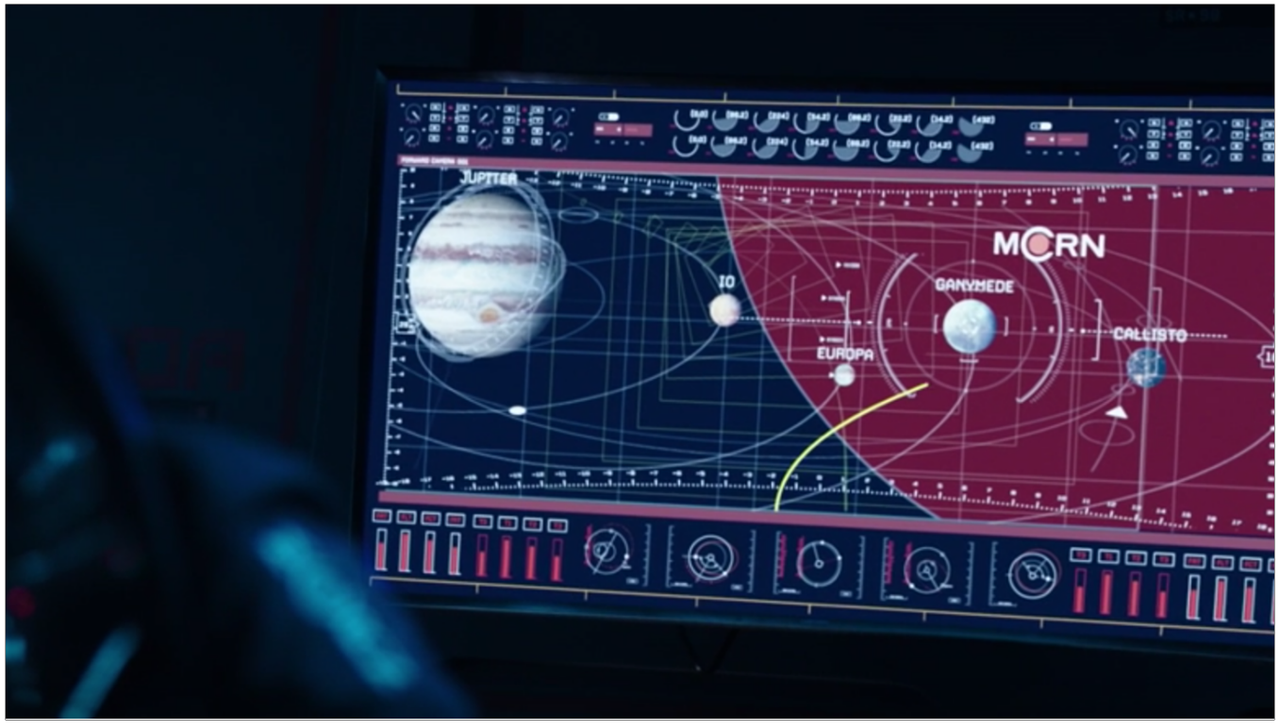
The reason for this is light. Now light is fast. It could go around the earth 7 times in a second



But not that fast. It takes a second to get to the moon.



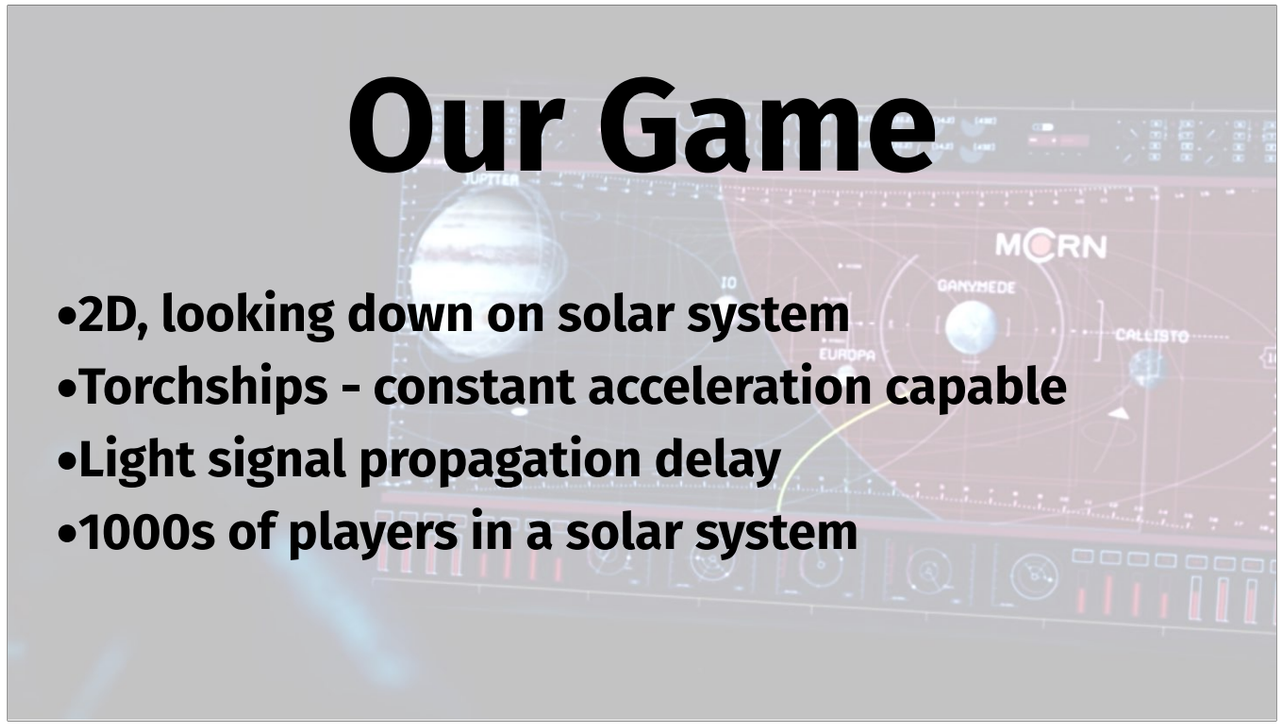
And by the time you get to interplanetary scale it's downright slow. 3 minutes to Mars. MINIMUM. Usually more like 20 minutes depending on where the planets are in their orbits. Outer planets - longer. Hours. Yeah, we're not gonna sit here and watch that all. I've only got 20 minutes.



