VR Animation and Locomotion Systems in Lone Echo

Jacob Copenhaver
Lead Gameplay Programmer
Ready At Dawn

Hello, my name is Jacob. I am a programmer at Ready at Dawn. I am going to be talking about animation and locomotion in Lone Echo.

Contributing programmers:

Filip Krynicki (Physics Constraints)

Dan Medeiros (Spine and Leg Animations)

Building Lone Echo is a team effort and credit for everything in this talk goes to the entire team. But, Filip and Dan are two people who made specific contributions to this talk, so I wanted to give them a special shout out.

What is Lone Echo? Lone Echo is a science fiction game set in a realistic, plausible future. The player inhabits the body of a robot and can see his robotic hands and arms. The player navigates space based environments using their hands to crawl over surfaces. We have an immersive story-driven single player mode and a team based multiplayer mode.

This talk is:

100% player tech

50% locomotion

50% animation

This talk is specifically about the technology behind our first person player. We are going to cover this in roughly 2 sections. First we are going to talk about the development and implementation of our movement model, then we are going to talk about how we animate the hands, arms, and body.

Gameplay Footage (PDF Version)

https://www.youtube.com/watch?v=A5tsdRb3PFw

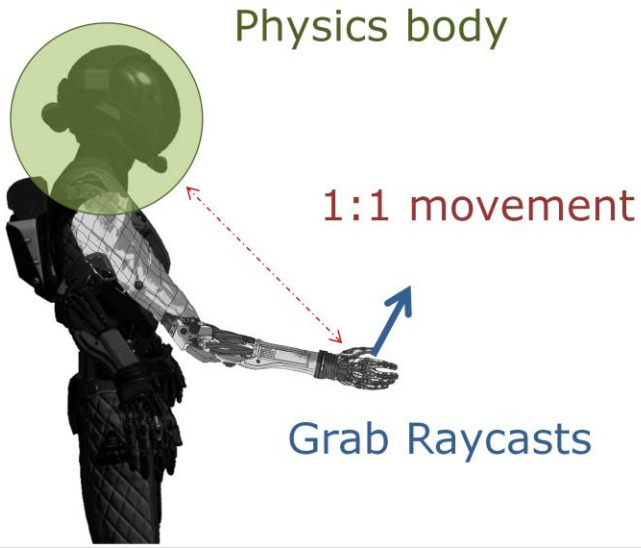
At the start of our project, our game director was watching YouTube videos of astronauts move aboard the International Space Station. The way they move is pretty fascinating. They drift very gentling and gracefully through the air. We also noticed that they move mainly using their hands, not their legs. This type of movement would become the basis of our player mechanics. Using your hands to push, pull, and climb in zero G.

The split screen footage shows an early prototype that we made just weeks after we received our first set of Oculus Touches. On the left is what the player is seeing, the green spheres are the hand locators. On the right is our game director playing the game. The key thing that you hopefully can see in this video is how our movement works: when you grab onto something, we start positioning the head 1:1 relative to the hand. This one mechanic allows the player to act out a wide variety of actions: crawling, climbing, pushing

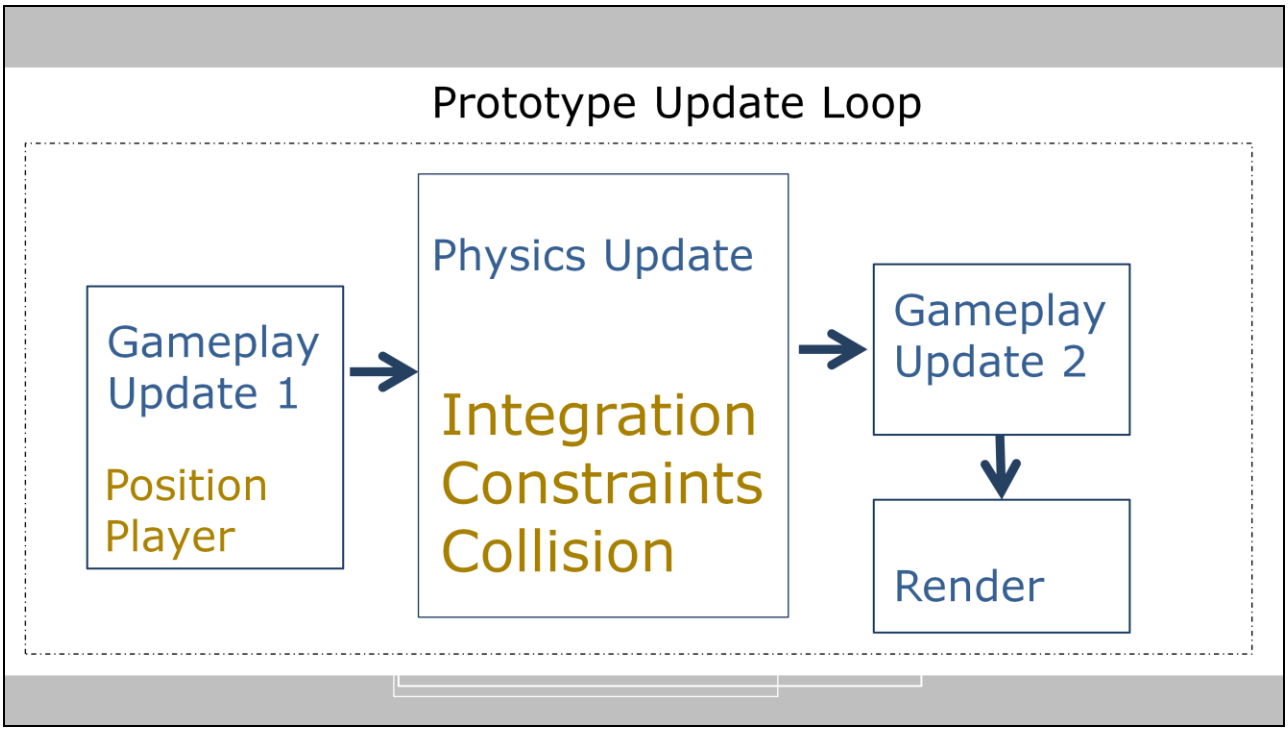
off, and stopping.

Finally, we have some footage of our current implementation. We have procedurally generated hand animations. Only one grip animation in the video is actually pre-authored, the rest are generated at runtime. You can grab any surface from any angle. We show the players arms and body, and have procedurally generated spine and leg animations. We have a much richer set of physics based interactions. You can interact with constrained objects like cabinets. You can climb on dynamic, moving objects like animated robotic characters.

Initial Prototype

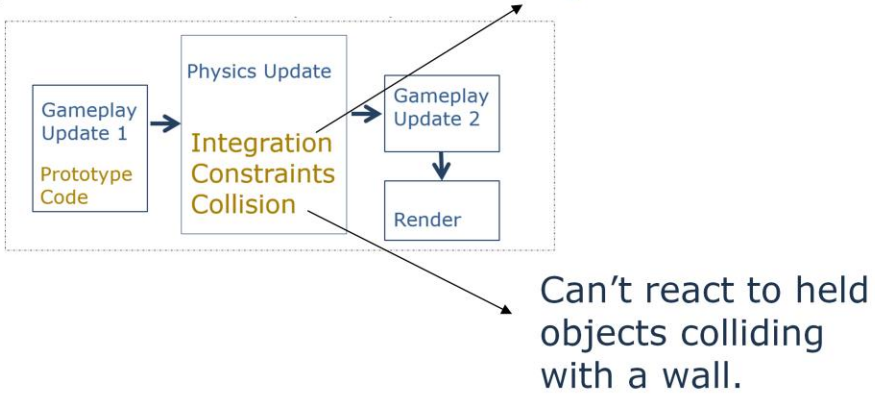


Our initial prototype was very simple, it was constructed using only 3 primitive systems. We added a physics body around the player's head, so they could float through the world. We cast rays from the palm to detect if the player is grabbing the environment. And finally, when grabbing the environment, we position the head 1:1 relative to the hand position. It worked great for climbing on static geometry, but couldn't really handle more dynamic environments like moving spaceships, levers, and physics interactions. The problem was how we were doing our 1:1 movement.

