

Networking Tutorial 2: Networking Issues in Games

New Concepts

- ▶ Types of Network Game
- ▶ The Real-Time Condition
- ▶ The Consistency Condition

- ▶ Zoning
- ▶ Interest Management

- ▶ Dead Reckoning

Network Games: The Early Years

- ▶ Purpose of network games is to permit multiple users to interact in a single gaming environment from different terminals
- ▶ Early approaches weren't much more sophisticated than an IRC client

Network Games: Multi-User Dungeons

- ▶ MUDs popularised the concept (1970s+)
- ▶ Generally followed tabletop RPG tropes
- ▶ Games were often protracted, session-based affairs
- ▶ Some had player-adopted DM roles, others automated

Network Games: MUD to MMORPG

- ▶ Graphical MUDs (like EverQuest and UO) pushed the genre forwards
- ▶ Also tested the limitations of network technology
- ▶ Other multi-player experiences of the period had significantly fewer interacting players (e.g., Baldur's Gate multiplayer had 6 people and dubious reliability)

Classifying Network Games

- ▶ Network games are a sub-category of Network Virtual Environments (net-VEs)
- ▶ Two active research areas (with increasing amounts of cross-pollination)
- ▶ Distributed Interactive Simulations (DISs)
- ▶ Collaborative Virtual Environments (CVEs)

Distributed Interactive Simulations

- ▶ Where a lot of net-game engineering comes from
- ▶ Focus is on real-time simulation and interaction
- ▶ Research often (but decreasingly) associated with military application
- ▶ Which kinda makes sense when you're developing Call of Duty

Collaborative Virtual Environments

- ▶ Collaboration over simulation
- ▶ Communication more important than real-time interaction
- ▶ Simplest CVE is a virtual board-room call over Skype
- ▶ Blackboard is a form of CVE

What about OUR game?

- ▶ Which techniques you select from what source depend on the nature of the game.
- ▶ Real-time?
- ▶ Turn-based?
- ▶ Somewhere in the middle?
- ▶ Level of detail - is position REALLY position? Is interaction REALLY interaction?

A note on bandwidth

- ▶ The rate at which the network can deliver data to a destination client.
- ▶ Typical (new) Ethernet provides 1000 Mbps (1,000,000,000 bits per second).
- ▶ Most online games played over the internet, where sustained bandwidth varies a great deal and, in the UK, tends to cap at about 40Mbps
- ▶ The more complex a virtual world is, the more bandwidth is required

A note on bandwidth

- ▶ 20 clients each connected in a peer configuration broadcasting information about every entity they're responsible for, every frame.
- ▶ Each client is responsible for 100 animated entities (2000 in total virtual environment).
- ▶ Each entity requires position vector information to be distributed for a frame rate of 25 frames per second (assume this information may be carried in 3 eight bit word lengths - we ignore packet dependent data here).
- ▶ $24 * 2000 * 20 * 25 = 24,000,000$ bits need to be distributed per second (bandwidth 24Mbps).

A note on bandwidth

- ▶ That's only considering simple data
- ▶ What about complex data?
- ▶ Data that updates more than once per frame?
- ▶ And how long does it take us to process and react to that data when it's received?
- ▶ And latency?