This delay adds a huge amount of drama and suspense. If you're in combat with a ship 10 light seconds away, you need to anticipate what they'll be doing 10 seconds from now and plan strategies on that, based on information you can see that is already 10 seconds old. I thought this could make for a game with really fun gameplay elements.

So how do we build that?

Our Game



- 2D, looking down on solar system
- Torchships - constant acceleration capable
- Light signal propagation delay
- 1000s of players in a solar system

"The first 90% of features account for 90% of the code, the next 90% of features account for 900% of the code."

(apologies to Tom Cargill)

If you've worked in any kind of online product that is iterating and growing, you'll have seen this.

The first 90%

Code starts out fresh and well factored, but over the years as more requirements are added, things get... clunky.

Imagine some spaceship actor code like this...

```
defmodule EnemyShip do
  defstruct [
    :health, :position, :velocity, :acceleration, :target
  ]

  def update(ship, world) do
    ship
    |> avoid_collision(world.all_entities)
    |> select_target(world.ship_entities)
    |> fire_if_target_in_range()
    |> point_at_target()
    |> accelerate_toward_target()
  end
end
```

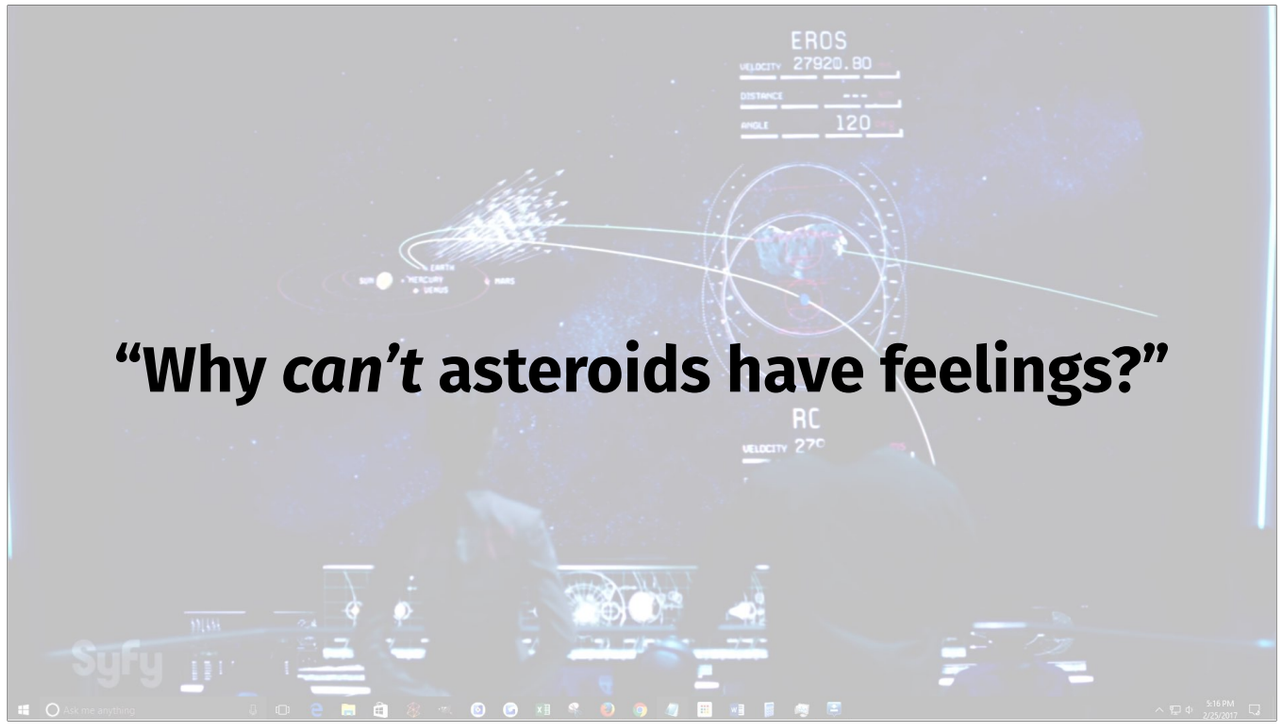
So imagine you have a spaceship game object implemented like this. This could be wrapped up in a GenServer or something. We've got health, movement data, a target we want to shoot at, and an update function that runs each game tick to move all that along.

And imagine some asteroid actor code like this...