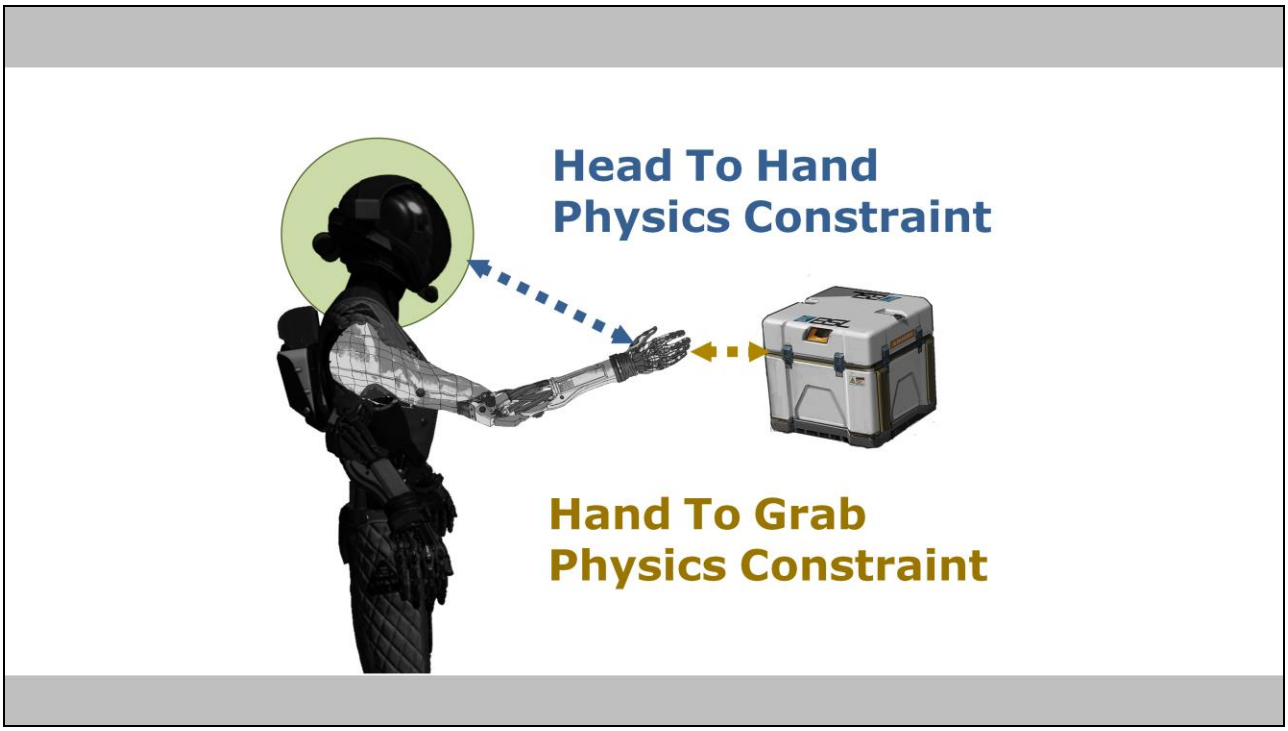


Our prototype positioned the player in the same way many games do. Gameplay code runs first, determines it's desired movement, then physics simulates. This is how many games work. This is how The Order 1886 worked.

Prototype Update Issues



But for this game it caused many limitations. Look at what is happening during the physics step. Integration moves objects, which means we were positioning the player before other objects moved. If the player could reach out and grab a moving spaceship, we were positioning him before that space ship moved. We were also positioning the player outside collision resolution, so if the player moved a held rock inside a wall we couldn't react by pushing back on the player.

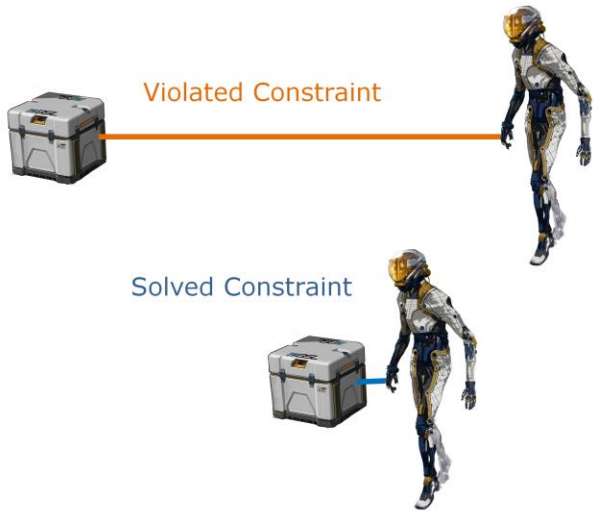


To fix the issues with positioning the player relative to moving objects and to allow player movement tighter integration with collision, we moved to modelling player movement using physics constraints. We added a constraint between the head and the hand, and a constraint between the hand a it's grab point.

If you are unfamiliar with constraints I will give a quick laymen's explanation. Without constraints, to the physics engine, the player and a held object are completely independent. You can think of adding a constraint as welding the two objects together: now the object the player is holding is welded to his hand. If the player is holding onto a spaceship and that spaceship moves: now the physics engine also knows it needs to update the "weld" giving us the opportunity to move the player along with the spaceship. If the player is holding a rock and shoves it into a wall, the physics engine will update the "weld" after collision resolves, allowing us to push back on the player.

Another additional bonus is flexibility. Constraint systems are designed to solve arbitrary chains, which allows us to model varied interactions like levers and cabinets.

Mass ratios cause floaty player movement



Let's go back to the metaphor of a constraint being a weld between two objects. If those two objects get separated, how does the physics system move them back together? The answer is that both objects move inward to satisfy the constraint, and they move in proportion to their masses. This makes sense: if I throw a heavy ball with an attached chain into the air, the chain dangles wildly while the heavy ball follows it's trajectory. So, using mass ratios to solve constraints is necessary to conserve momentum and have physically realistic simulations.

But if we take the example of a player holding a rock: solving his constraints in this manner would cause the players head to move every time he waves the rock around, regardless of how big it is. One major key to having the 1:1 motion work well and be comfortable, is for the player to feel in full control. This type of movement on the player is neither predictable or controllable by the player.

One thing we could do is make the player have infinite mass.

That would solve the floaty movement, but then no other constraints could *ever* move the player.

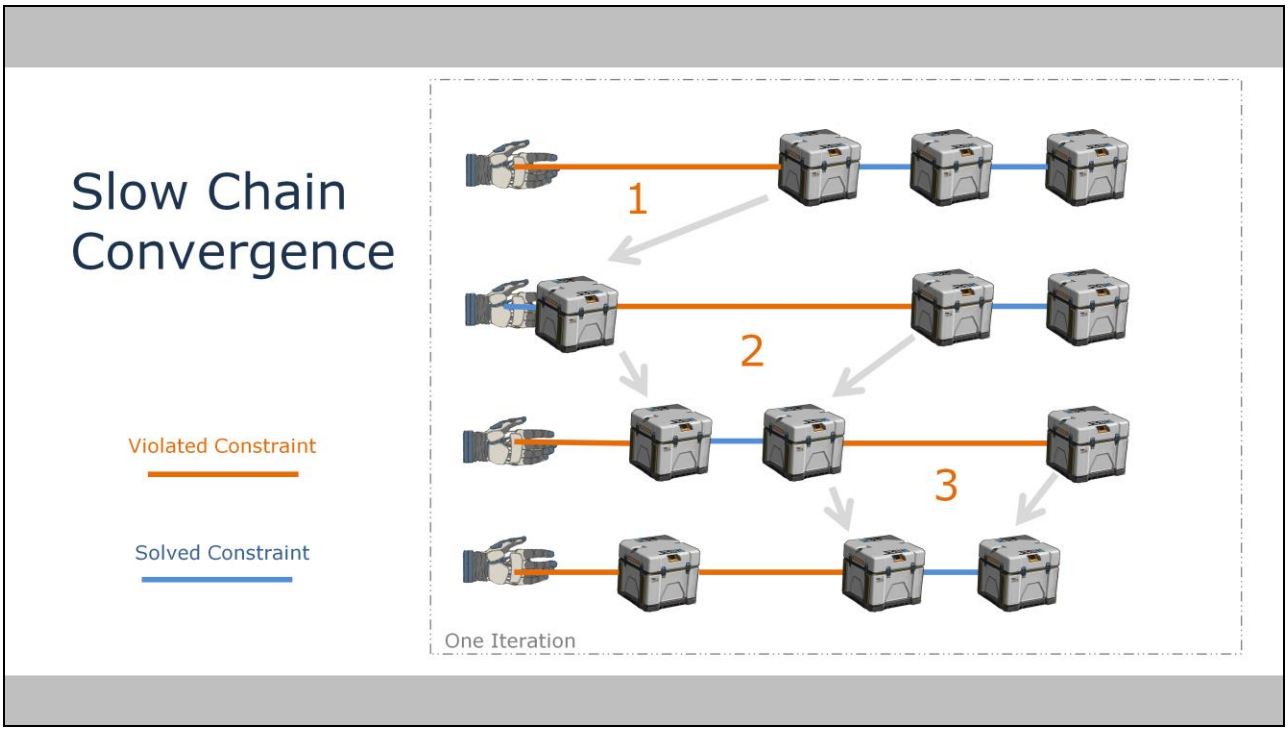
What we really wanted was ‘designer physics’. Don’t push the player around based on mass alone, but if collision with the level is violated, push back.

