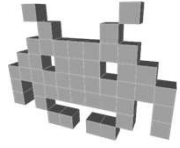
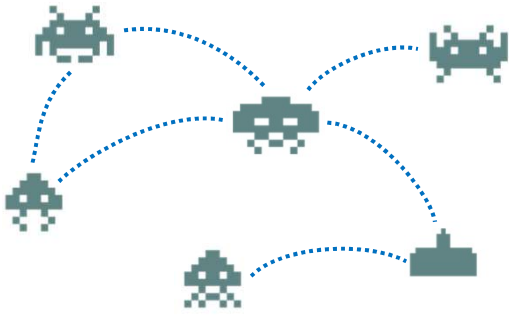


3D VideoGames
Università degli Studi di Milano

Networking for 3D Games




Marco Tarini



1

Course Plan



- lec. 1: **Introduction** ●
- lec. 2: **Mathematics** for 3D Games ●●●●●●
- lec. 3: **Scene Graph** ▢▢
- lec. 4: **Game 3D Physics** ▢●●●● + ●●
- lec. 5: **Game Particle Systems** ▢
- lec. 6: **Game 3D Models** ●
- lec. 7: **Game Materials** ●
- lec. 8: **Game Textures** ●●
- lec. 9: **Game 3D Animations** ●●
- lec. 10: **3D Audio** for 3D Games ●
- lec. 11: **Networking** for 3D Games ●📍
- lec. 12: **Interactive Agents** for 3D Games ●
- lec. 13: **Rendering Techniques** for 3D Games ●

For a more in-depth discussion of many of the subjects of this lecture, see the courses
«[Online Game Design](#)»
«[Computer Networks](#)»
&
«[Networks Science](#)»



★
bridge lectures

2

Player 2 has joined the game

(game design perspective)

- Types of multiplayer games
 - Hot-seat
 - players time-share
 - Local multiplayer (Side-to-side)
 - e.g., split screen
 - players share a terminal
 - **Networked**
 - each player on a terminal
 - terminals connected...
 - ...over a LAN
 - ...over the internet




3

Player 2 has joined the game

(see course on: Online Game Design)

- **Multiplayer** game types, according to **gameplay**
 - collaborative
 - competitive
 - versus
 - teams...
- *How much* multiplayer?
 - no: single player
 - 2 players?
 - 10 players?
 - >100?
 - > 1000? } («massively» multiplayer online, **MMO**)



4

Networking in 3D Games



Objective: remote players *see* and *interact with* a **common** 3D virtual world



The technical challenge:
how can this illusion be achieved?

5

Networking in Games



- One of the tasks of a Game Engine
- Scenarios can differ:
 - **number of players**? (2, 10, 100, 100.000?)
 - game **pace**? (real time action ≠ chess match)
=> tolerance to **latency**?
 - **joining ongoing games** : allowed?
 - **cheating** : must it be prevented?
 - **security** : is it an issue? (e.g. DoS attacks)
 - **channel** : LAN only? internet too? Bandwidth needed?

6

Key technical choices for a networked-game

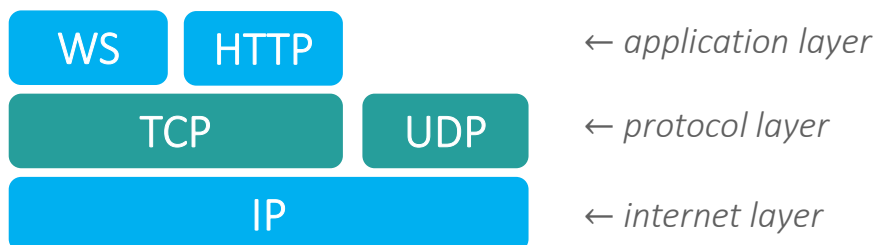


- **What** to communicate?
 - e.g.: complete statuses, status changes, inputs...
- How **often** ?
 - at which rate
- Over which **protocol** ?
 - TCP, UDP, WS ...
- Over which **network architecture** ?
 - Client/Sever, Peer-To-Peer
- How to deal with networking problems
 - **latency** ("lag") <== one main issue
 - limited **bandwidth**
 - loss of individual packets
 - connection loss

7


From the start: Protocols

(see course on: Computer networks)



8


Protocol layer



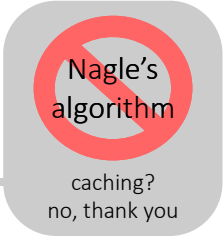
TCP sockets	UDP sockets
<ul style="list-style-type: none">● Connection based● Guaranteed reliability● Guaranteed ordering● Automatic splitting of data into packets● Flow actions● Ease of use: “feels like reading & writing data to a file”	<ul style="list-style-type: none">● <i>What’s a connection?</i> 🙄● No reliability● No ordering● Split your data yourself● No automatic flow actions● Hard: must detect and deal with problems explicitly

9

UDP vs TCP



- The problem with **TCP**: too many *strong* guarantees
 - they cost a lot in terms of latency (“lag”)
 - not designed for time-critical applications
 - if it must be used, at least enable the option `TCP_NODELAY`
- The problem with **UDP**: not enough guarantees
 - no concept of connection: no timeouts, no handshake, a port receives from anyone
 - List of guarantees:
 1. “a packet arrives either whole, or not at all”. The end. End of list.
 - packets can arrive...
...out of order 🤪 , ...not at all 🤪 , ...in multiple copies 🤪



Nagle's algorithm

caching?
no, thank you

10

UDP vs TCP

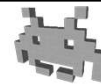


- Problem with **TCP**
 - too many costly guarantees
- Problem with **UDP**
 - not enough guarantees
- The hard way:
 - use **UDP**,
but *manually re-implement the needed guarantees*
 - best, for the most challenging scenario
 - fast paced games, not on LAN



11

Virtual connections over UDP: how-to (notes)



- add **connection ID** to packets
 - to filter out unrelated ones
- **time out** on prolonged silence (~ few secs)
 - declare "connection" dead
- add **serial number** to packets
 - to detect when one went missing / is out of order / is duplicate
 - (warning: int numbers *do* loop – solutions?)
- give **ack** back for received packets
 - optimize for lucky (& common) cases!
 - N (say 100) received msg == 1 ack (with bitmask)
 - resend? only a few times, **then give up (data expired)**
- congestion avoidance: measure **delivery time**
 - tune send-rate (packets-per-sec) accordingly
- obviously: **NON blocking** receives!

what **TPC**
doesn't
understand

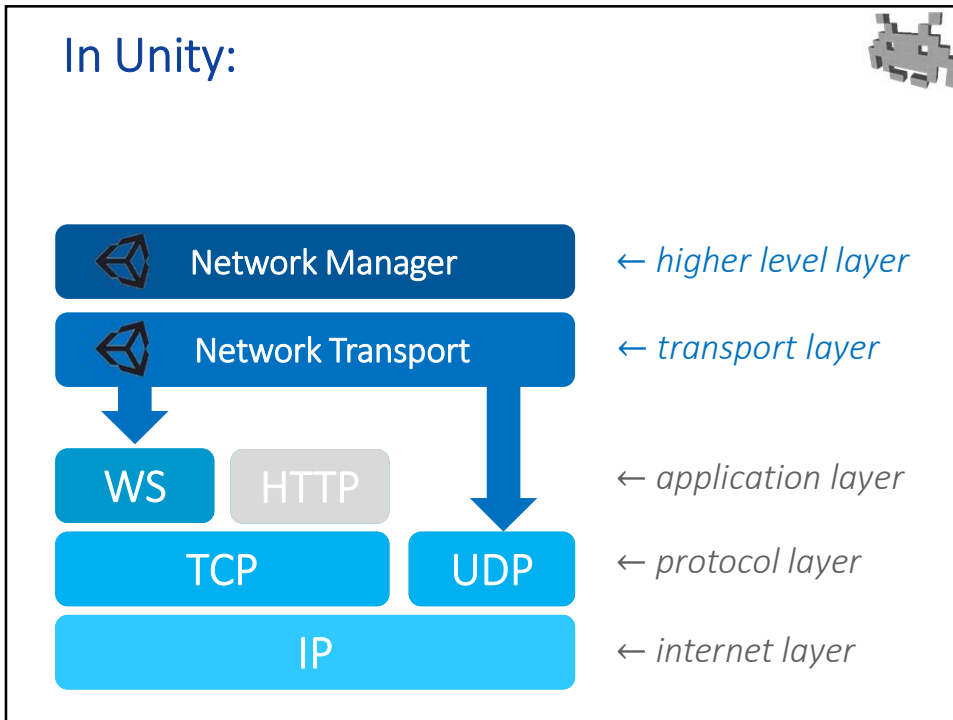


12

Choosing a protocol for a given pacing

- After all, it is a question of pacing
 - fast paced game?
 - action games, FPS, ...
 - (sync every 20-100 msec)→ UDP is a good match (unless LAN only)
 - slow paced game?
 - RTS, RPG...
 - (sync every ~500 msec)→ can get away with TCP
 - slower paced games?
 - MMORPGs, cards ...
 - (sync every few sec)→ why not just HTTP
 - traditional turn based ?
 - chess, checker
 - (sync every hour/day)→ may as well use EMAIL