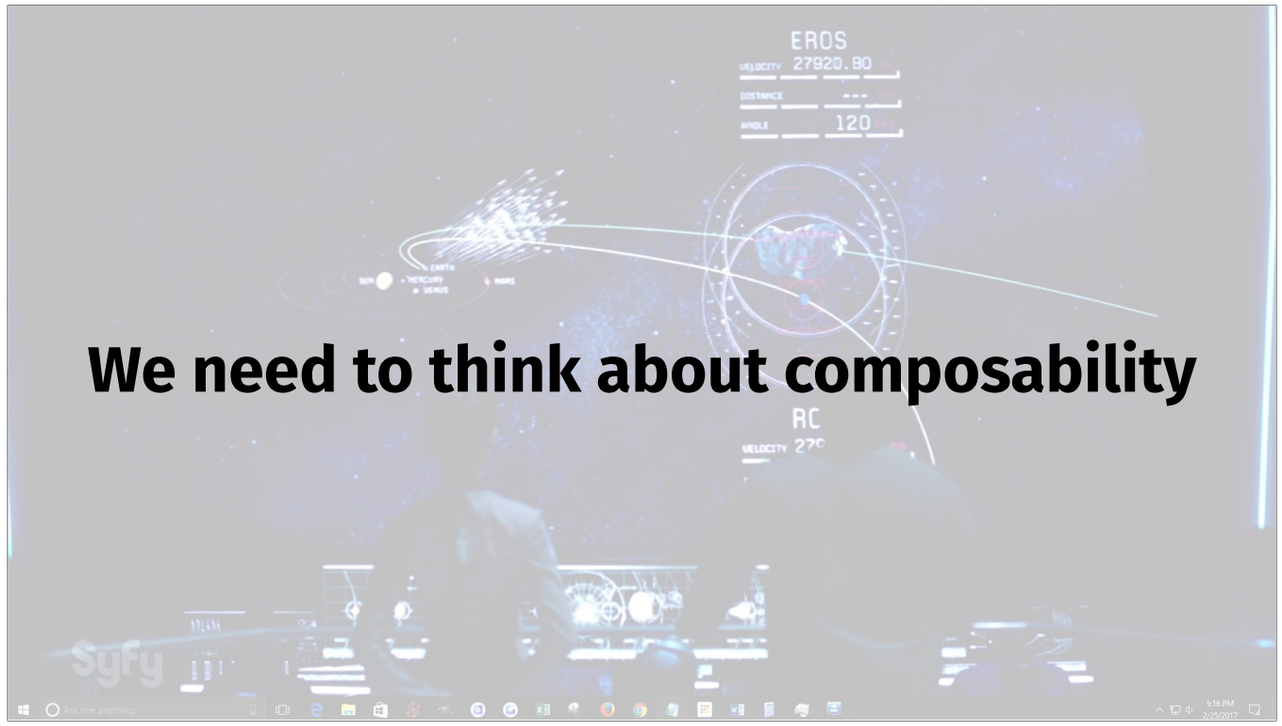
```
defmodule Asteroid do
  defstruct[
    :name, :size, :orbit_parameters, :position
  ]

  def update(asteroid, world) do
    asteroid
    |> calculate_position_from_orbit_params()
  end
end
```

And something representing asteroids. They run on rails in fixed orbits, so movement is easier to calculate, as it’s always deterministic.

Which is _great_ at first, we've got a working thing!



but before long, someone comes up with something like "well why _can't_ asteroids have feelings and be able to move, and have health and be smart enough to know if it's being targeted and move out of the way, and then decide it's annoyed with a planet, and fly off and smash it!"
Don't laugh, there was a plot line in The Expanse which was basically that...
But then one of the software devs says "well, you _can't_ do that! Asteroids only use the on-rails orbital mechanics for movement, and no AI! Duh!” And the product person is like *glare*. And says “Dude. Just ship it.”
So another if statement is born and added to the monolith



We need to think about composability

So it comes down to that old thing of how to decompose problems down into manageable chunks. You want to be able to look at something through a narrow enough lens to be able to understand it, but at the same time be able to compose your chunks in a way that lets you reason about the system as a whole. And you want to do this in a way that lets you build on this and add new features to your code after you've shipped the first version.

ECS

Entity, Component, System

Enter ECS. Entity. Component. System.

In game development, this is a commonly used pattern for structuring logic in real time games that need the performance to update their internal state every frame, while still maintaining the ability to iterate and tweak gameplay.

So lets break it down

Entities

- Coarse grained game objects
- *Literally* just an identifier

```
my_asteroid = UUID.uuid4()
# THAT'S IT! Nothing else!
```

Every course grained object in the game is an entity. Spaceships, planets, torpedos, space stations, sad asteroids, whatever. An entity is _just_ an identifier. Usually an integer or a UUID or something. That's it. Maybe name-string to help with debugging perhaps, but that's it.

Components

- Key/values associated with an entity
- Just value data. No logic

Entities on their own are pretty useless. So we can add arbitrary components to them.

These are data values that can be composed to define what properties entities have.

These are totally flexible. Any component can be added to any entity. Entities specifically DO NOT have type. Their behaviour is based solely on components they contain.

Components

Initially we have something like...

```
my_asteroid_components = [
  name: "Eros",
  size: [width: 11_200, length: 34_400],
  orbital_parameters: %OrbitalParameters{
    orbiting: sun_entity,
    eccentricity: 0.22,
    semi_major_axis: 218_000_000_000,
    argument_of_periapsis: 178.82 * :math.pi / 180,
    mean_anomaly: 47.24 * :math.pi / 180
  },
  position: { x, y }
]
```

So to build our sad asteroid, first we have this…

Components

Then we can change it to something like...

```
my_asteroid_components = [
  name: "Eros",
  size: {width: 34_400, height: 11_200},
  health: 100,
  position: { x, y },
  velocity: { v_x, v_y },
  acceleration: { a_x, a_y },
  target_tracking: %TargetTracking{
    target_priority: [razorback_entity, earth_entity, venus_entity]
  },
  collision_avoidance: %CollisionAvoidance{
    strategy: :randomly_move_sideways,
    avoid_anything_bigger_than: :massive_generation_ship
  }
]
```

Then we change it to this…
Want an asteroid with feelings that decides to move itself and smash into a planet? Sure! Add components for movement and for target tracking!
Want it to be smart enough to not get rammed by a 2 mile long spaceship trying to disable it? Sure! Add that too.
The nice thing is, we can do this per entity. Doesn't have to be global. You don’t have to alter ALL asteroids.
AND we can do it at runtime. Remove components, add them, whatever

This is great!