Ignoring mass ratios for player

- Forward Phase:
  - Solve constraints with player as infinite mass
- Backward Phase:
  - Solve constraints allowing player movement

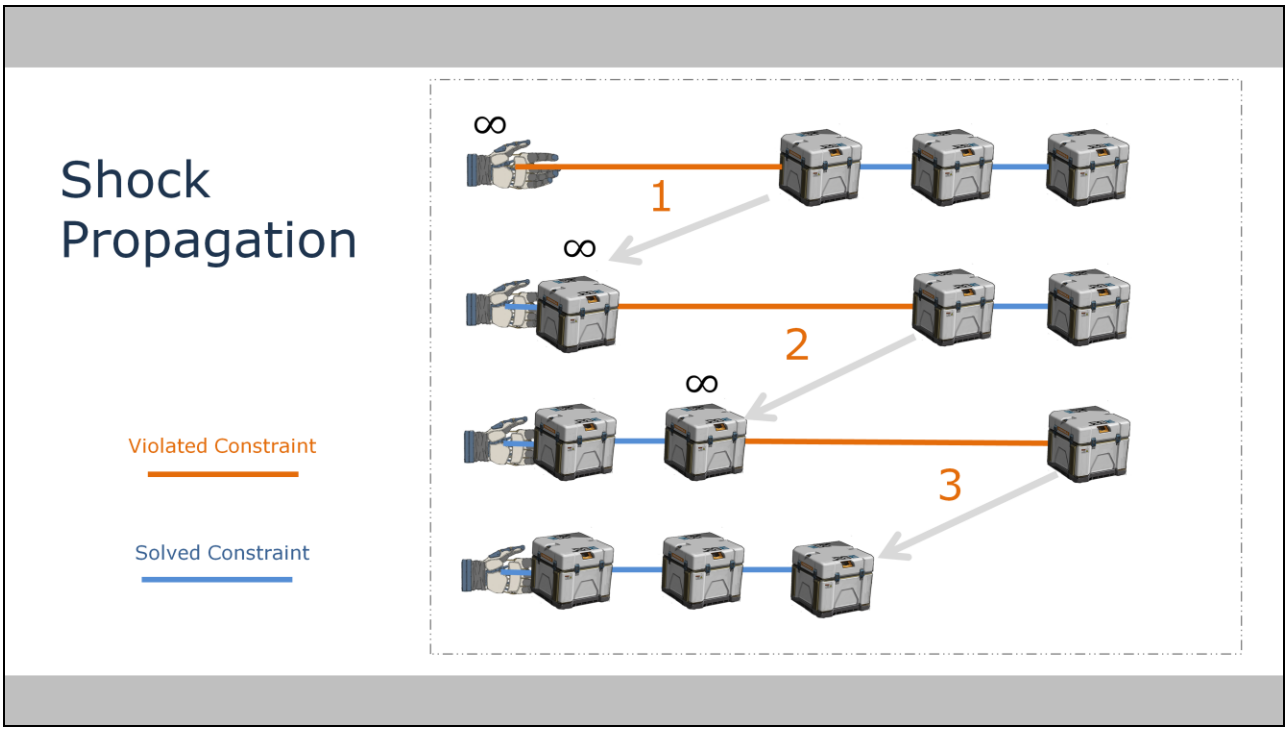
The way we achieve our 'designer' physics is by solving our constraints two times. In the first phase the player is essentially infinite mass and we try to force everything into place. You can think of this as us first trying to jam everything into a valid position from the perspective of the player. In the second phase we solve our constraints again, but allow the player to be pushed. This allows us to pick up player movement from constraints that are still violated, like level collision.

If I wave a rock around in the air, I get no player push back. But if I shove the rock into a wall, the forward phase will not be able to find a valid configuration. The backward phase will then pickup the remaining violations, pushing the player back.



At this point the player movement was working pretty well. But as time went on our designers started authoring more and more complicated setups, that had long chains of constraints. And these long chains of constraints were not fully converging. Convergence refers to how long a set of constraints take to become solved. If a set of constraints takes too long to solve, it won't happen in a single frame. If constraints don't converge in a single frame, you get incorrect behavior. Going back to the "welded together" metaphor: if constraints don't converge the welds start acting more like rubber bands, separating visually. A fixed lever can float off a wall. In our game, you get bizarre player movement.

This problem is not unique to our game, it is just a property of how iterative, local solvers work. Each constraint is solved independent of each other. Solving one constraint can cause a violation in another constraint. One link in the chain doesn't know about the next link down the chain. It just iterates, solving them over and over. The longer the chain, the more iterations it takes to converge.

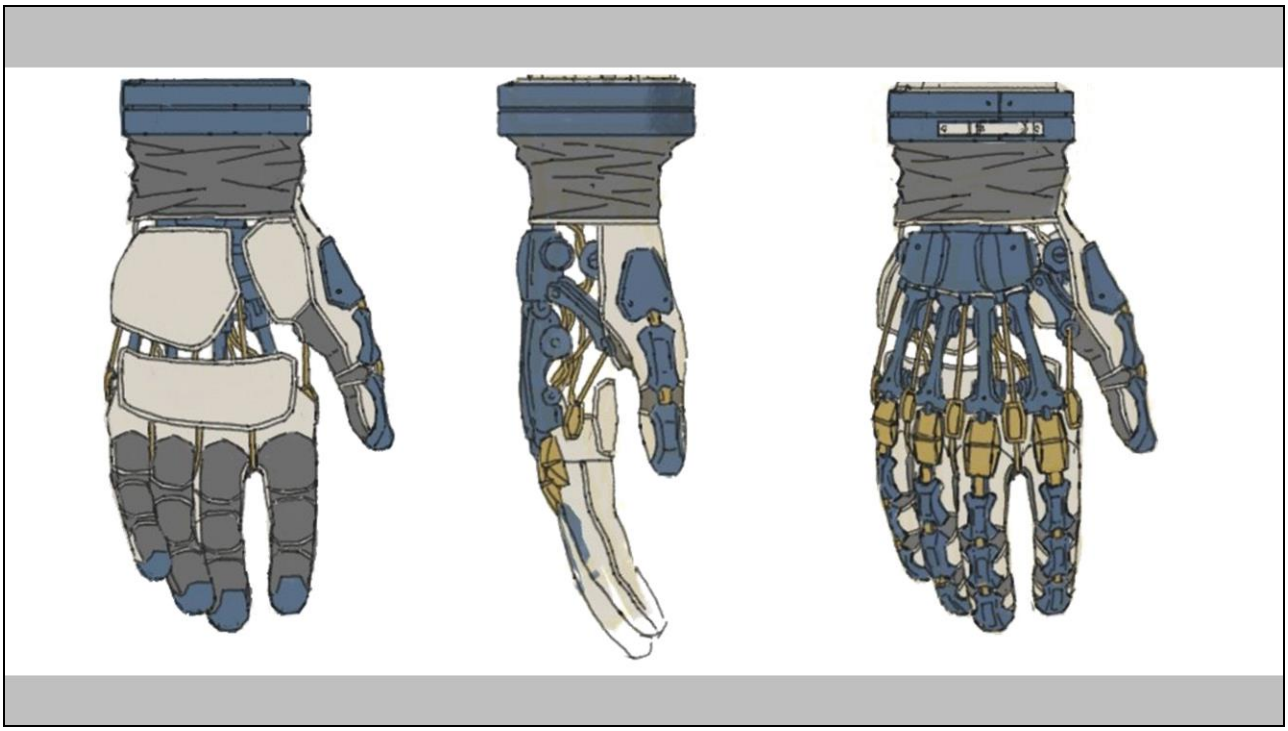


Our game depends on single frame convergence to avoid strange player movement, so we needed to support fast solving chains. We fixed our issues by using an existing technique called shock propagation. On the final iteration of our solve, we propagate infinite mass down the chain of objects. This allows the entire chain to solve in a single iteration.