13



14

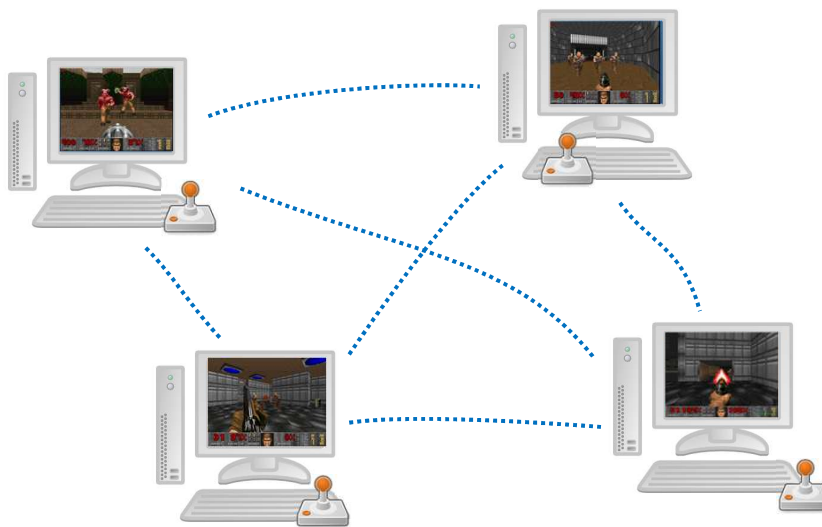
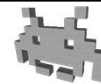
In Unity:



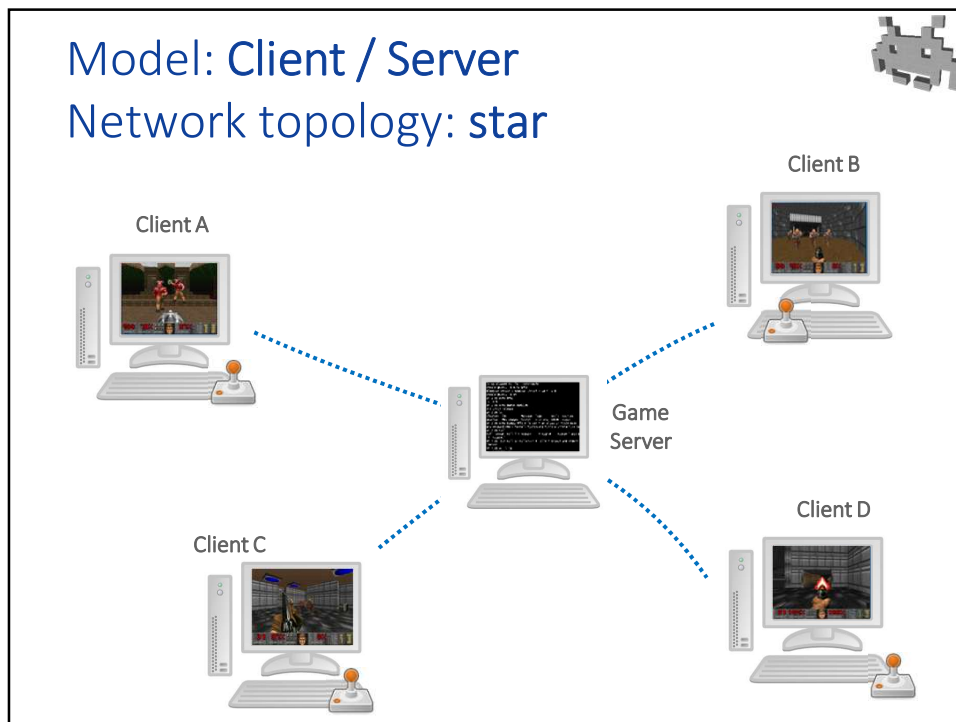
- Low level: **Transport Layer**
 - Implements guarantees over UDP (virtual connections)
 - As easy to use as TCP, but designed for games
 - see how-to notes list above
 - Can work over WS instead UDP (abstracts the differences)
 - WS is used for web games (the one with graphics on WebGL)
- Hi level: **Network Manager**
 - manages network connectivity
 - by default: “client hosted” games
 - the server is one of the players
 - controls shared state of the game
 - deals with clients
 - sends remote commands

15

Model: Peer-to-peer Network topology: complete



16



17

Networking paradigms for games -
we will see these ones:

- Deterministic Lockstep on P2P
- Deterministic Lockstep
- Game-Status Snapshots
- Distributed Physics (just notes)
- Game-Status Snapshots with Client-Side predictions
- Cloud gaming

on client-server

The diagram lists six networking paradigms for games. The first item is 'Deterministic Lockstep on P2P'. The next two are 'Deterministic Lockstep' and 'Game-Status Snapshots'. The next two are 'Distributed Physics (just notes)' and 'Game-Status Snapshots with Client-Side predictions'. The last item is 'Cloud gaming'. A large curly bracket on the right side groups the last four items under the label 'on client-server'.

18

A preamble:

- Pro-tip: *every game is a turn-based game*
 - action games just have... (e.g.) 60 “turns” per second
 - BTW this is common knowledge between gamers, e.g.:
“In Dark Soul 3, when you roll-on-ground...
agility = 105 → 15 invincibility frames
agility = 90 → 12 invincibility frames.”



19

actions and Agent: example (a useful abstraction)

```
void Player::maybe_fire_gun(){  
    if (gun_cooldown > 0) return;  
    if (ammo == 0) return;  
    gun_cooldown = FIRE_RATE;  
    ammo --;  
    // Spawn new bullet object.  
    // Activate particle system.  
    // Trigger sound effect.  
    // Etc  
}
```

```
// keyboard callback  
void on_key_press( int keycode ) {  
    if (keycode == FIRE_KEYCODE) {  
        hero.maybe_fire_gun();  
    }  
    ...  
}
```

An example of something which changes the state of the game / the physics


Callback function:
[event-based programming]
a function invoked by the system in response to an (external?) event. Here, key-presses.

BAD IDEA

game logic performed asynchronously

20

actions and Agents: example a useful abstraction



```

class Actions {
    bool is_pressed[ N_ACTIONS ];
};

void on_key_press( int keycode ){
    if (keycode == FIRE_KEYCODE) {
        actions.is_pressed[ FIRE ] = true;
    } ...
}

void on_key_release( int keycode ){
    if (keycode == FIRE_KEYCODE) {
        actions.is_pressed[ FIRE ] = false;
    } ...
}

void physics_step( int keycode ){
    if ( actions.is_pressed[ FIRE ] ) {
        hero.maybe_fire_gun()
    }
    // rest of game logic / dynamics
}
    
```


Callback functions:
(asynchronous!)

Called every
phys frame
regardless of
actions

GOOD IDEA

21

Actions a useful abstraction

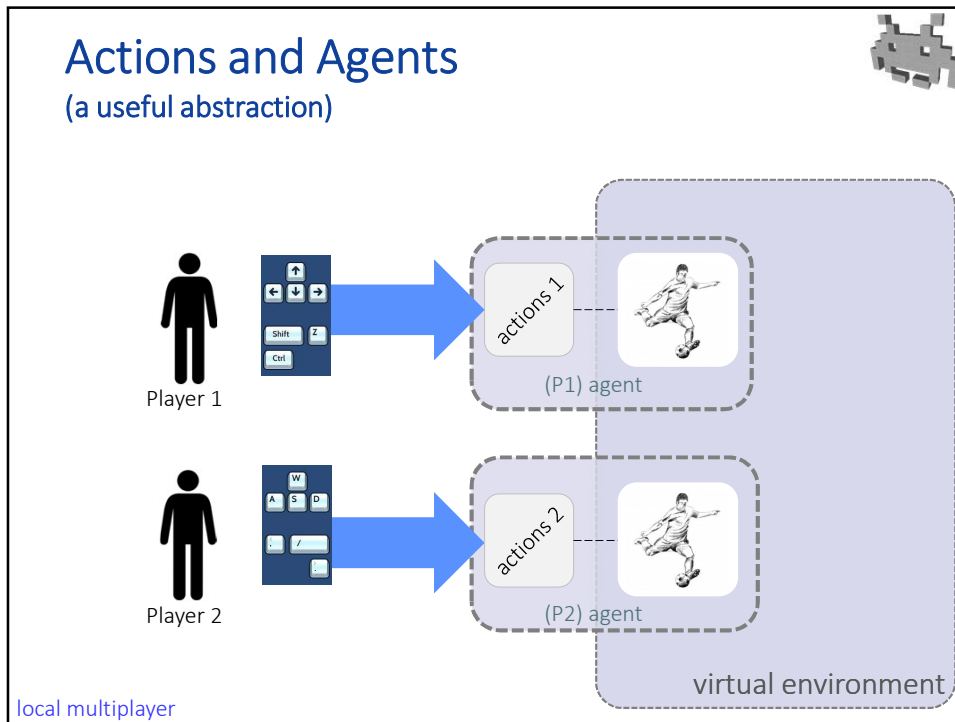


- Good design pattern for videogames
 - even for single player games: keep game evolution reproducible / easy to actions
 - multiplayer games: one set of “actions” object per player
- Good for...
 - Abstracting game logic from key-bindings & input devices (e.g. game actionslers VS keyboard etc)
 - Same game logic for local multiplayer games & networked games (networked = actions are sent over – see next)
 - Same game logic for Playing Characters & NPC-AI-actionsled NPCs (NPC’s AI = procedurally generated actions) (NPC as “simulated player”)
- “Actions” structure are light!
 - e.g. (depending on the game): 8 bits = 1 byte
 - Convenient to communicate (v. low bandwidth)
 - Can be stored for playbacks (e.g., 1 byte per frame = under 2Kb per min!)
 - Used (for example) in “attract mode” of old coin-ups – even them could afford it!