- Can add features easily
- Not constrained by initial design
- Still don’t have any logic yet though!

Systems

- Logic for ONE THING that *just* deals with entities with components it cares about
- Reads/updates component state
- Runs globally within game context, *not* per actor

The last piece of the puzzle are systems. These contain logic that just deal with entities ONLY with components it cares about. So the movement system does movement, the sad feeling system handles sad feelings - which might involve changes to how movement works too.

Systems

```elixir
defmodule MovementSystem do
  # ...
  def update(system_state) do
    for {entity, components = %{
      position: p,
      velocity: v,
      acceleration: a
    } } <- system_state do
      new_vel = Vector2.add(v, a)
      new_pos = Vector2.add(p, v)
      {entity, %{components | position: new_pos, velocity: new_vel} }
    end
  end
end
```

Our movement system could look something like this. We have system_state containing the data for all entities it needs. When it runs, for each entity, we grab the position, velocity, acceleration, then do some basic physics to move our things and update the state for our whole system

This is JUST movement. We don’t have to deal with AI or collision detection or input or whatever getting coupled in here.

Entities, Components, Systems

Data doesn’t have to live together

The thing to note though, is they don’t run per game object. So not an an actor or a struct, or an object (if we were OO). But across ALL instances of that type component, within some context, like a single game. Sort of like a columnar database.

Data doesn’t have to live together

Components can live where ever

So the example we saw there was using a list of maps to store all the components for the systems, but it doesn’t have to.
One way to get a performance win is by making the data you are working with smaller, If we’re just working with a component of an entity, rather than ALL the data for an entity in our logic, we have less data. In a low level language like c, that means memory access is done in a more cache friendly way,
in Elixir it means smaller messages and less memory copying to send them. And probably more cache friendly too. But like, you should be dropping to Rust in a NIF if you’re optimising at that level.

Components can live where ever

In Elixir, this means no shared state, processes, and message passing

So for Elixir that means:
no shared state,
processes and
message passing

Let’s make our system concurrent

```elixir
defmodule MovementSystem do
  use GenServer

  def init(game_id) do
    ECS.EventBus.subscribe(game_id, component_types: [:position, :acceleration, :velocity])
    Process.send_after(self(), :do_update, 1000)
    state = build_state()
    {:ok, state}
  end

  def handle_info(:do_update, state) do
    new_state = move_everything(state)
    Process.send_after(self(), :do_update, 1000)
    {:noreply, new_state}
  end

  def handle_info({:ecs_events, events}, state) do
    new_state = events |> Enum.reduce(state, &handle_ecs_event/2)
    {:noreply, new_state}
  end

  defp handle_ecs_event(%PutEntityEvent{entity_id: id}, state) do
    add_entity(id, state)
  end

  defp handle_ecs_event(%PutComponentEvent{entity_id: id, type: type, data: data}, state) do
    add_component_to_entity(id, type, data, state)
  end
  # ...
end
```

So implementing that as a gen server would look something like this. We keep state in the process, and maintain it in an event sourced way by passing game events as messages.

Now state could be a plain elixir term, or an ets table, or external database, or whatever. It could even spawn child processes to spread work further. The important thing is it’s a black box to the outside world.

That’s a wall of text. Let’s break it down.

A more “Elixir” system

```
def init(game_id) do
  ECS.EventBus.subscribe(game_id, component_types:
    [:position, :acceleration, :velocity])
  Process.send_after(self(), :do_update, 1000)
  state = build_state()
  {:ok, state}
end
```

First, we subscribe to an event bus for the game.
That could be based on pubsub or Kafka or whatever. In this case, I just implemented it by hand as a genserver. The important bit is that you give it some way of specifying components you’re interested in as part of the subscription. So if any of those components change, and the entity has all of them, ECS will send our system a message to be notified about it, and it can update it’s local state based on that.
So if you squint hard enough, we’re doing event sourcing, and this is a projection

A more “Elixir” system

```elixir
def handle_info(:do_update, state) do
  new_state = move_everything(state)
  Process.send_after(self(), :do_update, 1000)
  {:noreply, new_state}
end
```

Then we send a message to ourselves to update state every so often by using Process.send_after and repeat

A more “Elixir” system

```elixir
def handle_info({:ecs_events, events}, state) do
  new_state = events |> Enum.reduce(state, &handle_ecs_event/2)
  {:noreply, new_state}
end

defp handle_ecs_event(%PutEntityEvent{entity_id: id}, state) do
  add_entity(id, state)
end

defp handle_ecs_event(%PutComponentEvent{entity_id: id, type: type, data: data}, state) do
  add_component_to_entity(id, type, data, state)
end
```

And we implement handle_info to receive the messages passed to us.

So on each event, we update our local state to replicate what is going on out in the world.

Cool... what about the light speed bit?