Leads to our Constraints

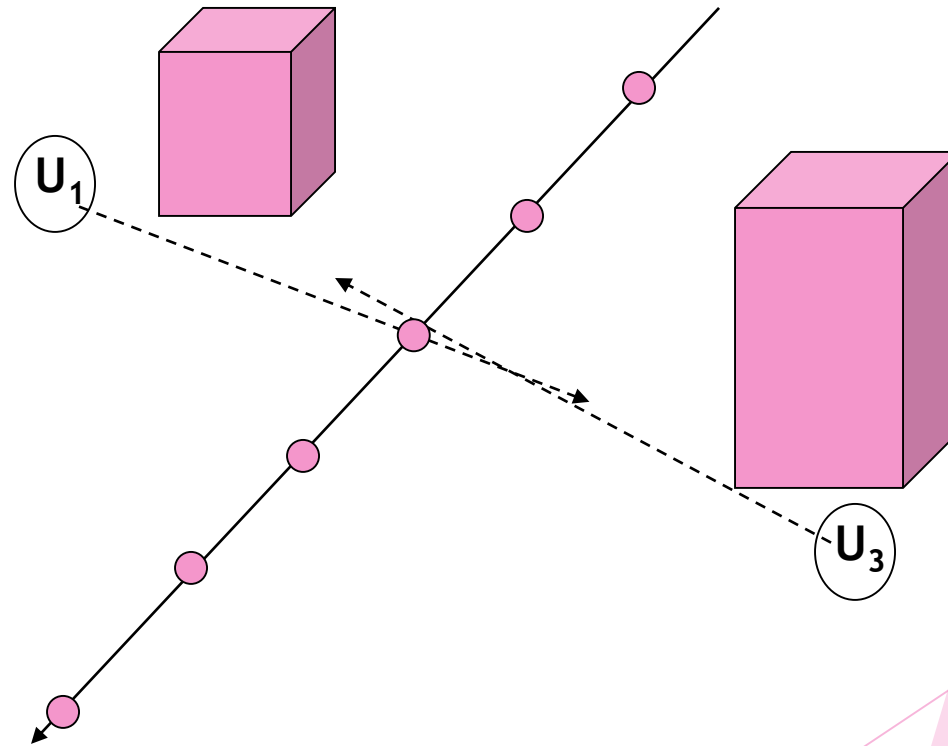
- ▶ Two constraints are key to how we go about structuring our net game
- ▶ Real-time Problem - our simulation should be updated in real-time. Can we actually do this? What impact is there if we can't?
- ▶ Consistency Problem - our clients should share a consistent world view. Can we actually do this? What impact is there if we can't?
- ▶ And where does bandwidth tie into it?

Real-Time Problem

- ▶ If player U_1 fires a gun by time t , then **all** players should see player U_1 fire a gun by time t

Example of the Real-Time Problem

- ▶ Three users (U_1 , U_2 , U_3) participate in an online FPS.
- ▶ U_2 appears in 6 different positions in consecutive frames (running in a straight line).
- At frame 3, U_2 enters the line of fire of U_1 and U_3 .
- Due to message latency, U_1 views U_2 at frame 3 sufficiently late for U_3 to gain an unfair advantage



Real-Time Problem

- ▶ First consideration:
 - ▶ Is this element one which requires real-time updating?
 - ▶ Gravity? Precise animation frame? Probably not
- ▶ Second consideration:
 - ▶ Can this update be managed client side without real-time communication with the server?
 - ▶ E.g., time-reactive skybox? Updates in real-time, but only needs to periodically sync with the server's view of time

Real-Time Problem

- ▶ If element DOES need real-time solution
 - ▶ Need to minimise impact of delayed packet receipt
 - ▶ Need to minimise packet size
 - ▶ Need to minimise time taken to process packet (balancing act with packet size)
- ▶ Approaches to resolving this:
 - ▶ Zoning and Interest Management
 - ▶ Predictive Modelling

Consistency Problem

- ▶ If player U_1 shoots player U_2 then **all** players should see player U_1 shoot player U_2 .

Consistency Problem

- ▶ Solutions to this tend to be divided between network engineering (i.e., socket type employed) and design considerations of the game
- ▶ Design approaches often involve completely extracting any real-time considerations from the problem which needs perfect ordering
- ▶ Example: Looting in an MMO. You never pick up a physical sword.

Addressing the Real-Time Problem

- ▶ Zoning
- ▶ Interest Management
- ▶ Dead Reckoning

High Level Zoning

- ▶ Continents in WoW
 - ▶ Planets in TOR
 - ▶ Map Zones in Warhammer Online
 - ▶ Basically, anywhere you can stick a loading screen
-
- ▶ If normal distribution of players, normalises server workload (and traffic), reducing per-server bandwidth and improving performance
 - ▶ Reduces traffic to each individual client, improving performance