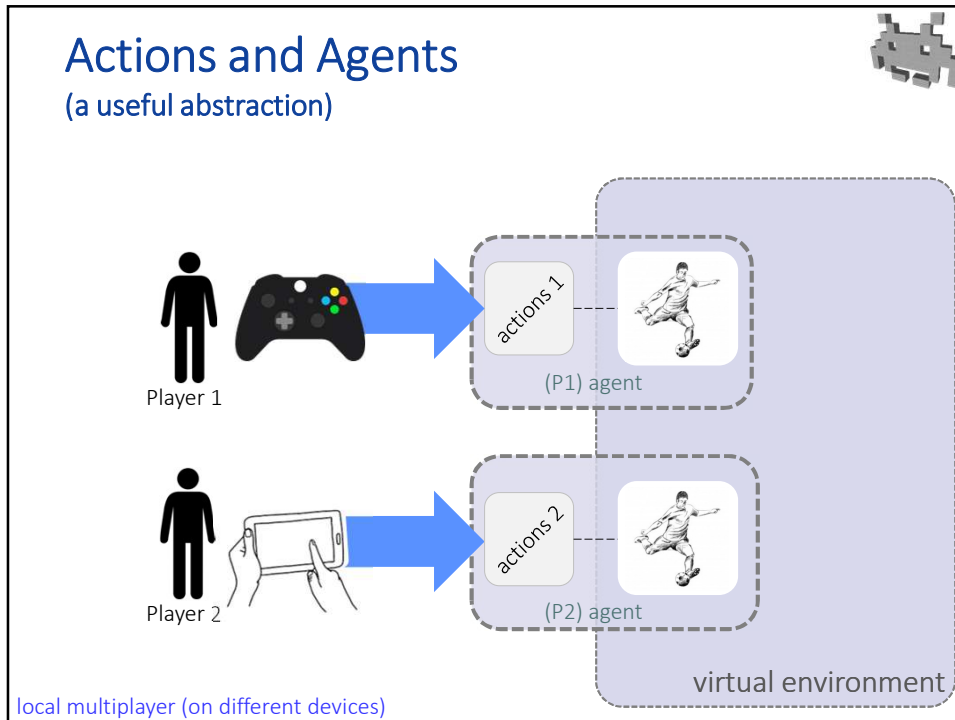
what’s relevant for today’s lecture

assuming determinism & reproducible initial conditions

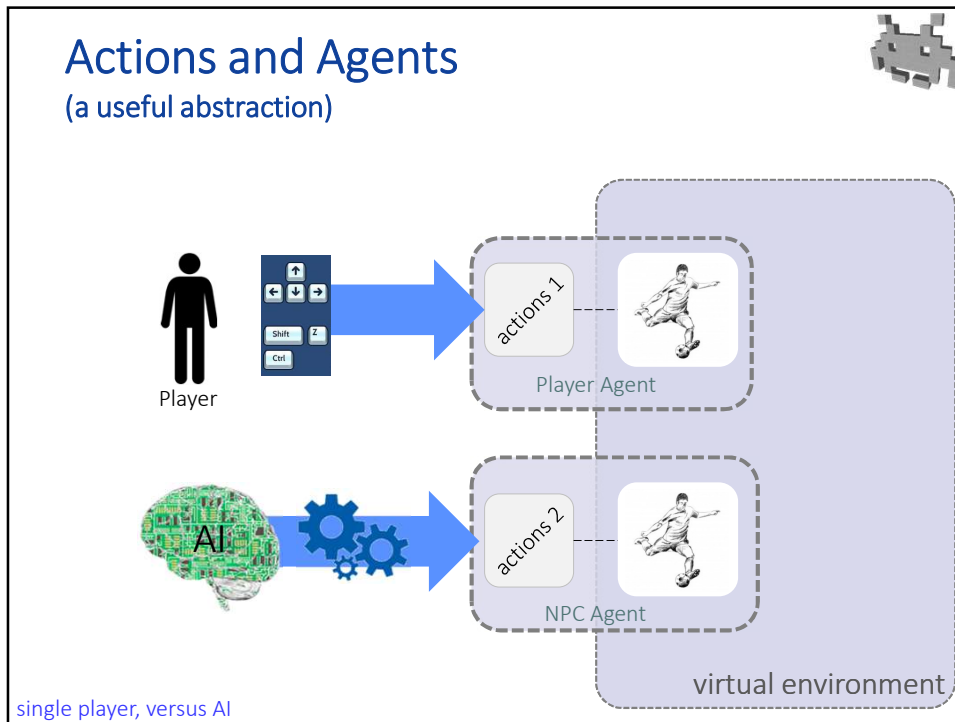
22



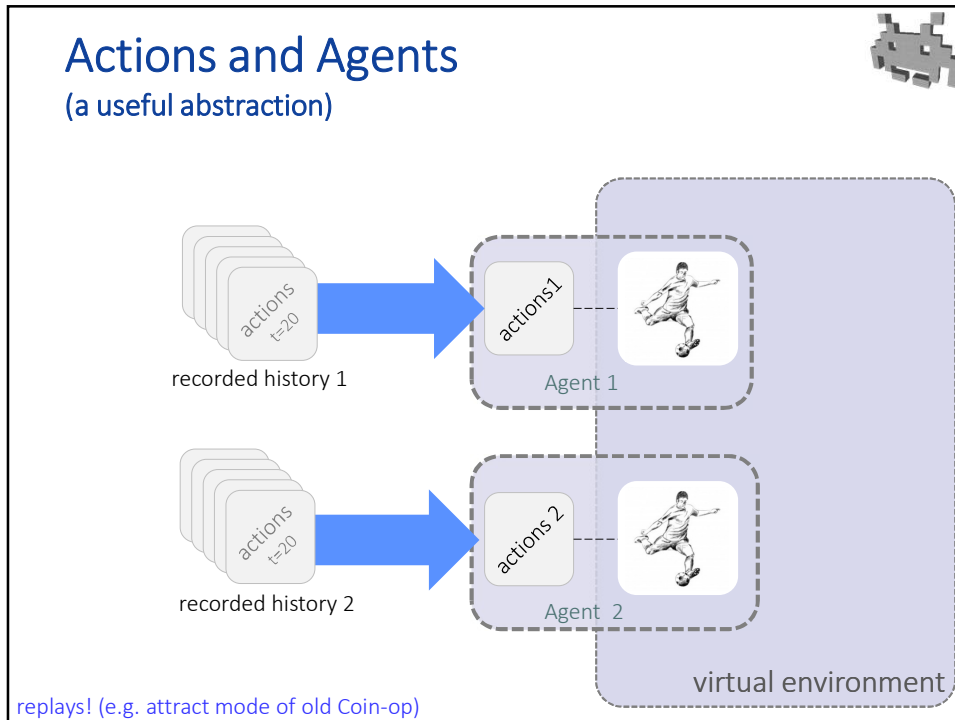
23



24



25



26

Digression: “attract-mode” in old coin-op arcades



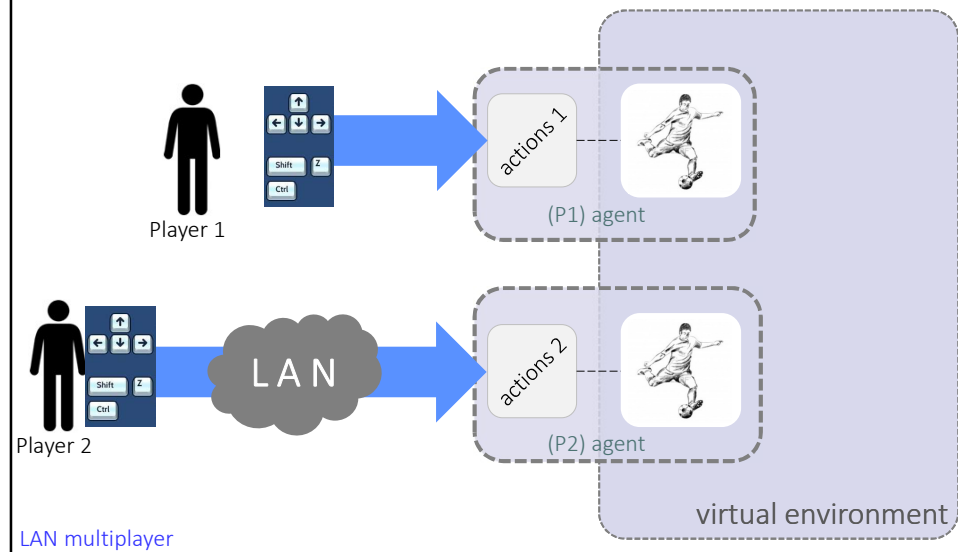