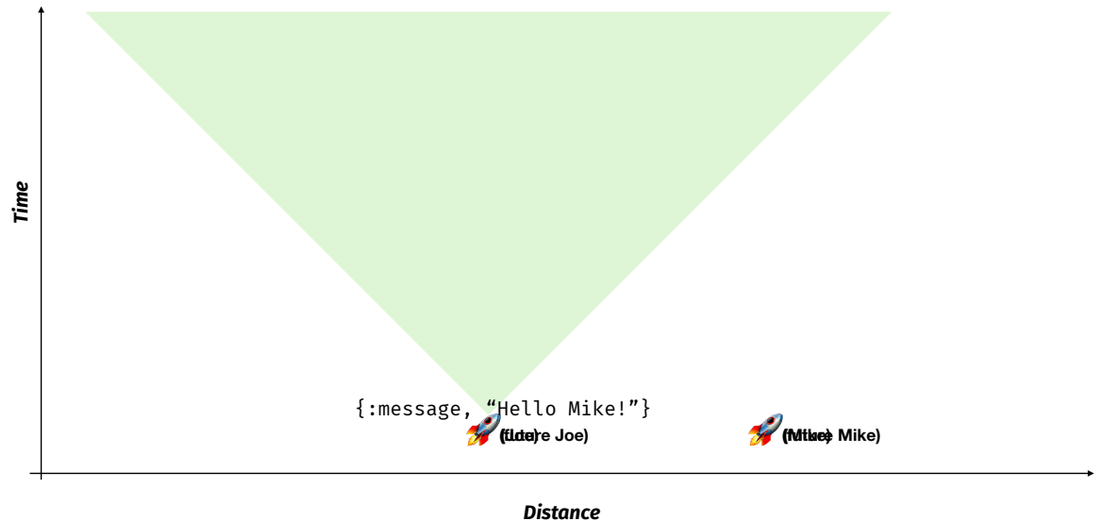
What is time?

So what is time exactly? The uber driver we had last night had some ideas about it. I think the gist of it was that the past or present don’t exist, only the future. No… Hang on. Only the present. Or the past. But tarot readers, can sense your aura, and they can use your aura to tell the future - which doesn’t actually exist, but it does, but tarot readers are just doing it for money, and they should use their talents to benefit humanity.
Nod. Pause
Dude still charged us like $40 to drive across town, so so much for using his talent to benefit humanity instead of making money off it

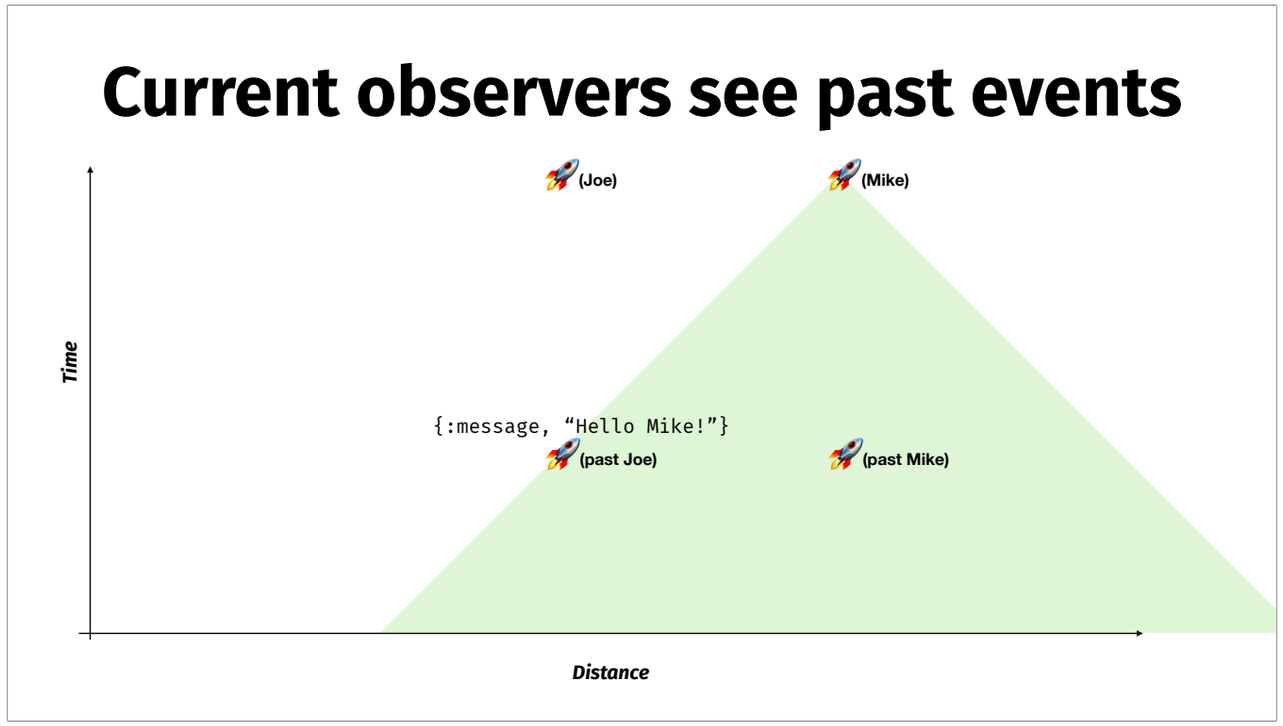
What is time?

So say we’ve got two spaceships. And one of them sends a message out into space. That message moves at the speed of light, and when it reaches the other one, they have observed it occurring. My 6 year old came out the other day with “time has to exist, otherwise everything would happen all at once”. He’s big on science videos on YouTube - but that’s pretty much it. It’s really just another dimension to put events into.