High Level Zoning

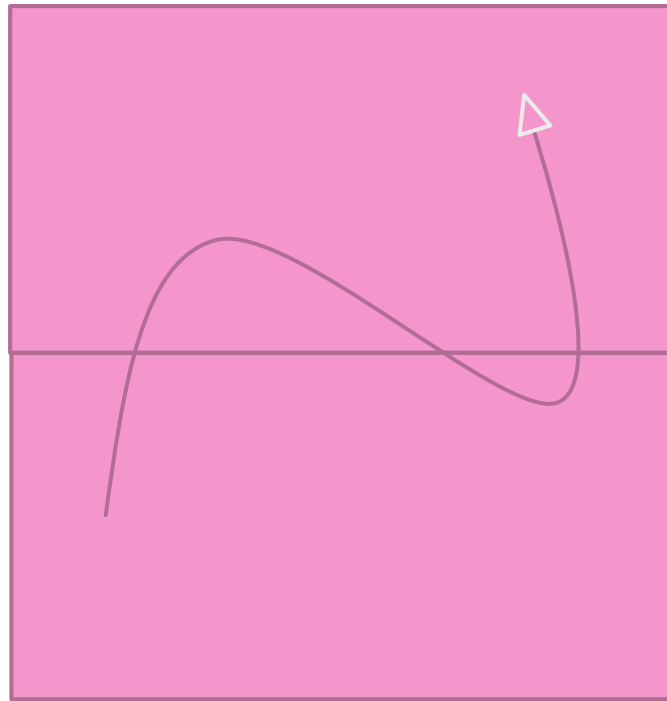
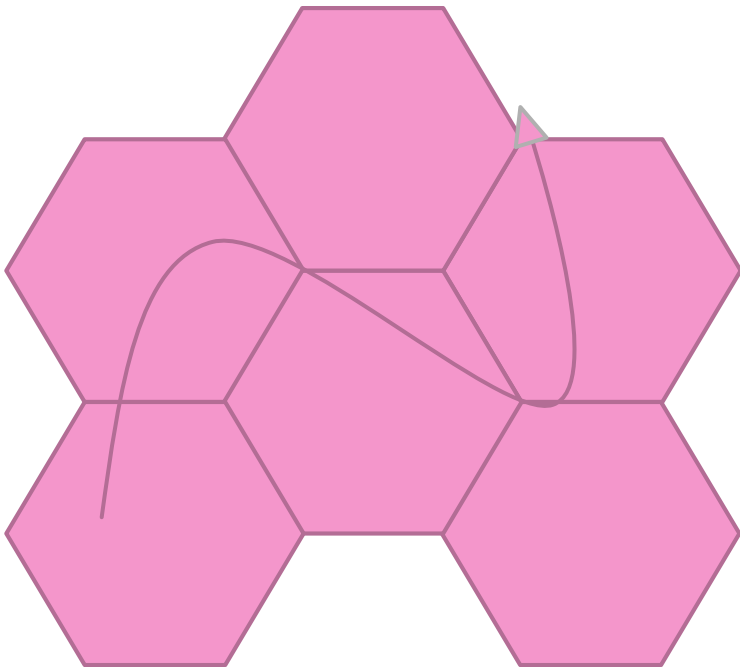
- ▶ Vulnerable to the same issues as world-space partitioning in collision detection
- ▶ If all of your players are in one zone, you have overhead of management with no benefit
- ▶ If you have large-attendance events in one zone, you artificially engineer your own worst-case scenario
- ▶ If your game has level-based progression, you *also* artificially engineer your own worst-case scenario *unless* you have some incentive for high-level players to return to low-level zones (e.g., GW2)

Spatial Zoning

- ▶ Map is logically divided into multiple zones
- ▶ Each zone encompasses players in the same vicinity
- ▶ A player moving from one zone to the next is disconnected from one server, and joins another
- ▶ Zones are geographically connected, normally using squares or hexagons as these interconnect neatly (and can easily be checked against position), while irregular shapes are more complex to interconnect (and more complex to compare against position)

Spatial Zoning

- Density and shape of zones relates to the rate at which a client will be moved between servers:

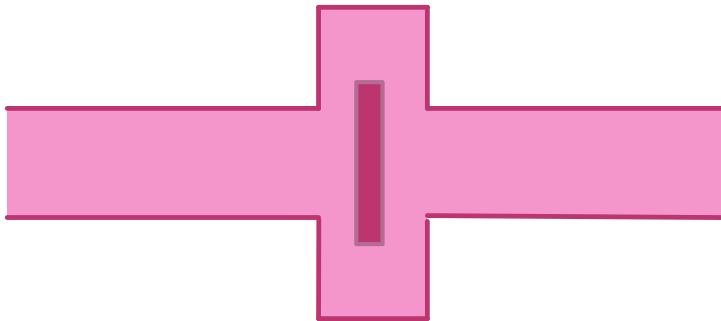


Spatial Zoning

- ▶ The goal is to ensure that a given client never 'feels' like he is changing servers - no loading screen, seamless performance
- ▶ Maintaining this can be difficult, and often requires some overlap in these spatial zones, or duplication of data between zones to avoid players 'popping' into existence as they cross a boundary
- ▶ As the servers managing these zones have dedicated, physical connections and are in the same geographical location, communications issues between servers are minimised
- ▶ An additional layer of complexity if employed for distributed-server architectures

Spatial Zoning

- ▶ Design elements can help
- ▶ Reduce visibility across zones with terrain features (e.g., mountain ranges blockading most of a region's edge, with a gap in the middle)
- ▶ Making the transitioning area a corridor, rather than an open space:



Megaservers - Zoning, SDRAWKCAB

- ▶ These approaches can be used to merge low-population virtual spaces, to make better use of computational resources.
- ▶ As servers can be spun up and spun down, nowadays, in realtime, this can be a significant cost saving while also having strong gameplay benefits (GW2 world events - no fun if your realm is empty)

(Area Of) Interest Management

- ▶ Two basic approaches:
 - ▶ Static Geographical Partitioning
 - ▶ Behavioural Modelling
- ▶ As in all things, Game Developers cherry-pick from both.

Static Geographical Partitioning

- ▶ Essentially maps to Zoning
- ▶ Most interesting interactions occur in specific regions in the game world (such as cities)
- ▶ Players are not expected to remain outside of these 'key regions' for long, by way of intervening territory being intentionally uninteresting
- ▶ Client Area of Interest, then, is the key region they occupy, or the largely unoccupied wilderness between key regions, with no finer grain approach being applied. Second Life employs this approach.

Behavioural Modelling

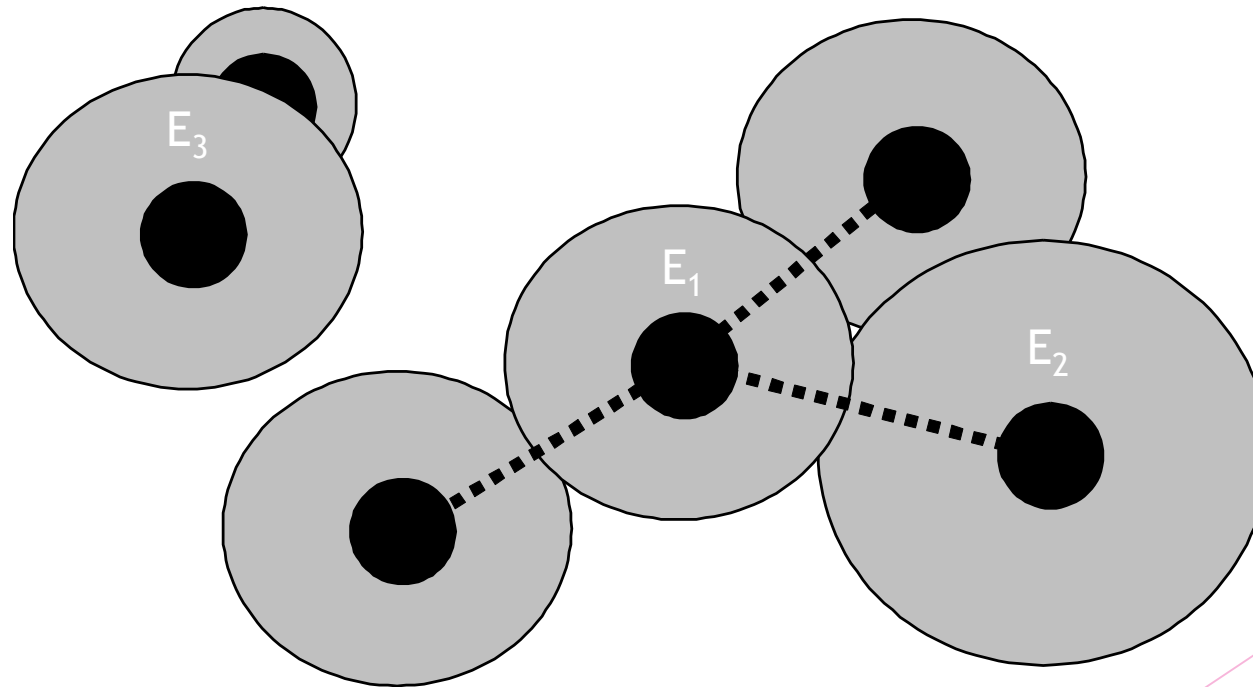
- ▶ Military simulation example: A plane and a jeep
- ▶ The two entities have different:
 - ▶ Speed ranges
 - ▶ Visual detection ranges
 - ▶ Area of influence ranges
- ▶ As such, the area of interest of the plane is modelled differently to the area of interest of the jeep.
- ▶ Not as deeply explored as geographical partitioning