Ghost'n Goblins, 1985 Capcom



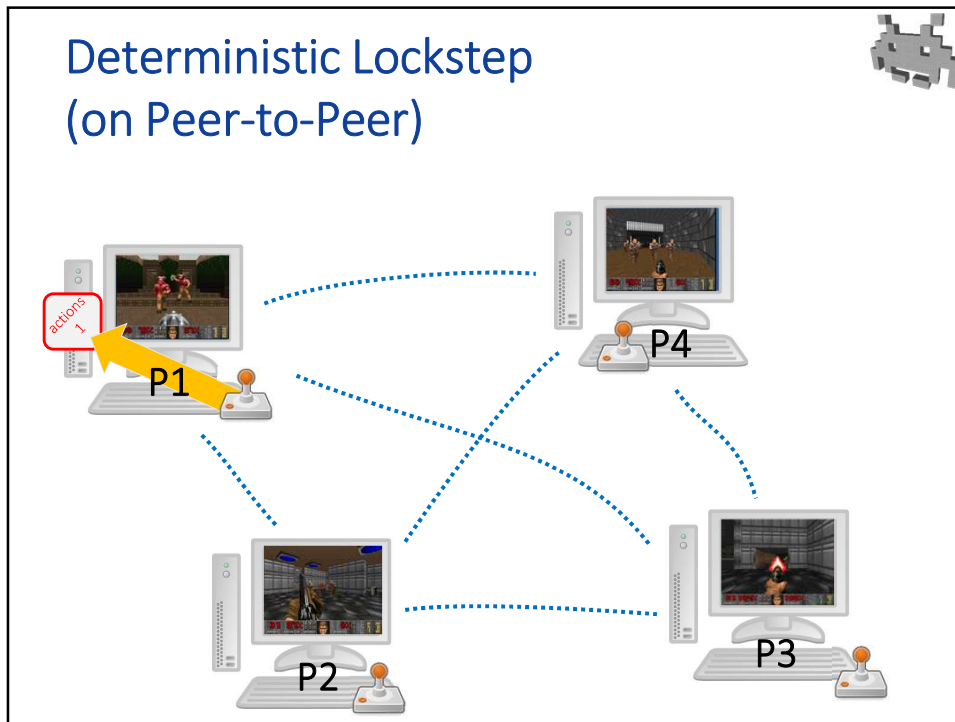
Bubble Bubble, 1986 Taito

28

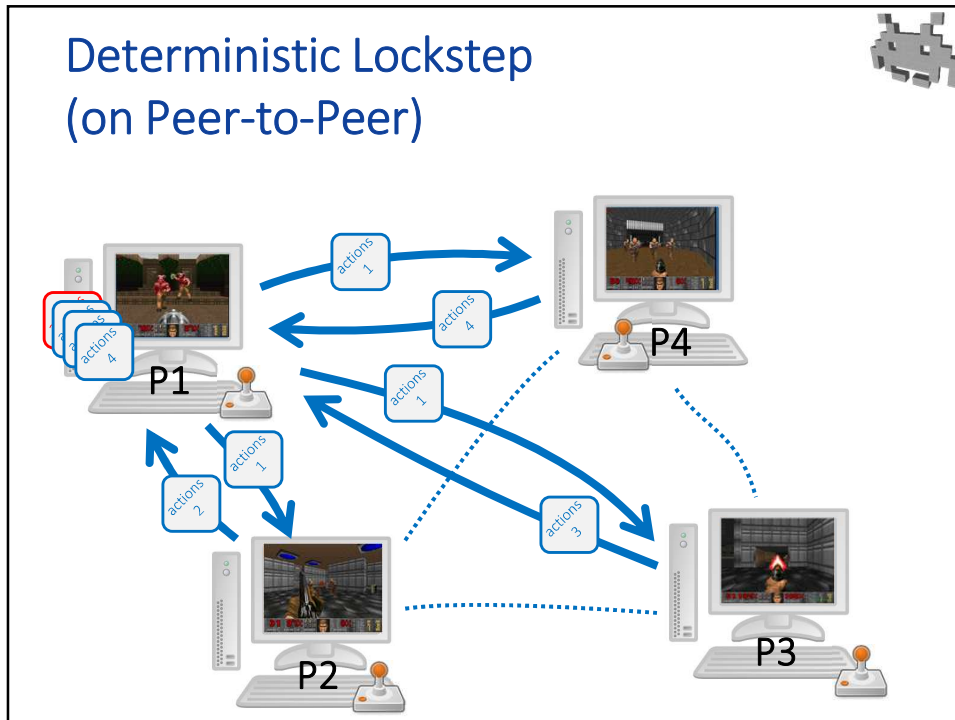
Actions and Agents (a useful abstraction)



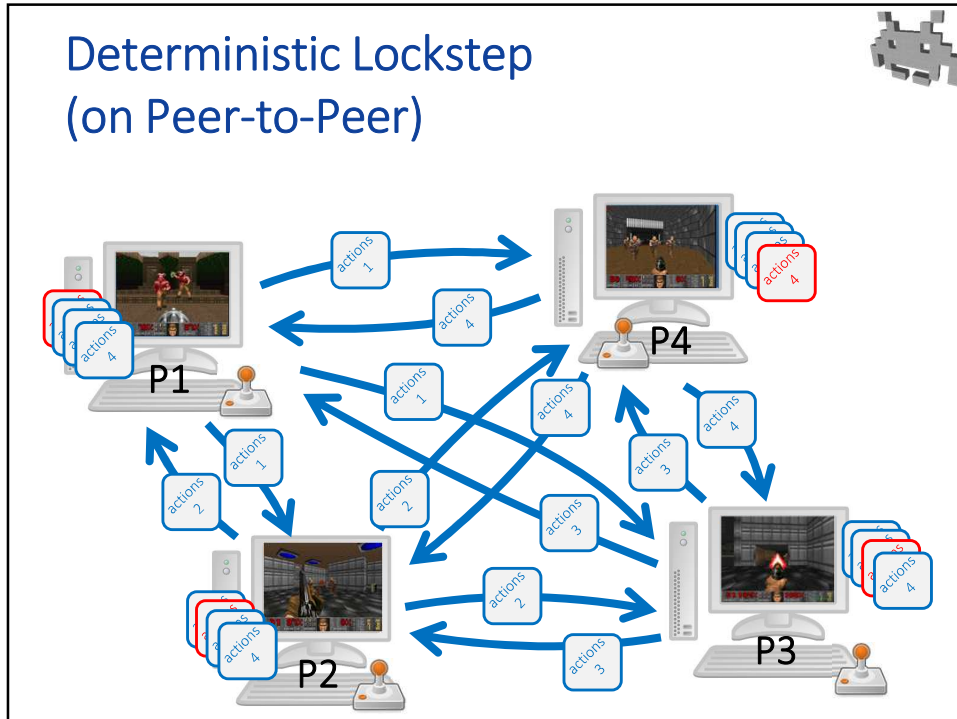
29



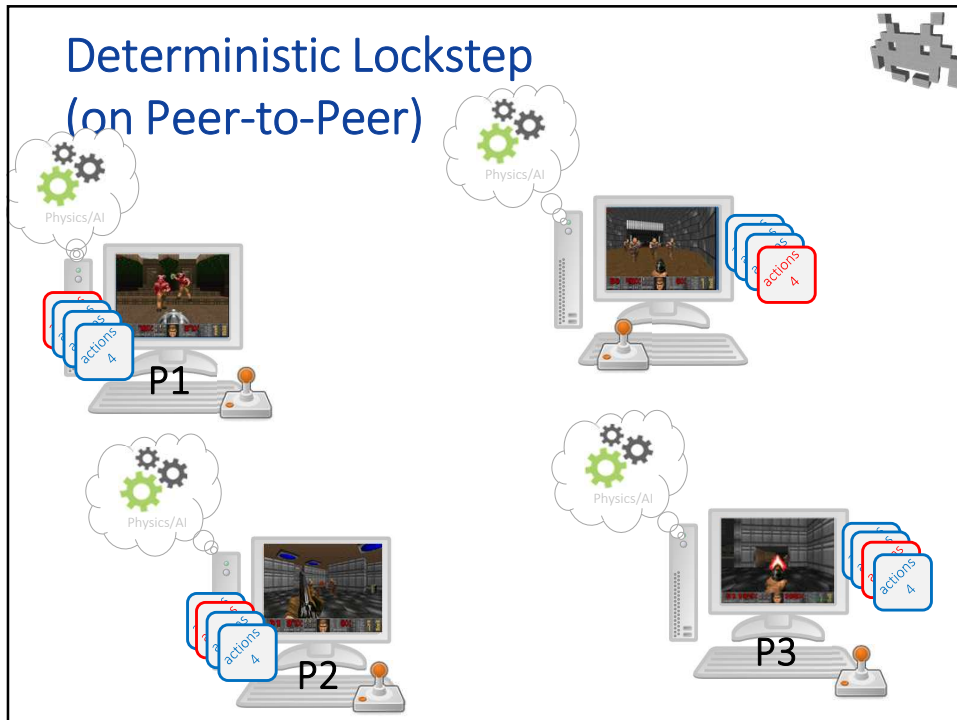
30



31



32



33

Deterministic Lockstep (on Peer-to-Peer)