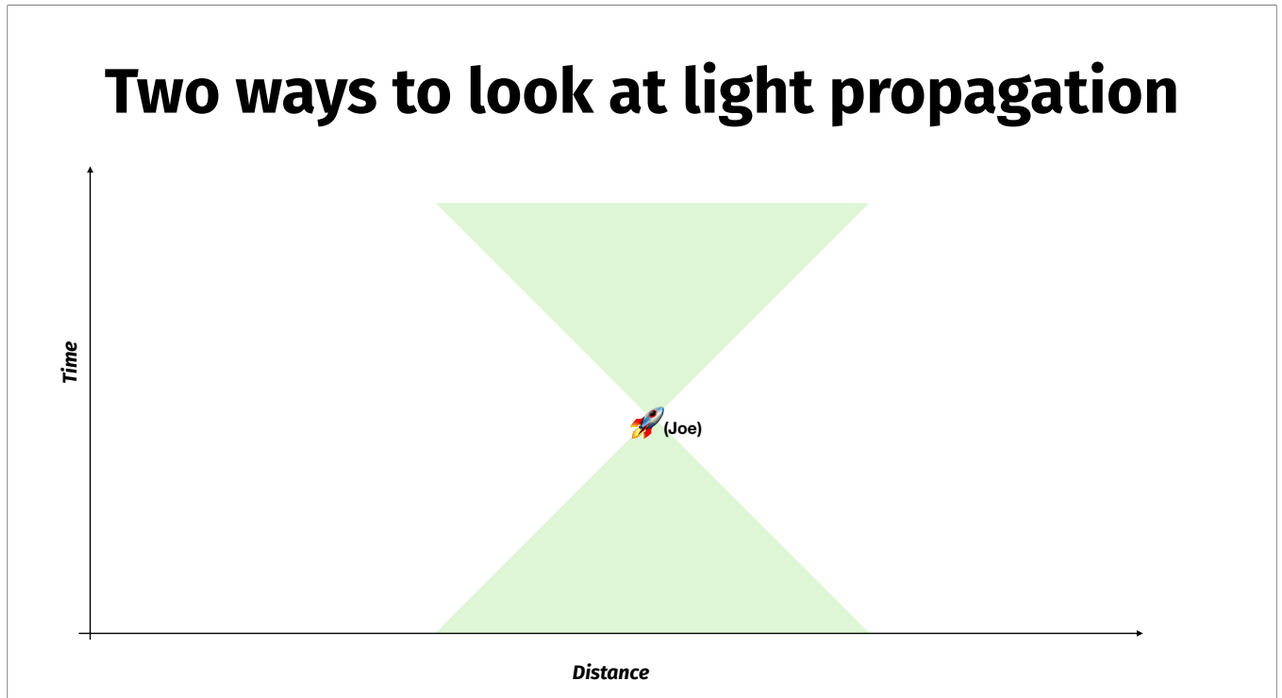
Future observers see current events



So say we had a universe with just one dimension for space, and one for time. So we have 2D spacetime. We’d get a spacetime diagram like this. When Joes sends a message, it creates a “light cone” of points in spacetime that can observe the event. Everything on the edge of the triangle sees the event occurring, everything inside has seen it already, and everything outside can’t see it. So everything moves forward through time. Joe and Mike stay still, but the message is moving outwards at the speed of light. When Mike hits the light cone for it, that’s when he observes the event.



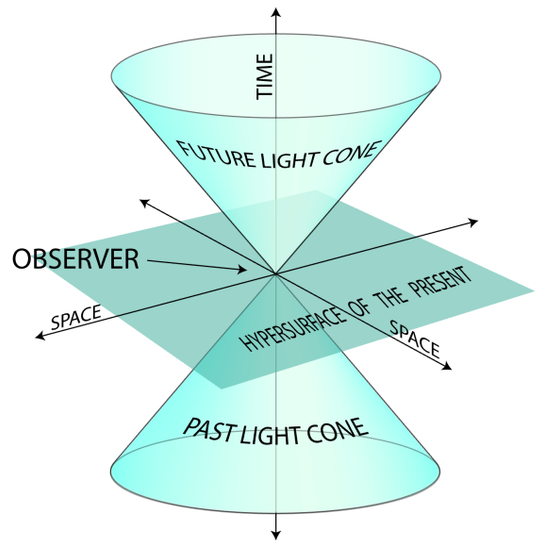
And we can look at it the other way too. If we go from the point of Mike, we can project a light cone backwards to “query” what events in spacetime we can see.



So, for any point in spacetime, we get two light cones - one for events we are creating projecting forward through spacetime, the other containing all the events backwards through space time that we can observe

In 1D space, that’s a triangle

In 2d Space, this is a “light cone”



in 2D spacetime, we get a 3D cone instead.

If we were thinking in OUR 3D world, we get a hypercone, which you can imagine as a ball expanding outwards from a point at the speed of light. It gets trickier to intuitively imagine how that looks as a 4D thing though. Our brains don’t work that way.

These are just more ECS systems!

So we’ve got two potential strategies for how we can do this. And with our ECS set up, we can just implement that as systems without messing with existing game logic we’ve got.

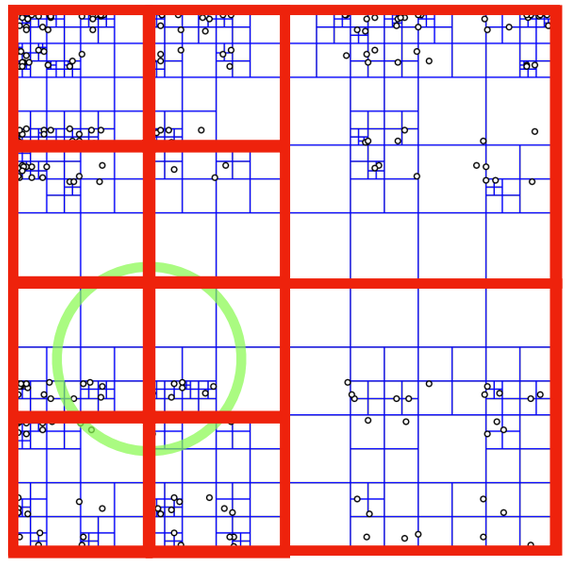
Future Light Cone

- `FuturePropagationSystem`
- Tracks current position of entities that can observe (🚀 Mike)
- Tracks current position of entities that can emit propagating events
- Spawns a `Task` process when a propagating event occurs
- Task starts at event origin, and expands outwards at the speed of light
- Each iteration through, query current position of entities that can observe
- And publish a `PutComponentEvent` on them containing the propagated original event

Future Light Cone

- **Problem:** How can we efficiently track and find all the entities that could observe an event, even when they are moving around in 2D space?
- **Answer:** Use a QuadTree

QuadTrees



So a quad tree is a data structure that stores everything in a 2D bounding box or quad. And recursively divides it along each axis into smaller and smaller quads in a tree structure until there is a set max number of elements in each “quad” has it’s own box. To find things in it, you check the top level. Then divide it into 4. See which ones you intersect with. Discard the ones you don’t. And repeat until you’re at leaf nodes and you have data you can check. These are used a lot for collision detection in 2D - which is effectively what we’re doing here.