For us this was a quick and easy way to ensure one frame convergence, but it definitely isn't a one size fits all solution. The main point that we are trying to illustrate is this: modeling the player movement as constraints unlocked power, but it also created a set of subtle technical issues that we needed to solve. It wasn't a free lunch.

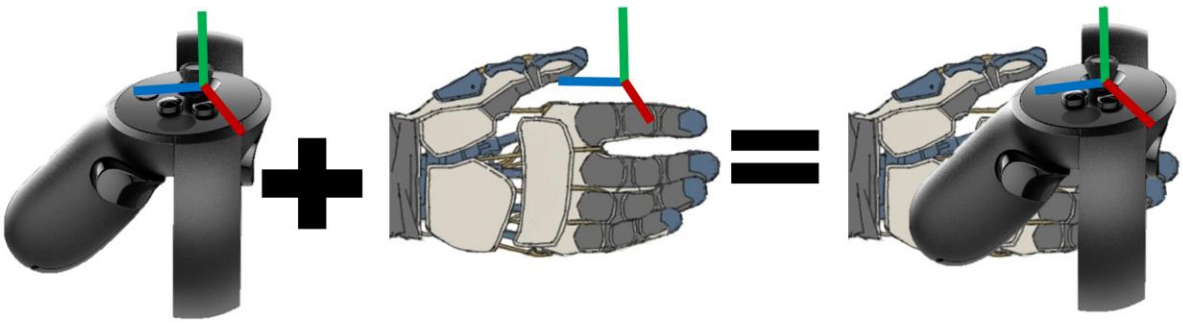
If you run into this issue, the easiest way to fix this problem is to simple design around it. Only use chains of constraints that converge in the amount of time you have. Another way to fix the problem is to simply increase the number of iterations used by your solver. It is also quite possible that your engine

already has a solution to this issue out of the box.



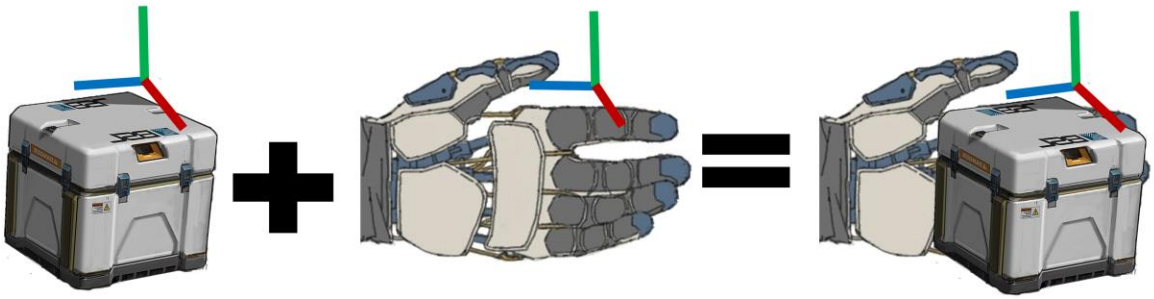
At this point we had our movement model working decently well. How did we go about adding hands?

Positioning the hand



Our first step in positioning the hand was to get an accurate model of the controller. Then we added support for displaying the controller both in game and in our animation package. Finally, we added a joint to the hand that defines how the controller and hand are positioned relative to each other. This allowed our animators to directly visualize exactly how the hand and controller would line up in game.

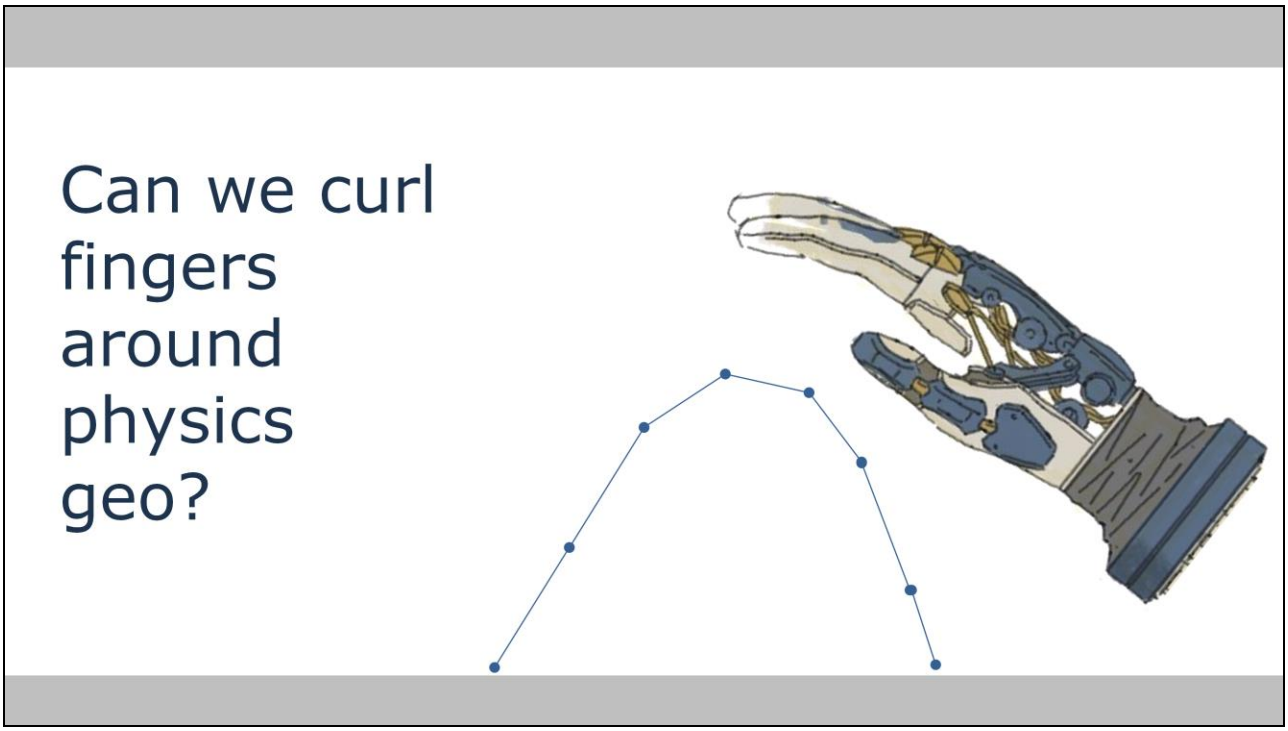
Pre-authored grip animations



We used a very similar technique for creating pre-authored grip animations. In the game footage at the start of this presentation, we use a pre-authored grip to grab a gun handle. This is a great use case of how we use pre-authored grip animations: for items that have very specific usage and require very specific finger articulations.

However, our game has the requirement that the player can grab any and every surface in the game world. The player's robotic hands are capable of Spiderman style grips: they can scale a sheer wall by simple placing their fingers on the surface.

However, pre-authoring animations for all surfaces in the game did not seem feasible.



So if we can't author grip information everywhere, inspecting the physics geometry seems like an obvious alternative.