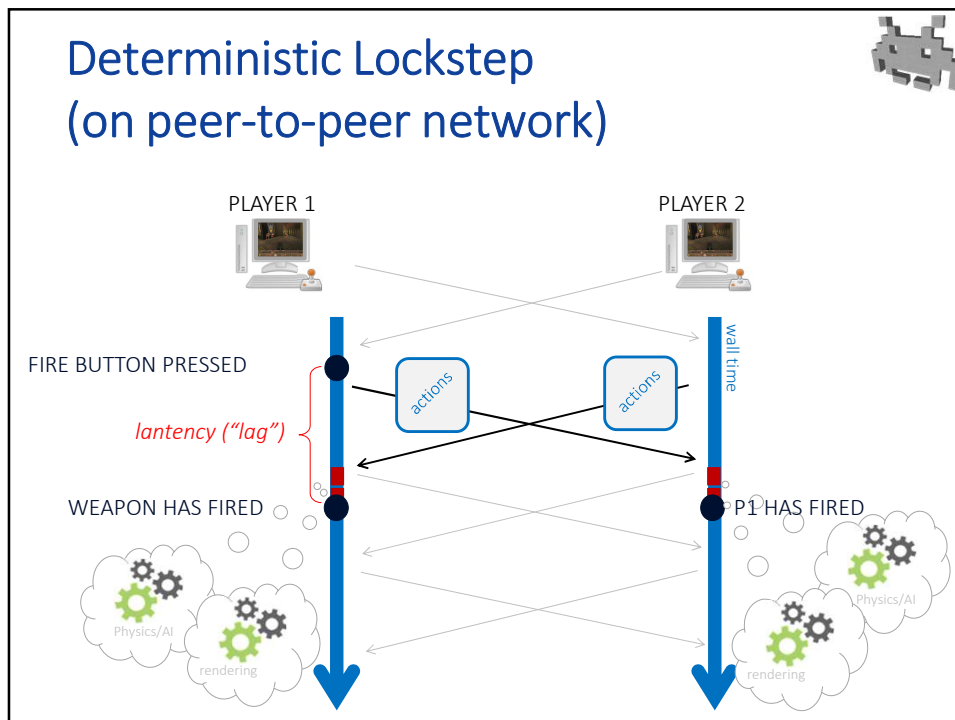
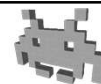
- Game evolution = sequence of “turns”
 - e.g. physics steps (fixed dt !)
- Each node sends its current *actions* (inputs)
 - to everybody else
- After all actions are received, each node computes its own evolution
 - deterministically:
same input \rightarrow same result

even if
independently computed

in Italian:
lockstep = “*pari passo*”

34

Deterministic Lockstep (on peer-to-peer network)

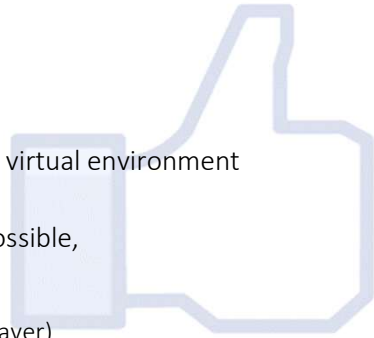


35

Deterministic Lockstep: the good



- elegant and simple! ☺
- minimal **bandwidth** needed
 - only sent data = actions
 - compact! (e.g., a bitmask)
 - does not depend on complexity of virtual environment
- **cheating**: inherently limited
 - but a few ways to cheat are still possible, e.g.:
 - aim-bots (unlawful assist from AI)
 - x-rays (unlawful reveal of info to player)
- mixes well with:
 - non-cheating AI, replays, player performance recording...
- can use simple **TCP connections**
 - because we need 0% packet loss anyway (but...)



36

Deterministic Lockstep: can as well use TPC instead of UDP ?



- why yes:
 - **TPC** is simple to use
 - It takes care of everything
 - works well, when no packet loss
 - on loss, we need resend it anyway: let TPC do that
 - makes little sense to use **UDP** and then...
try to re-implement all **TPC** over it
 - at the beginning of dev,
UDP is a (premature) optimization
- why not:
 - to degrade better with packet loss
 - e.g.: use redundancy – instead of resend-on-failure
 - actions are small: send 100+ actions in every packet
 - keep resending until ack received

37

Deterministic Lockstep

- Common, e.g., in:
 - RTS
 - actions = orders
 - can be fairly complex
 - but game status = much more complex
 - first generation FPS
 - actions = [gaze dir + key status]



Starcraft Blizzard 1998-2015



Command and Conquer
EA / Westmany et al
1995..2012



Age of Empires
Ensemble Studios et al,
1998..2015



Doom ID-soft, 1998



...why not anymore?


39

Deterministic Lockstep on peer-to-peer: the limitations


- latency:
 - input-to-response delay of 1 x delivery time (even locally!)
 - (you cannot act immediately even on your own local input)
- does not scale with number of players
 - quadratic number of packets
 - 2P ok, 100P not ok
- input rate = packet delivery rate
- delivery rate = as fast as the slowest connection allows
- if connection problems (anywhere): everybody freezes!
- assumes full agreement on initial conditions
 - not problematic, even with procedural levels etc (just use the same randseed)
 - but...
- joining ongoing games: not possible
 - Barring ad-hoc expensive solutions
- assumes complete determinism!
 - this can be problematic (see next)

40

Deterministic Lockstep on peer-to-peer: the limitations




- **latency:**
 - it's incompressible!
 - even on the local client, you are not allowed to react to your own player's inputs right away
 - Need to wait for others actions to arrive!
- on **connection delay** => everybody freezes!
 - Cannot assume "no new action by that player" on timeout (how to know that other clients didn't receive the msg either?)
 - Cannot even kick the unresponsive player out (how to know that others client will do the same?)



41

Determinism: what can break it



- Pseudo-**Random**? → not as problem
 - fully deterministic (just agree on the seed)
- ⚠ **Physics:** many preclusions and traps
 - ⚠ variable time step? **bad**
 - ⚠ time budgeting? **bad**
 - ⚠ hidden threats:
 - order of processing of particles/constraints
- ⚠ anything that depends on **clock**?
→ **poison** to determinism
- ⚠ **GPU** computations? **very dangerous**
 - slightly different outcome on each GPU model
- ⚠ **floating point** operations?
 - **hidden dangers**,
 - different hardwired implementations
 - best to assume very little (**fixed point** is 100% safe)
- ⚠ **NOTE:** 99.999% correct == **not correct**
 - virtual world is faithful to reality enough to be *chaotic* → butterfly effect: the tiniest local difference == expect completely different outcomes soon

The entire game system must be designed from the start with "determinism" in mind ...