Interest Management Models (Publisher-Subscriber)

- ▶ Publisher creates an event; subscriber consumes the event.
- ▶ A client is therefore a publisher of their own actions, and a subscriber to the actions of others who can influence them in some fashion
- ▶ When a client can no longer be influenced by the actions of another publisher, it no longer receives their events
- ▶ Permits a level of control of broadcasting - counter cheating

Interest Management Models (Aura-Nimbus Approach)

- ▶ Aura-Nimbus approach is straightforward to understand, and commonly employed.



Interest Management Models (Aura-Nimbus Approach)

- ▶ Aura
 - ▶ Client's range of influence
- ▶ Nimbus
 - ▶ Client's range of interest
- ▶ If a client's range of influence intersects with another's range of interest, it becomes a publisher to that client

Interest Management Models (Aura-Nimbus Approach)

- ▶ Implementation is analogous to simple sphere intersections in physics
- ▶ Drawback is lack of scalability - if everyone is in everyone else's regions of interest, all the communications overhead and additional sorting overhead of the algorithm

Other Perception-Based Approaches

- ▶ Purpose of interest management is reducing network traffic to client
- ▶ Simple examples are visibility and reachability checks
- ▶ Other contextual checks based on nature of the game - possibility of clients being geographically co-located but unable to interact or affect one another?
- ▶ Possible optimisation of which elements are sent based on this - e.g., if no friendly fire possible, only need to know if an ally dropped a grenade, not how much damage it did, etc.

So, Dead Reckoning

Dead Reckoning: Bandwidth

- ▶ Consider 4,000 Clients connected to a single centralised server (WoW model)
- ▶ Server is responsible for tens of thousands of entities, of which 400 (PCs and NPCs) on average need updating to any given client.
- ▶ Assume same position vector information to be distributed at 25 frames per second.
- ▶ 38.4Mbps upstream from the server; 2.4Mbps downstream to the server.

Dead Reckoning: Latency

- ▶ The amount of time required to transfer a bit of data from one point to another.
- ▶ LAN latency may be less than 10 milliseconds, transcontinental connections may be 60-100 milliseconds and intercontinental connections may be over 200 milliseconds (taking the Internet as an example).
- ▶ To ensure successful interaction the latency should not exceed 100 milliseconds (200 milliseconds delay overall).

Dead Reckoning: Latency

- ▶ Consider the consequences of latency on our update rate in the hypothetical system.
- ▶ Remember that a physics engine running at 120FPS will update every 8ms.
- ▶ As such, broadcasting every physics update is pointless in a network game - it's infeasible, and by the time an update has been received and confirmed, a dozen other updates might have occurred.
- ▶ If we use our physics engine's update rate, we can easily increase our bandwidth consumption four-fold.

Dead Reckoning: Can We Do Without It?