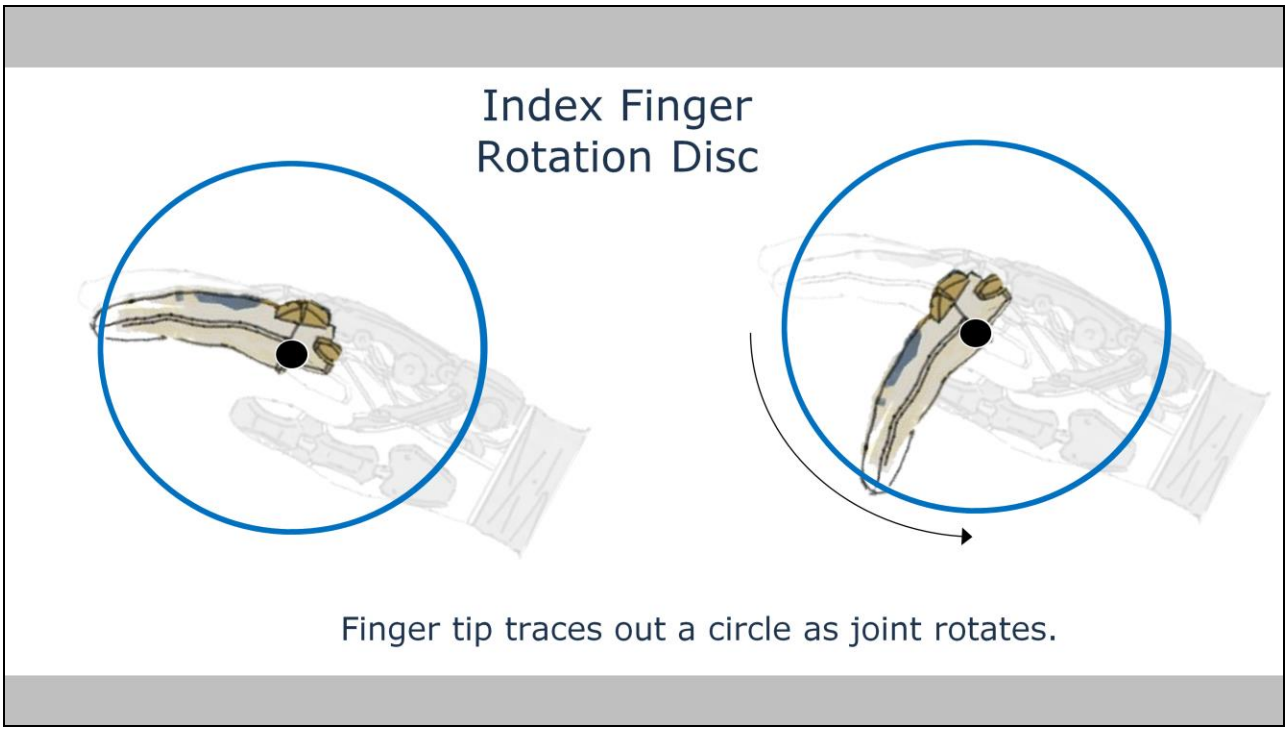
One possible option would be to use something like an active ragdoll. Each finger segment has a physics body and we try to drive their positions using motors. We were worried about two issues with this approach. One is performance: it would take about 30 physics bodies to represent our hand joints and all these bodies would be in high collision contention. The other thing is graceful failure. At times our hands are allowed to go through physics geo. At times our hands are allowed to show artifacts. We don't want our hand collision getting stuck on surfaces and we don't want our hand collision exploding if it is forced into invalid configurations.

Traversing physics as a graph

- Can search similar to navigation mesh
- Search for 'finger path' on geometry below the finger
  - Then modify joints directly to make contact

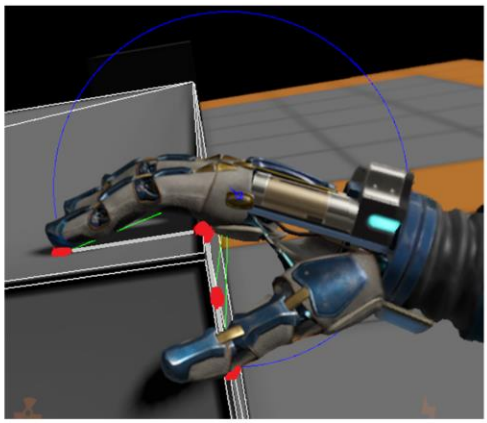
We decided to look for a method that would logically behave similar to "ragdoll" hands, but be much faster and fail more gracefully under invalid configurations.

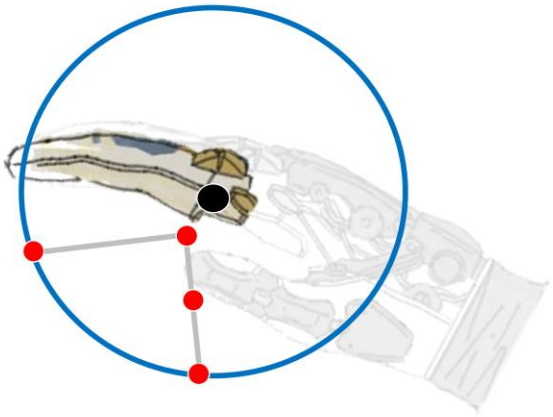
After looking at our physics engine, we noticed that for each triangle it was already packing in links to the neighboring triangles. Which meant that we could search across the geometry like a graph. So our idea was simple: could we trace out the path of triangles below each finger, then lower the finger to make contact?



Let's first look at doing just the index finger. If we assume the index finger only curls in a single direction, like a hinge, the tip of the index finger will trace out the path of a circle as it rotates. The normal of the circle is the axis of rotation. We do all our calculations in the 2D space of this circle.

Rotation Disc Triangle Intersections

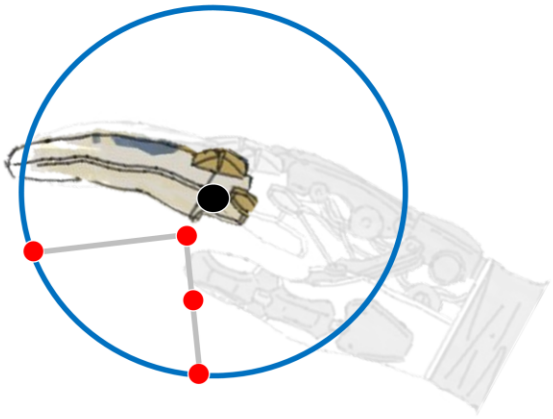




We need to find the intersections between the disc and the physics geo. Plane–triangle intersections have only two cases: the triangle doesn't intersect at all or two edges of the triangle intersect. So, what we are trying to find is a set of points: where the edges of the triangles intersect the disc.

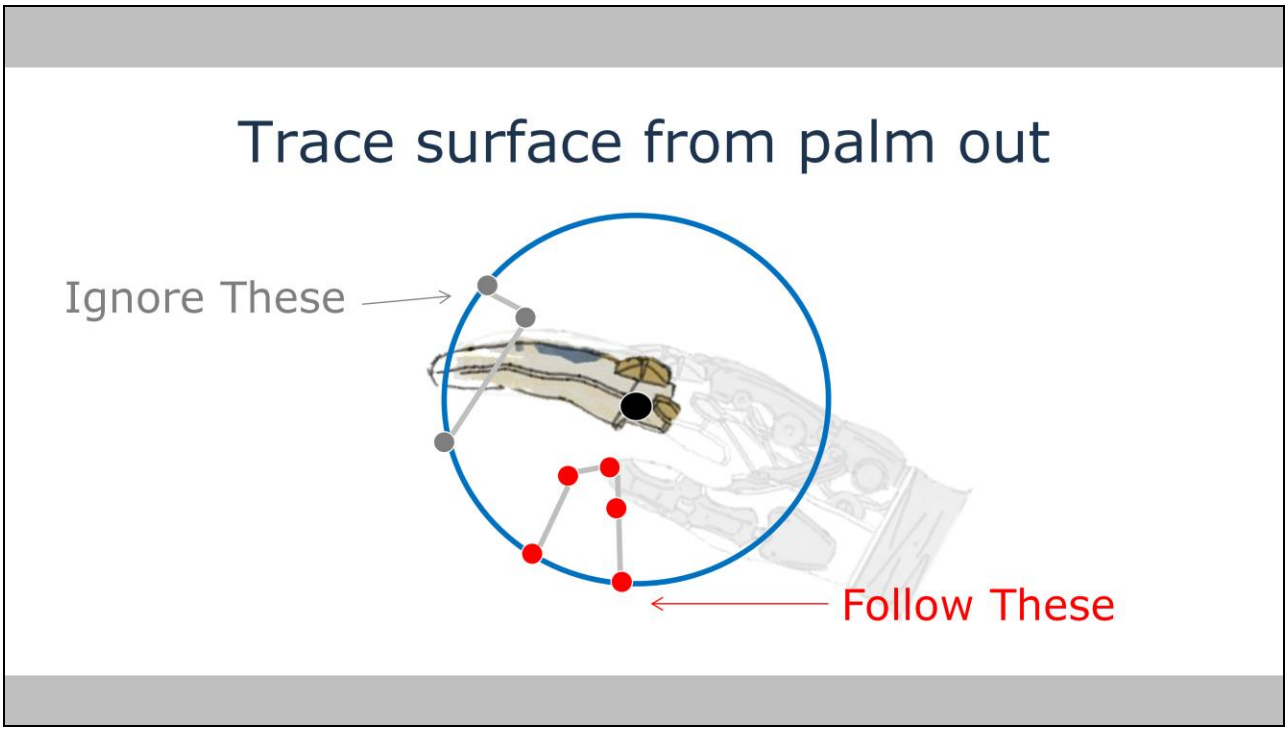
Finding Triangles on Disc

- 1: A* search to find triangle intersection nearest the palm
- 2: Then walk intersecting edges to find all triangles



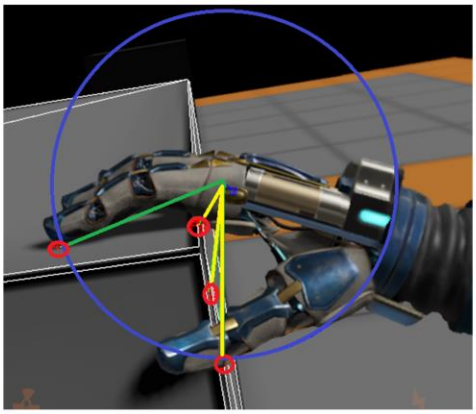
We find the triangle intersections in two phases. We do an A* search to find the first triangle intersection, using a heuristic that tries to find the intersection that is nearest to the palm. Then we walk the colliding edges to the left and right until we exit the disc.