...and still, it difficult to get (and debug)

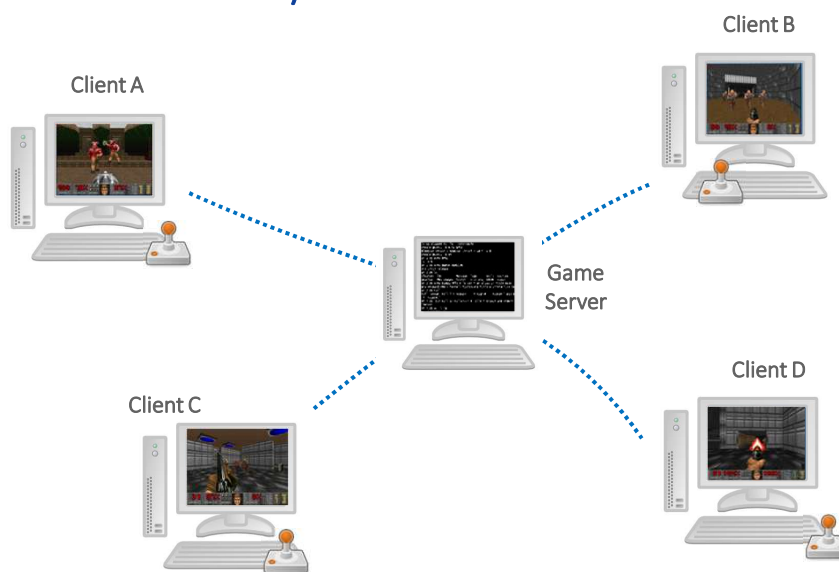
42

A limitation of any peer-to-peer paradigm

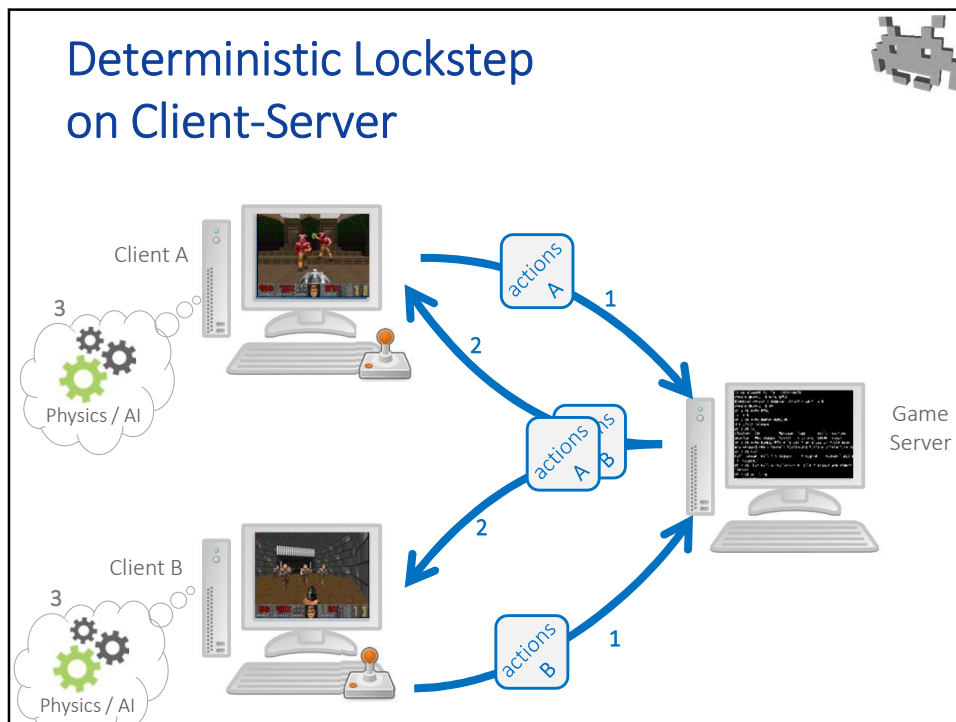
- They **do not scale** with number of players
 - quadratic number of packets necessary
 - 2Players ok, 100Players not ok
- Every client needs to send/receive from all other clients
 - Topology of the network: fully connected
 - E.g. if implemented with TCP: tons of connections
 - Not always practical (firewalls, etc)
- Next: Client / Server paradigms (topology of the network: star)

43

Network topology: Star Model: Client / Server



45



47

Deterministic lockstep on Client-Server

- Server sits on the central node
- Protocol:
 - Each client sends his actions to server
 - Server collects all actions and sends them back to clients
- Advantage:
 - scalability:
number of packets is linear (not quadratic)
- Cost:
 - **responsiveness:**
latency = 2 × delivery time :-O hurts gameplay!
- Bonus: the server can now be made **authoritative**
 - Many new options available. For example...

48

Summary : rules of thumb



- Comparing network layouts
 - **peer-to-peer** :
 - ☺ reduced latency
 - ☹ quadratic number of packages (with number of players)
 - **client-server** :
 - ☹ doubled latency
 - ☺ linear number of packages (with number of players)
 - *REQUIRED*, for num players >> 4-6
 - allows to make the server **authoritative** (see next!)

49

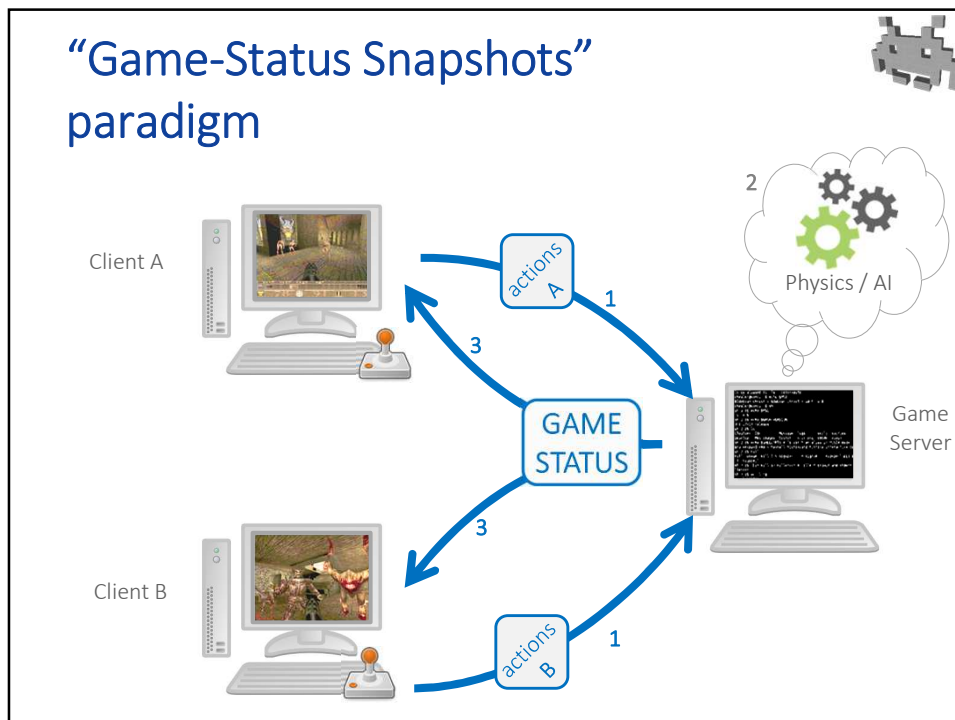
“Server is the man” * (authoritative server). Concept:



- The server has the last word
- For example:
 - Packet loss from player 3?
Server makes up actions for player 3 (instead of waiting for them)
 - Note: server *defines* what player 3 did, not player 3 itself!
 - i.e., clients take server’s word even for *their own actions*
 - Packet loss affects one player only

* Tim Sweeney (Unreal / Epic Games)

50





52

Game-Status Snapshots

- models the current status of the in-game universe, e.g:
 - all the local transforms in the scene graph
 - remember: transforms are light structures!
- sent by the server to each client
 - with optimization such as:
 - send only the modifications from last snapshots
- clients = visualizers of game-status snapshots

54

Game-Status Snapshots

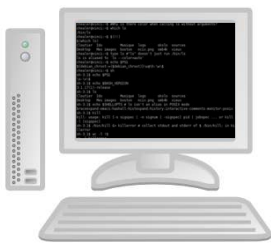



Client:

- just a remote visualizer of the current status
 - status is “read only”
- remote input collector


Server:

- computes evolving status
 - including physics
- it’s where the “real game” runs



55

Game-Status Snapshots

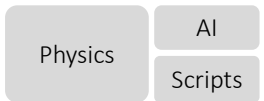
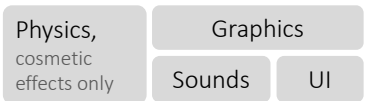


Client:

- connected: to server only
- captures input
- sends actions
- receives game status
 - or relevant portions of it
- renders it
 - using all relevant assets

Server:

- connected: to all players
- receives all actions
 - missing? doesn’t matter
- updates game status
 - physical simulations, etc
- sends current status
 - to all



56

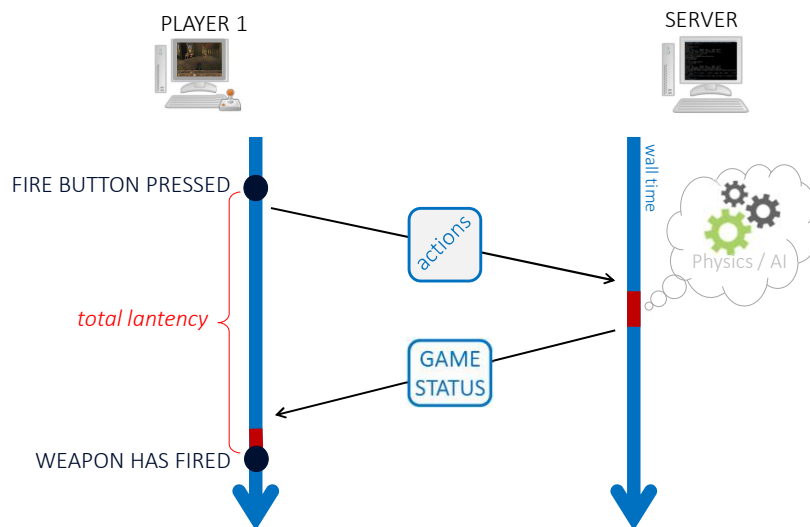
Game-Status Snapshots (compared to deterministic lockstep)