Future Light Cone

- **Problem:** How can we share QuadTree state across processes?
- **Answer:** It'll have to be `ets` based, NOT a persistent data structure in process state

  Some yaks to shave to make free of concurrency bugs...

Future Light Cone

- Didn’t end up pursuing this too far
- There are way more events than entities, so LOTS of processes spawned
- Quad tree with dynamic updates is trickier than a static one
- Backing it with ets is another layer of complexity

So because of that, didn’t go too far

Processes are cheap in Elixir, but not FREE. If you have millions, it mounts up.

Past Light Cone

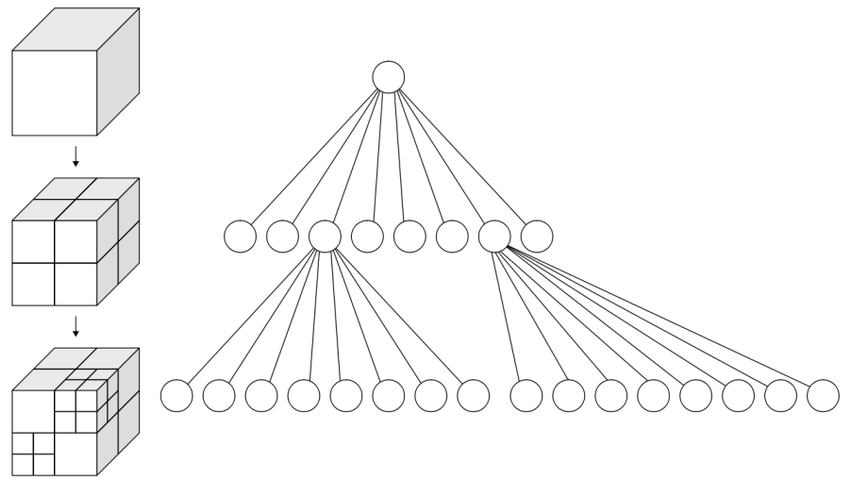
- `PastPropagationSystem`
- Tracks time/position of past events (🚀 Joe `{:message, “Hello Mike”}`)
- Run in a single process
- Each update, iterate through all observing entities
- Find all the events that meet their past light cone
- And publish a `PutComponentEvent` on them containing the propagated original event

The other alternative is to look at events in the past light cone

Past Light Cone

- **Problem:** How can we efficiently track and find all the events in 2D space over a period of time that an observer could see?
- **Answer:** Use an Octree

Octree



Octree is the next dimension up from a quadtree. We split a cube bounding box along 3 dimensions instead of two in 3D. Why 3D? Remember we've got a 2 dimensions for space + 1 dimension for time.

With the future light cone, we were using… actual time… as time, so it was a 2D quadtree. For the past though, we need to encode that as a dimension

Octree

- Can be efficiently implemented as a persistent data structure
- Past events are immutable - so most of tree is static, and easier to implement
- Events are only added
- Suitable to live in one process
- Fast enough. 100,000 observing entities can query/s with 10,000,000 events stored

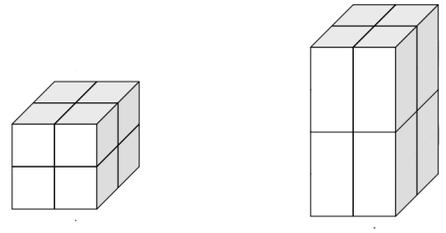
tree structures in general are a good fit for persistent data structures with immutable data like we have in Elixir.
Quadtrees are too, it's just our use case was a less good fit.
For the past light cone though, we've got a set of requirements that fit them better
We can do this in one process, so don't need to share or replicate state.
And they're plenty fast for what we need. The one I built has had some optimisation, there's more to squeeze out, but it's still more than good enough.

Although, yeah, if you're a C or Rust, or Assembly programmer you're probably snickering at how bad that is. But my mum said it was cool.

Octree

- **Problem:** How do we handle bounds growing? All new events by nature are outside of the time (z) boundary?

- **Answer:** If we get an event outside of the bounds, just double the bounds on that axis to fit it, then rebuild the octree (naive, but it works ok)

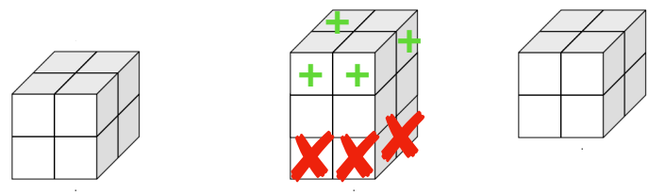


So usual Octree implementations set a fixed bounding box, and expect all elements to fall within that.
By the nature of our data, EVERY event is at a later time, so we're going outside the bounds with EVERY insertion.