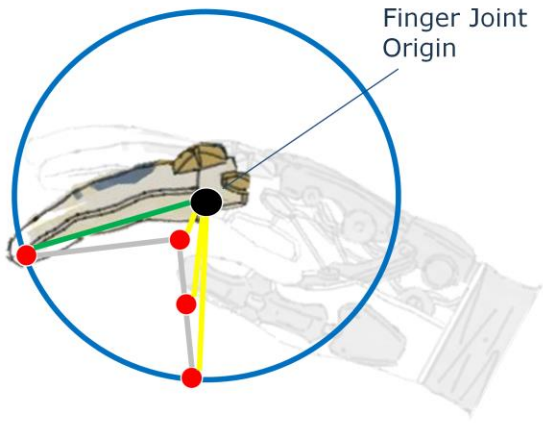
This may seem like an overly complicated way to find the intersections, but we have two reasons for doing so. First is performance. We initially tried a simple breadth first search, but at times we were visiting up to 200 triangles for a single finger. Our final approach will only visit 5 to 6 triangles in these same cases.



The second reason is that we don't actually want to collect all triangle intersections on the rotation disc. What we actually want to do is trace the surface from the palm out.

Rotate finger to highest angle





We then calculate the angles to make contact with each intersection point. Finally, we select the highest angle and rotate the finger.

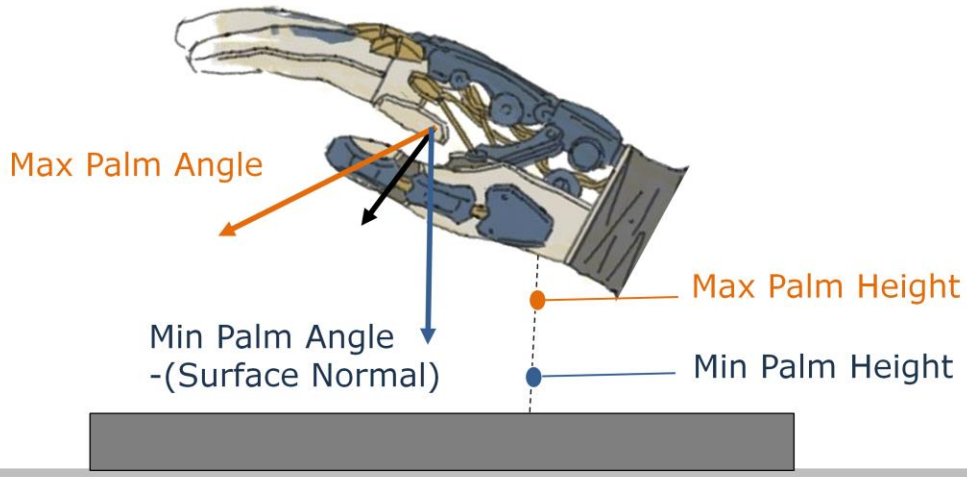
The other fingers can be done in a similar manner, with the exception of the thumb. The thumb is too complicated to model as a simple hinge joint, but handling it is outside the scope of this presentation.

Invalid Palm Positions

If we only curl the fingers and don't bother syncing the palm position, the algorithm can still work and feel good. In VR it feels great to have your hand do zero syncing when you grab objects.

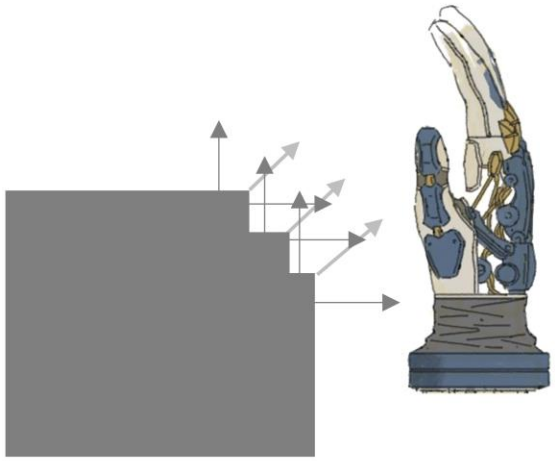
The problem is that the artifact rate is very high because it is easy to get your palm into invalid positions.

Palm-Surface Constraints



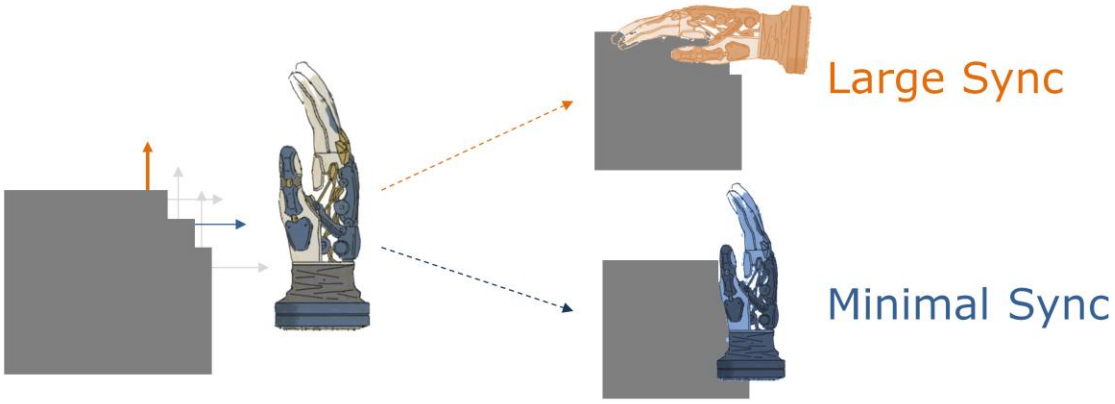
To fix invalid palm positions we implemented a set of palm-surface constraints. We enforce a minimum palm height, a maximum palm height, and a maximum palm angle. These constraints are evaluated every frame, so the player can twist and turn his fingers across a surface at runtime.

Picking a surface to sync to



Adding palm constraints created a huge reduction in the number of finger artifacts we were seeing. But it created a new type of artifact: if we picked the wrong surface we would get a massive sync. Our grab point is just a ray cast, so using it directly is essentially using random surface selection.