- the gains:
 - determinism: not a required assumption anymore
 - joining ongoing games: becomes trivial
 - packet loss: bearable (hurts that player *only*)
 - to profit: use **UDP**
 - slower connection: bearable (affects that player *only*)
- the losses:
 - packet size: a lot bigger! optimizations, to counter this:
 - compress world status
 - send only the portions of the status which changed or which interest a player
 - **perceived latency:**
 - from input to effect = delivery time :-(
 - from input to visual = 2 x delivery time :-O

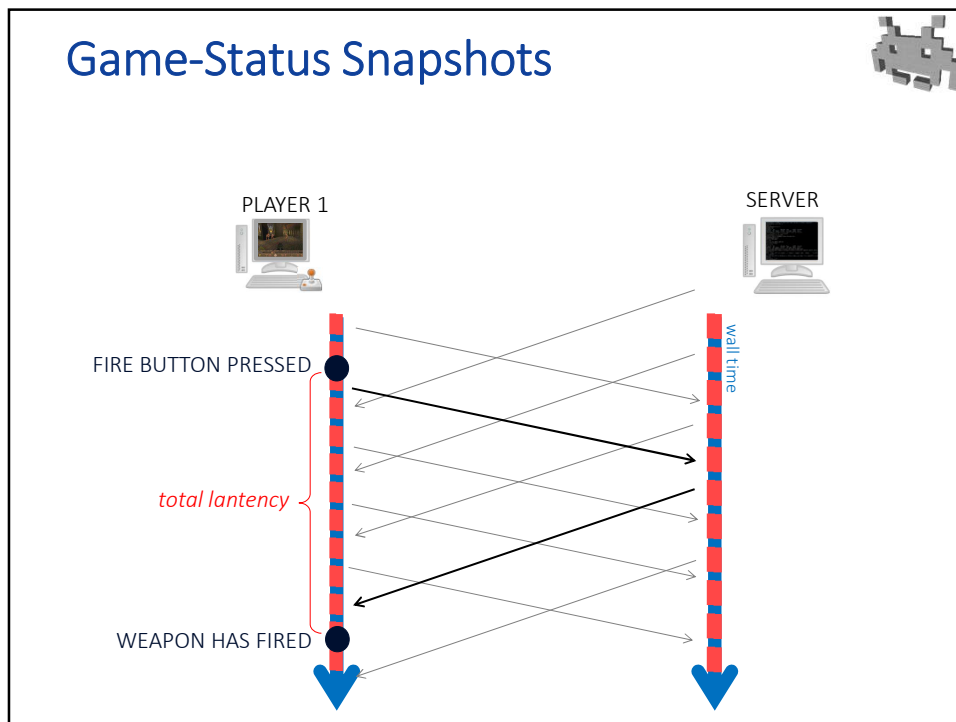
Loss of responsiveness hurts gameplay!

57

Game-Status Snapshots



58



59

Partial Client-side Game Evolution (aka *distributed physics*): the idea

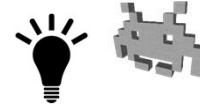
- Each client:
 - in charge for game evolution (including physics) of a part of the scene
E.g. the avatar of his own player, their flying bullets
 - communicates to others
 - a reduced game-status snapshot:
 - describes only status of its parts
 - receives other partial snapshots
 - merges everything up
 - (updates statuses of other players)
- Simple, zeroed latency
 - immediately responsive to local player actions
 - remote agents updated according to "what their client says"
- Problem: can still need determinism
 - (who keeps NPCs / environment in sync?)
- Problem: **authoritative clients**: prone to **cheating!**

Here, we will not investigate this approach any further.

to server,
or, in a P2P network,
to each other peers

62

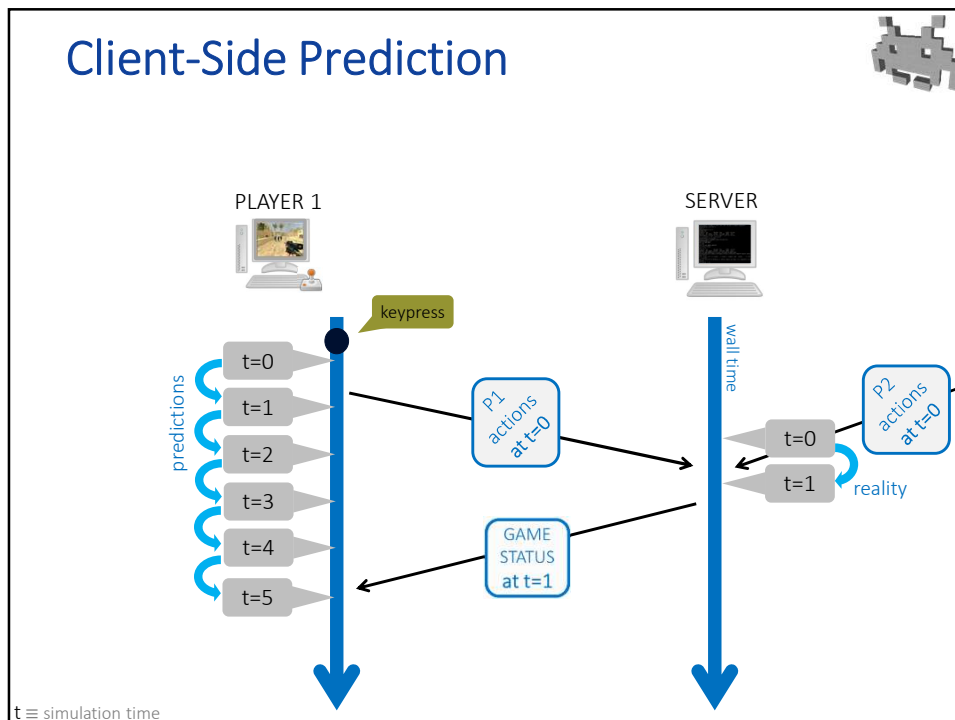
Client-Side Prediction: the idea



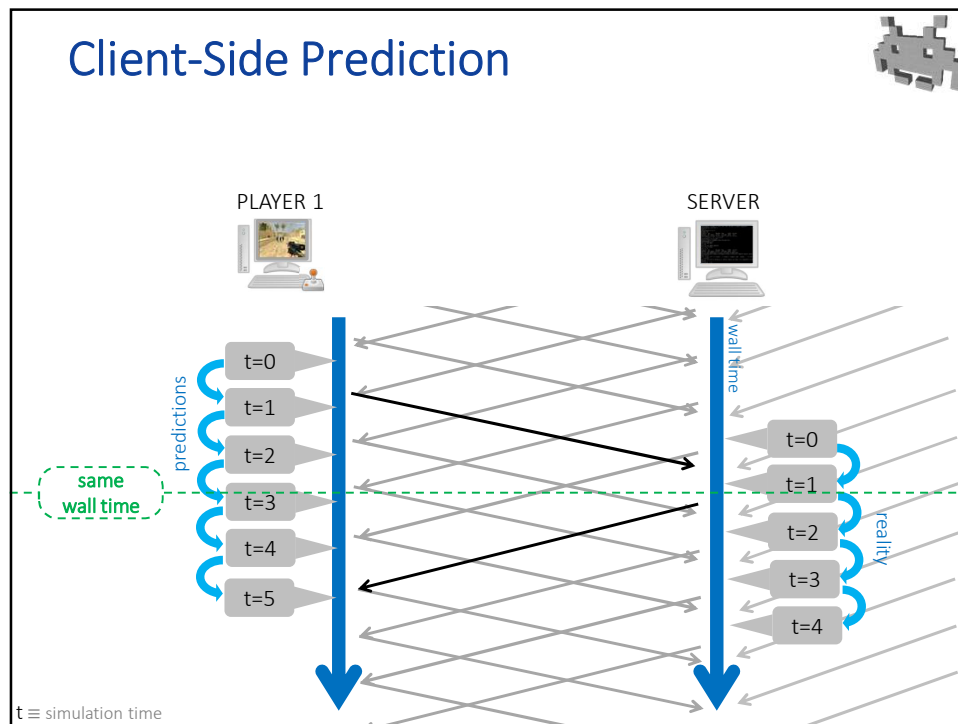
- Client:
 - get Commands from local inputs
 - sends Commands to Server
 - computes game evolution (the prediction)
 - maybe “guessing” other players commands (which it ignores)
 - zero latency!
- Server:
 - receives Commands (from all clients)
 - computes game evolution (the “reality”)
 - server is authoritative
 - prevents many forms of cheating
 - sends Snapshot back (to all clients)
- Client:
 - receives Snapshot (the “real” game status)
 - corrects its prediction, *but only if needed*

63

Client-Side Prediction



64



65

Client-Side Prediction with corrections from the server

- The server-side “actual” simulation lives x msecs in the past of the client-side “predicted” one
 - x = package deliver time
 - remember: simulation time \neq wall time
 - at the same wall time, client simulation time is t , server simulation time is $t - x$
- When server correction arrives to client, it refers to $2x$ msecs ago (for the client)
- Q: how to correct... *the past*?