- ▶ So, we've established that we can't send every physics update.
- ▶ Even if we consider the MMO model where clients only update the server about a single entity, latency makes it impractical.
- ▶ Imagine an MMO where every character's position is only updated once every quarter of a second, pinging from place to place.
- ▶ Bad engineering, bad gameplay, bad game.

Dead Reckoning: The Problem

- ▶ The Key Issue: There can be no such thing as a 'god-like' view of every object's absolute position at any given time in a networked game, which is shared by the server and all clients.
- ▶ Can't happen unless you have a 0-latency, 0-packet loss network. Which can't happen, because light speed and network engineering.
- ▶ All any client has at any given time is its own perceived truth: as such, the best-case scenario is a believable guesstimate as to where any entity is.

Dead Reckoning: The Solution

- ▶ Comes from the term “Deduced Reckoning”, and is based on physics of our simulation.
- ▶ As with physics system, we’re making a best guess on where something ought to be, based on where it was, and where it was headed.
- ▶ Difference is that someone else can mess with this object without our knowledge, and we only find out after the fact!

Dead Reckoning: Prediction

- ▶ General solutions to dead reckoning use derivative polynomials.
- ▶ 0th Order polynomials
 - ▶ Pure position update - frequent state regeneration
 - ▶ Analogous to Projection Method in collision response
- ▶ 1st Order polynomials
 - ▶ Employs velocity
- ▶ 2nd Order polynomials
 - ▶ You can see where this is going...
- ▶ 2nd Order is most popular, but not always most appropriate...

Dead Reckoning: Prediction

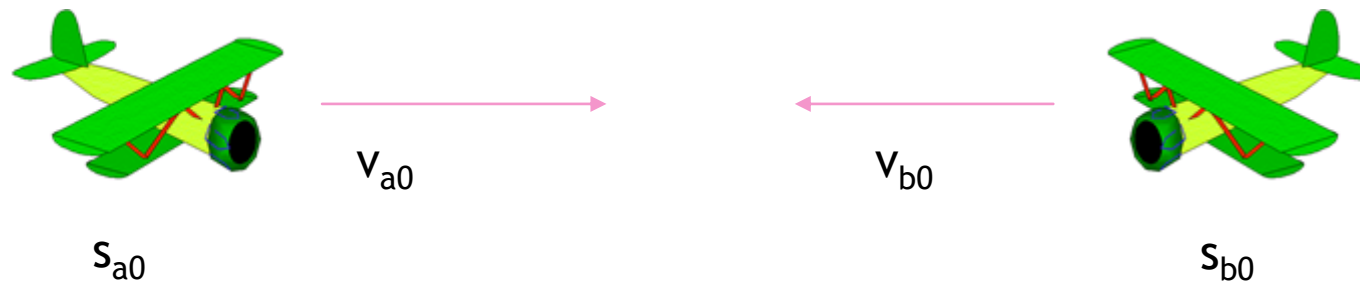
- ▶ Second order polynomials may appear to present more accurate prediction.
 - ▶ They are more computationally expensive than order one polynomials.
 - ▶ They may not be better at predicting position. For example, acceleration may be changing frequently making this parameter almost useless in calculations.
- ▶ It may be wise to allow a choice of which polynomial order to use on a per entity basis at run-time.
 - ▶ This provides the greater benefit of lowering bandwidth requirements while aiming to preserve as much consistency as possible.
 - ▶ Consider the example of an object with highly volatile acceleration, but relatively normal velocity (i.e., acceleration is oscillating to maintain a velocity, like an aeroplane autopilot).

Dead Reckoning: Prediction

- ▶ Sometimes polynomial prediction is not appropriate.
 - ▶ For example, a tank traveling down a road or a particular maneuver carried out by an aircraft.
- ▶ In the above examples we can predict the movement of an entity by identifying some preset maneuver that an entity will be participating in for some time.
 - ▶ This makes prediction a little easier.
- ▶ Sometimes polynomial prediction in 2D will suffice for 3D environments.
 - ▶ This is especially the case when objects are traveling along the ground.