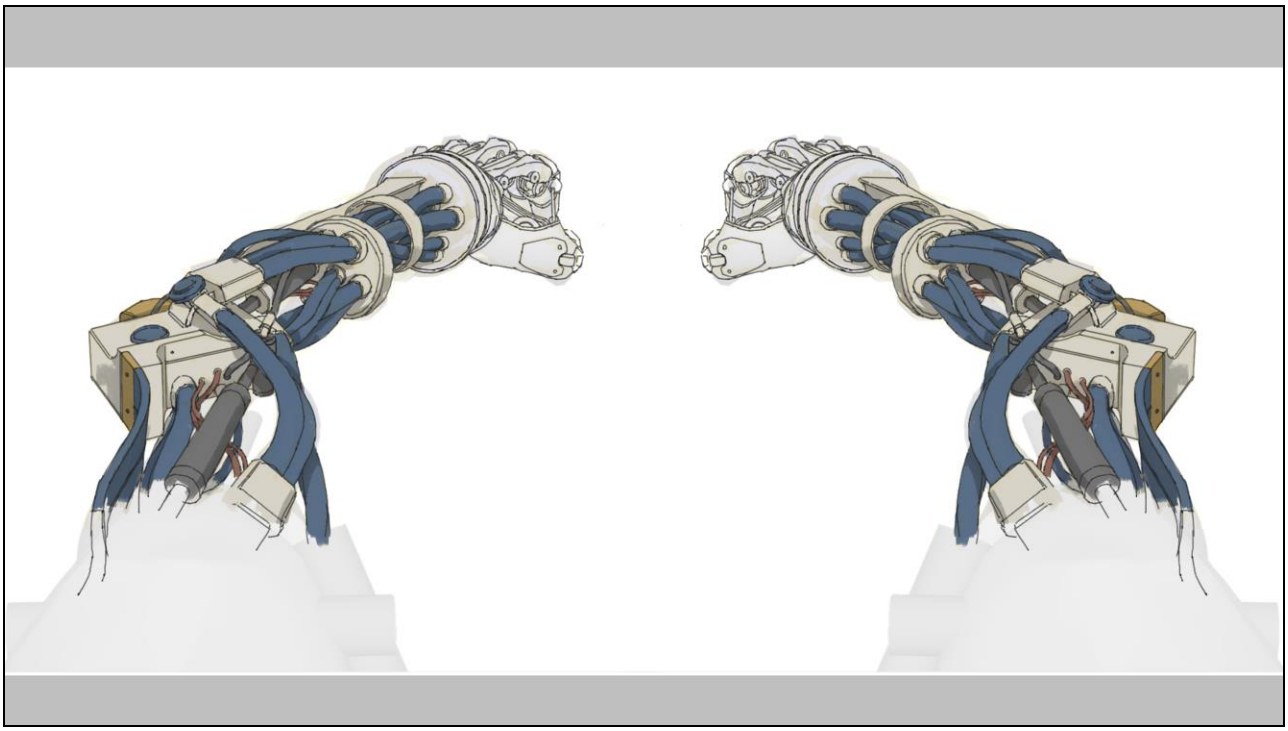
Breadth First Search For Minimal Sync



To fix this we run an additional breadth first search on the triangle geometry, centered around the initial grab ray cast. We select the surface that will cause the minimal amount of palm syncing when we apply our palm-surface constraints.

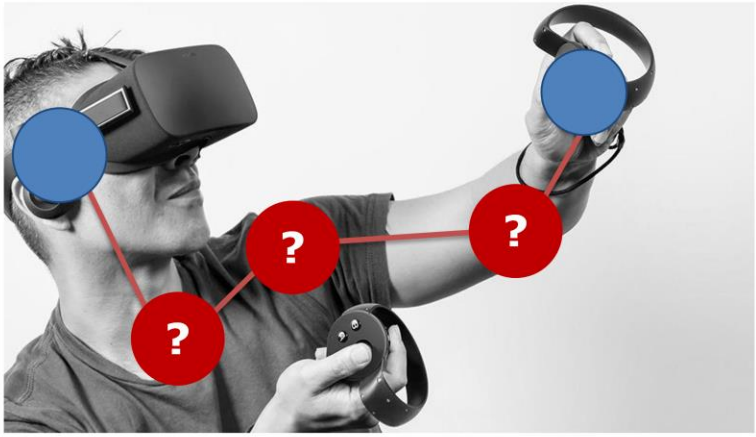
This is interesting because it is not the way you would search for a grab position in a normal AAA game. In a normal AAA game, you would search for a grab point that minimizes finger artifacts and allows for a realistic amount of grip strength. But in VR, because it feels so horrible to de-sync the hand, it is better to leave the hand as close as possible to where it is positioned. Moving the hand 10 inches to the left to give a better hand animation actually feels worse.

Going for minimal sync also capitalizes on the intelligence of the human user. The human user has positioned their hand and decided to grab: by respecting their positioning we capitalize on the fact that they normally place their hands in a physically viable position.



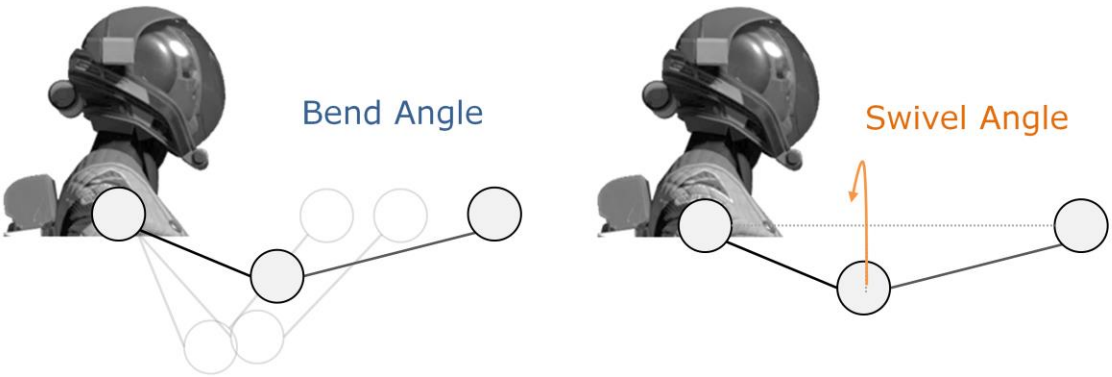
Now we our movement model and hands. How do we add arms?

Arm IK is underdetermined



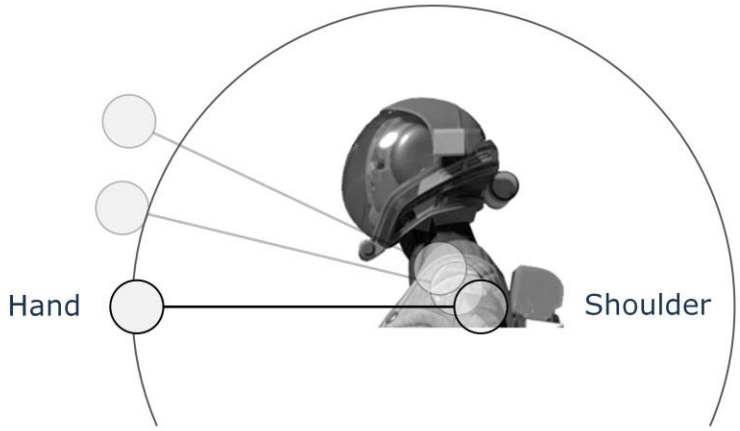
We only know the headset and controller positions. This doesn't give us enough information to truly solve for the chest, shoulder, or elbow. We need to come up with a set of heuristics to estimate these unknown positions.

Estimating Elbow Position



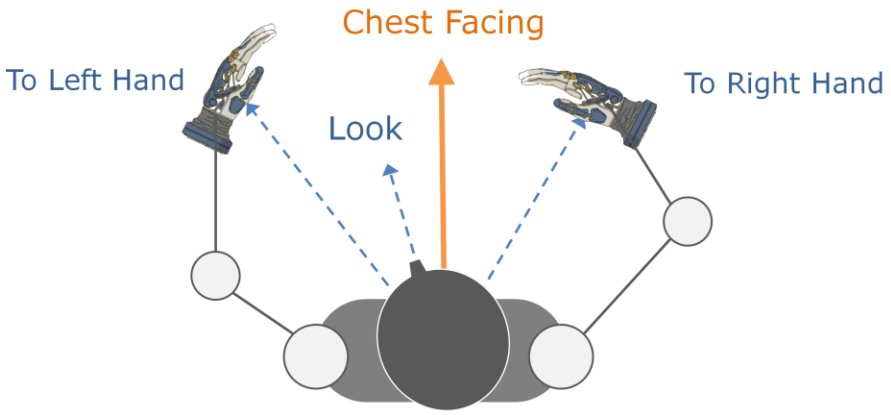
For the elbow we actually only need to estimate a single angle: the swivel. If we assume we already have an estimate for the shoulder position, we already know the direction from the shoulder to the hand. The only other parameter we need is the bend angle. Upon examination, we can see that we cannot change the bend angle without changing the distance from the shoulder to the hand.