So we do the simple thing, and just build a brand new octree when we go outside, pull everything out, and reinsert it. But we make the axis on the bound that was exceeded double the size required to fit that element. So after a while this is a very rare occurrence. Rebuilding the octree is slow, but not that slow, and amortised over all the insertions it works out pretty cheap

Octree

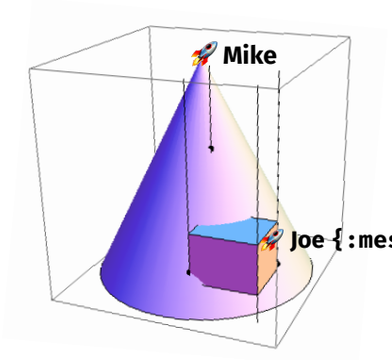
- **Problem:** How do we deal with data growing?
- **Answer:** Can just throw away the bottom 4 octants (and their data) periodically, and add 4 empty ones on top to replace them



With an append only data structure, we’re gonna get bigger and bigger over time. Especially if this is a long running process with games that run for days or weeks or months or years.
So the solution for this is pretty simple too. Just throw away the bottom of the octree, and add a new top layer. Because the z axis is time, that means we can cull old data this way.

Octree

- **Problem:** How do we query events in the past?
- **Answer:** Find the intersection of a cone with the octree. The math is boring, but simple and efficient.




```
{:message, “Hello Mike!”}
```

Octrex library

- https://github.com/madlep/octrex
- Persistent octree data structure in pure Elixir
- Super rough, but querying is fast enough (Inserting needs optimisation though)

```
                     ##### With input size 1000 elements #####
Name                       ips        average  deviation         median         99th %
Octrex.find_matching    263.29 K        3.80 µs   ±493.78%           3 µs          11 µs

                     ##### With input size 10000 elements #####
Name                       ips        average  deviation         median         99th %
Octrex.find_matching    221.65 K        4.51 µs   ±449.66%           4 µs          17 µs

                     ##### With input size 100000 elements #####
Name                       ips        average  deviation         median         99th %
Octrex.find_matching    104.23 K        9.59 µs   ±180.58%           9 µs          25 µs

                     ##### With input size 1000000 elements #####
Name                       ips        average  deviation         median         99th %
Octrex.find_matching    155.76 K        6.42 µs  ±2234.97%           6 µs          19 µs

                    ##### With input size 10000000 elements #####
Name                       ips        average  deviation         median         99th %
Octrex.find_matching    108.18 K        9.24 µs  ±7101.63%           8 µs          24 µs
```

I pushed some code that does this. It’s pretty rough, but it works. Not on hex yet. Just github.

But it’s fast. So finding elements that intersect with some query takes 9 microseconds when we have 10,000,000 events in our octree.

Propagation : The game!

Development status: HalfLife 3 ƛ

Coming soon! Stay tuned

So. The Game!

Development Status: HalfLife 3.

Yeah, still vapourware sorry. I was hoping to get a demo you could play today, but life happens. I’ve got all the big pieces working, I just haven’t put them together yet.

But coming soon!

Thank You!

Real Time Strategy

At Light Speed

Julian Doherty
@madlep
juliandoherty.com

And to find out when you can play it! follow me on twitter, or my site.

Thank you!

Components can live where ever

```
@components = {
  position: {},
  velocity: {},
  acceleration: {},
  sad_feeling: {}
}

def put_component(type, entity_id, component)
  @components[type][entity_id] = component
end

asteroid_id = 123
put_component(:position, asteroid_id, Vector2.new(x: 1, y: 2))
put_component(:velocity, asteroid_id, Vector2.new(x: 11, y: 22))
put_component(:acceleration, asteroid_id, Vector2.new(x: 101, y: 202))
```

So let’s look at how that would work in practice.

Apologies here, as I’m showing this in Ruby to set up a straw man to beat up. Same thing can be done in Elixir, but it’s a little clearer explaining it this way.

So we have a map of all our component stores. One for each type. To create a game object, we just put a bunch of components in different stores for that entity id.

Systems only care about what they care about

```
def do_movement_system()
  (@components[:position].keys() &
    @components[:velocity].keys() &
    @components[:acceleration].keys()
  ).each do |entity_id|
    acc = @components[:acceleration][entity_id]
    vel = @components[:velocity][entity_id]
    pos = @components[:position][entity_id]

    new_vel = vel + acc
    @components[:velocity][entity_id] = new_vel

    new_pos = pos + new_vel
    @components[:position][entity_id] = new_pos
  end
end
```

Here we just find entity ids that have all the components we care about - position, velocity, acceleration. And then do simple newtonian physics to move them.

Systems only care about what they care about

```
put_component(:sad_feeling, asteroid_id, "feeling sad")

def do_sadness_system()
  (@components[:sad_feeling].keys() &
    @components[:velocity].keys() &
    @components[:acceleration].keys()
  ).each do |entity_id|
    if @components[:sad_feeling][entity_id] == "feeling sad"
      @components[:acceleration].delete(entity_id)
      @components[:velocity].delete(entity_id)
      @components[:sad_feeling][entity_id] = "feeling a bit better”
    end
  end
end
```

Then we can have a separate system handling our sad asteroid. If it’s sad, it just stops.

So systems can read and write components.

Systems can be parallelized

```
Thread.new do
  loop do
    do_movement_system()
    sleep(0.1)
  end
end

Thread.new do
  loop do
    do_sadness_system()
    sleep(0.1)
  end
end
```

And having data split up, means they can be parallelized.

I can’t think of ANYTHING that could POSSIBLY go wrong with concurrent, shared mutable state...

```ruby
def do_movement_system()
  # ...
  @components[:velocity][entity_id] = new_vel
  # ...
end

def do_sadness_system()
  # ...
  @components[:velocity].delete(entity_id)
  # 😱
end
```

They can ge be parallelized… 😯

Now I can’t think of ANYTHING that could POSSIBLY go wrong with concurrent shared mutable state.

😬

Yeah, good luck debugging that when it comes up

Don’t feel too smug... yet

You can screw things up this much in Elixir too with things like ets or databases

You can still mess this up in Elixir too.

First cut of this I did exactly that by backing it with an ETS table. Which seemed like a good idea until I thought about it for a bit that there is no way to do concurrently transactions on an ets table and data gets splatted real quick