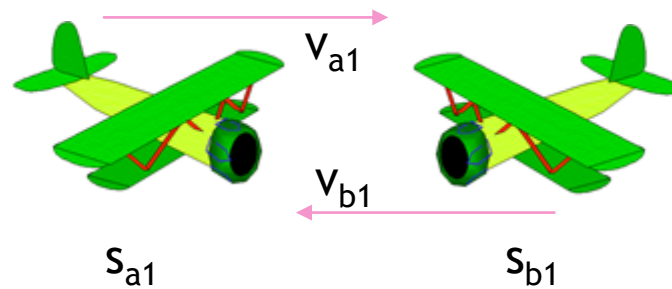
Dead Reckoning: Convergence

- Consider a scenario where we are playing a game against another player; both of us control planes, A and B (e.g., War Thunder)



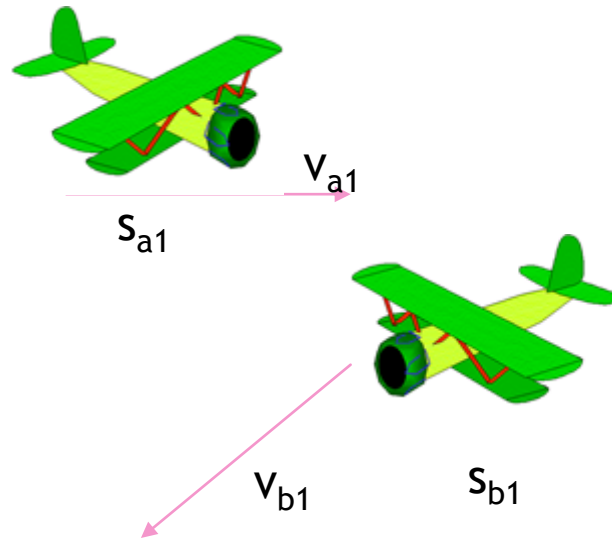
Dead Reckoning: Convergence

- ▶ Let us consider the scenario unfolding from the perspective of our client machine (governing plane A).
- ▶ Our system performs an update - based on input data for A, and based on dead reckoning for B:



Dead Reckoning: Convergence

- ▶ Except... Let's consider that same update for plane B:



Dead Reckoning: Convergence

- ▶ Plane B adjusted its velocity to evade us (which makes sense!)
- ▶ But now we have two conflicting realities, one for each client.
- ▶ And they are a scenario we can't rationalise while maintaining believability - because we waited too late to consider the problem.

Dead Reckoning: Convergence

- ▶ Often, approaches to avoiding this conflict of world view are design based
- ▶ We could limit angular velocity of the planes, so evasion would be impossible once inside a reasonable latency threshold
- ▶ We could simply force the clients to assume the planes survived until the server dictated otherwise
- ▶ Consider the example of the cast-bar in an MMO
- ▶ Can also employ frequent state regeneration for very specific variables - like `shouldIBlowUp`.

Dead Reckoning: Convergence

- ▶ Less dramatic scenario: small adjustment to attitude in flight, B moving slightly out of formation with A
- ▶ We can simply snap to the corrected position when the server next updates (with a DR correction for the time since the update was first received from client B)
- ▶ Breaks believability
- ▶ Some MMOs do this, specifically to address lag-spikes (where believability has already been lost)
- ▶ More often, we'll interpolate and slowly correct our position - or move towards correctness.

Dead Reckoning: Convergence

▶ Linear Interpolation:

- ▶ Identifies convergence point at some position in the future, based on the latest predicted path.
- ▶ Linearly animates an entity from its current position to the convergence point.
- ▶ Better than zero order when large corrections are required but may still promote unnatural movement as entities suddenly change direction.

▶ Splining

- ▶ Generate a smooth curve of motion which tends us towards where we believe we ought to be
- ▶ Computationally expensive, but least 'jarring' for the player

Dead Reckoning: Eventual Consistency

- ▶ All of these approaches to correcting conflicting world views lead us towards consistency
- ▶ But we never reach it, because by the time we have, someone's already moved again
- ▶ Only time we ever reach it is if everything stops moving and interacting
- ▶ As with our linear solver in our physics engine.

Implementation

- ▶ Consider the concept of dead reckoning.
- ▶ Explore the implementation of various orders of dead reckoning (and convergence) in your project.