Estimating Shoulder Position

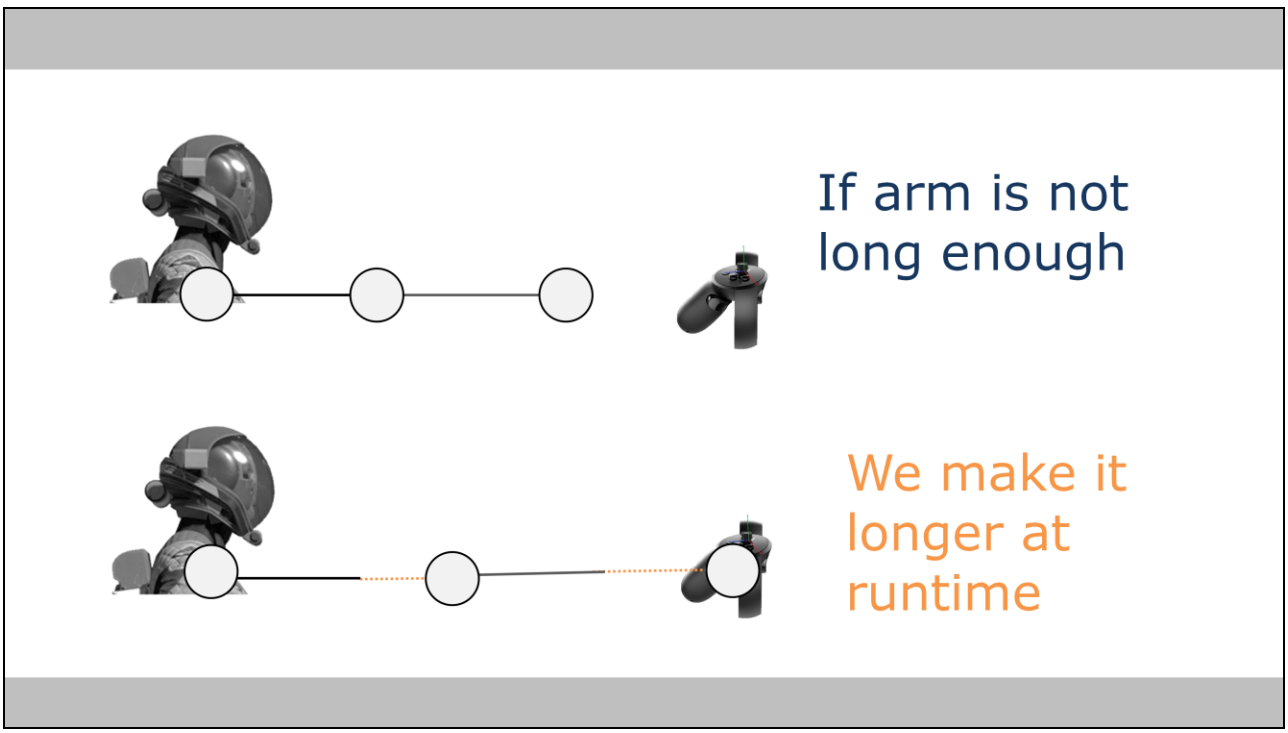


Estimating the shoulder position involves estimating clavicle extension and clavicle direction. We do this by projecting the position of the hand upon the plane shown in the diagram. Clavicle direction is assumed to always point in the direction of the hand. Clavicle extension is estimated as a simple curve that maps hand-to-shoulder distance to an extension magnitude.

Estimating Chest Facing



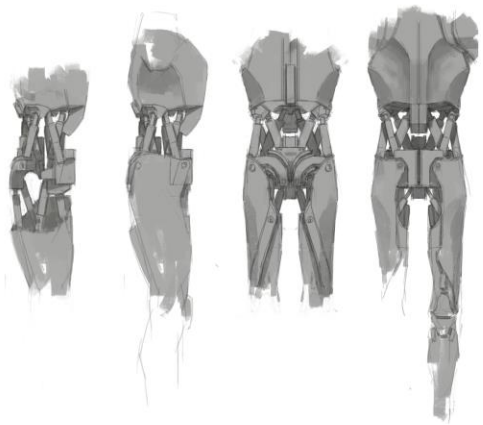
We estimate chest facing as a weighted blend of three directions: head look, head to left hand, and head to right hand. We also dynamically adjust the weights of the hand directions at runtime. As the hands come in closer to the chest, their weights go down. As the hands go behind the body, they are eventually ignored.



Our rig is setup to allow dynamic adjustment of the player arm length. We never let the player hand not match 100% 1:1 with the controller position.

1. Estimate chest facing
   - Rotate model to face chest direction
2. Estimate arm extension (stretch)
   - Adjust arm joints to desired length
3. Estimate clavicle extension
   - Input extension angle and distance into an additive animation
4. Estimate elbow pole vector
   - Blend between a base rest pose and wrist direction
5. Two bone IK solve for arms
   - http://mrl.nyu.edu/~perlin/gdc/ik/ik.java.html

This is an overview of the order in which we calculate out Arm IK estimates.



At this point we have covered our movement model, how we animate the hands, and how we animate the arms. The final piece to the puzzle is adding the body.

Spine/Legs Procedural Movement

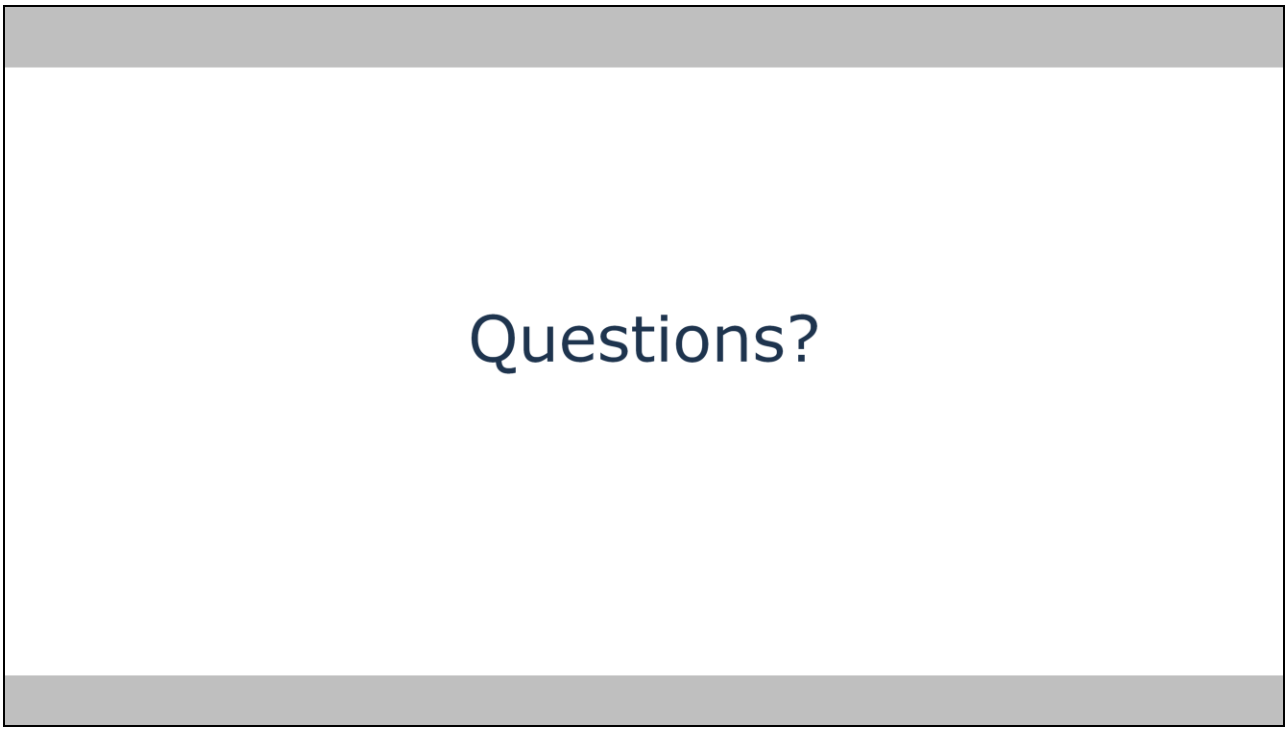
- Generated separately from arm IK chain
- Create angle constraints for each joint from chest downward
  - Then move based on player velocity
  - Motion propagates down chain link by link

Spine/Leg animations are also procedurally driven by the code. We solve this chain of bones completely separately from our arm IK chain. We create angle constraints for each joint from the neck downward. As the player's head moves through space we propagate that motion down this chain, link by link. The result is floaty, momentum based movement where the motion of the player snakes down to the rest of his body.

Spine/Legs Additive Animations

- Additives layer on top of procedural movement
  - Idle, Push off, Look Direction
- Allow animator controlled 'natural' movement

Finally, we layer a set of additives on top of our procedurally generated Spine/Leg animations to give it a layer of animator controlled movement.



Questions?

Contacts

Jacob Copenhaver: jake@readyatdawn.com

Filip Krynicki: filipk@readyatdawn.com

Dan Medeiros: dan@readyatdawn.com

Feel free to contact us with questions.