66

Client-Side Prediction: correction from the server



- Q: How to correct... *the past*?
- A:
 - Replace current state with “correct” new state from server?
 - **WRONG!** that state refers to the past

67

Client-Side Prediction: correction from the server



- Question: how can we correct... *the past*?
- Answer:
 - keep last N statuses in memory
 - including own actions
 - as the “real” status (the correction) of the past arrives from server...
 - ...compare it with stored past status (at corresponding time):
 - does it **match?** ← maybe: within some tolerance
nothing to do
 - does it **mismatch?**
discard frame *and following ones*,
rerun simulation to present (reusing stored actions)

68

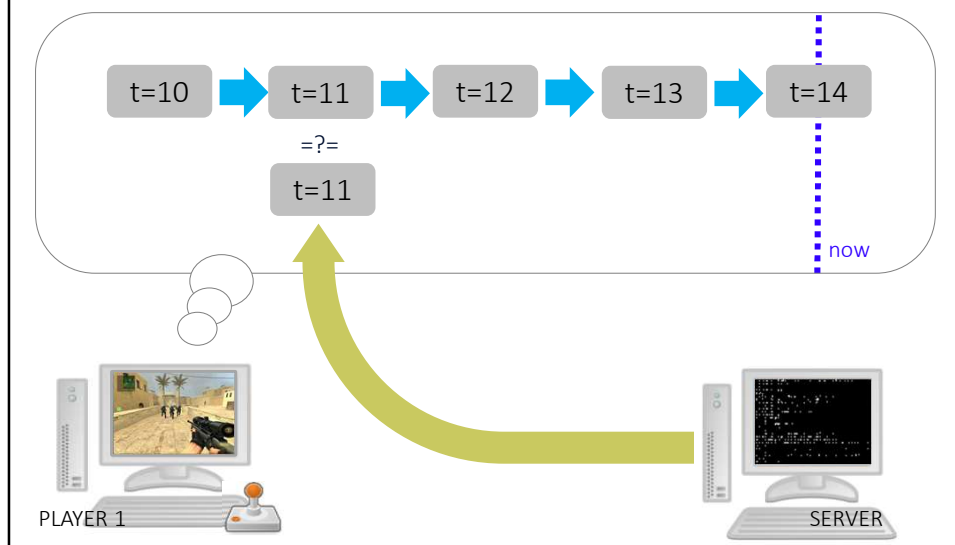
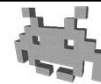
Re-running the physical simulation



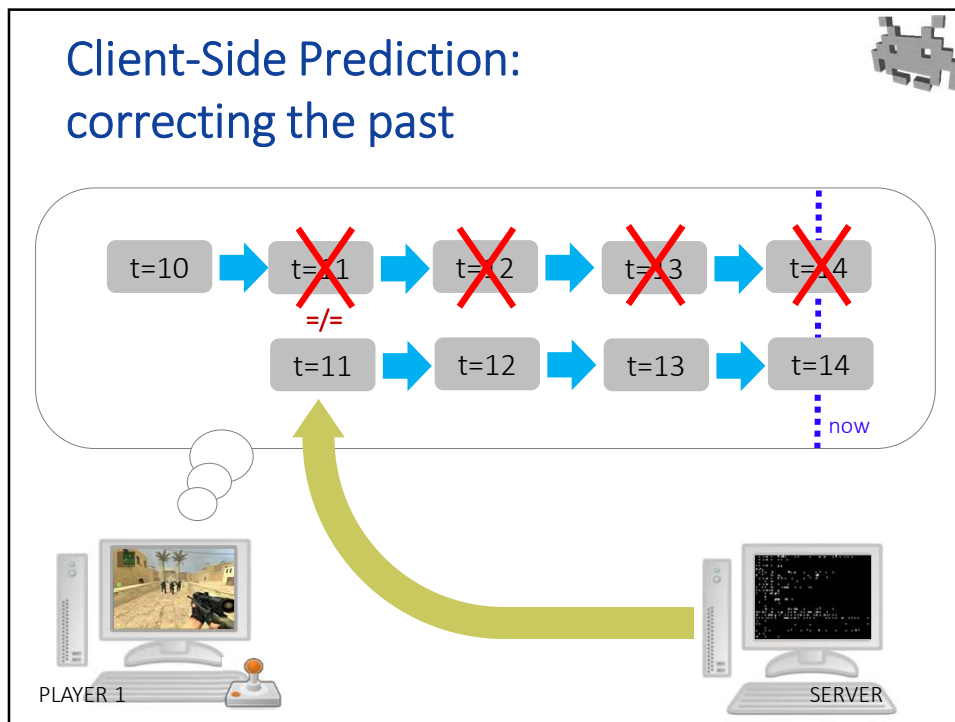
- We just need to catch up with the present
- Physics (and game mechanisms) only
 - no graphics, no sound rendering, no cosmetic physics, particle system...
- We want full speed: can use larger dt if necessary
 - This only compromises accuracy a bit -- tolerable
- Must reuse same actions of own player and other's
 - They must all be cached also
- Note: player is never shown these intermediate steps; only the final result
- The price to be paid: glitches when switching from current present to a different (corrected) present

69

Client-Side Prediction: correcting the past



70



71

Client-Side Prediction: what causes mispredictions?

- Lack of **determinism**
 - e.g., physics was approximated – *soft* real-time!
 - see above for more possible causes of this
- Didn't account that *own* actions were **not received** by server (in time)
 - server: “actually, back then, you didn't jump”
 - authoritative server – server *defines* the truth (even when the client is in a better position to know)
- Didn't account for **other players' actions**
 - server: “actually, back then, that other player jumped”
- Note: none of the above breaks gameplay too much
 - it just causes minor / temporary glitches (usually)

minor/rare issues

commonest cause!

72

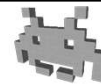
Client-Side Prediction: optimizations 1/2



- reduce snapshots size
(==> to lower bandwidth, increase packet frequency)
 - partial snapshots: refresh more often the parts which are most likely to be predicted wrong / or which changed w.r.t. previous information
 - drastic space reductions!
 - just make sure that every part is eventually refreshed
- reduce correction computation
(==> making corrections quicker)
 - partial physic steps: update only the parts affected by the error
 - use bigger *dt* (fewer steps to get to present)

73

Client-Side Prediction: optimizations 2/2



- tentatively predict also unknown data
(==> to reduce correction frequency)
 - e.g., also predict other player's actions
 - easiest prediction: players do what they did last frame
- trigger correction only when status differ *enough*
(==> to reduce correction frequency)
 - e.g., when any spatial position difference > epsilon
 - tolerate small discrepancies
 - (warning: discrepancies tend to explode exponentially with virtual time – because Chaos)

74

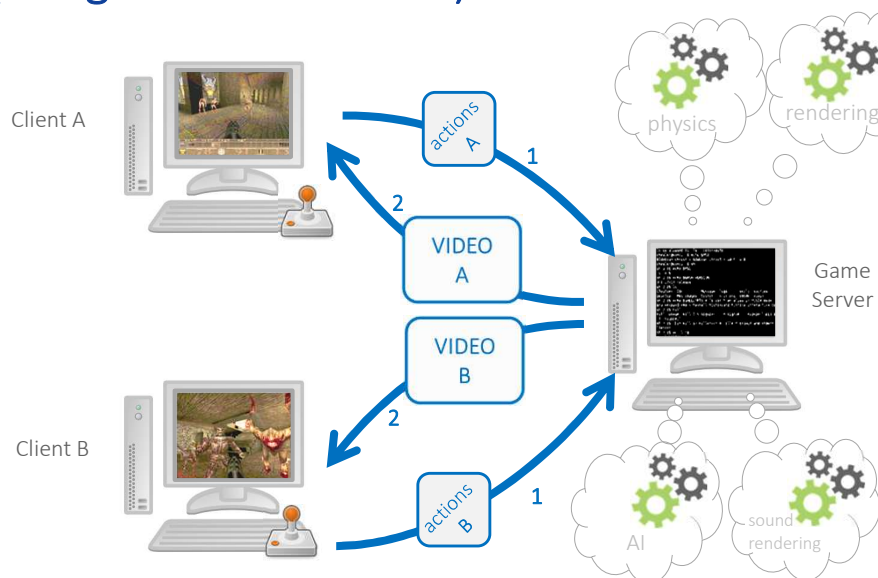
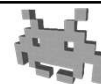
Client-Side Prediction: notes



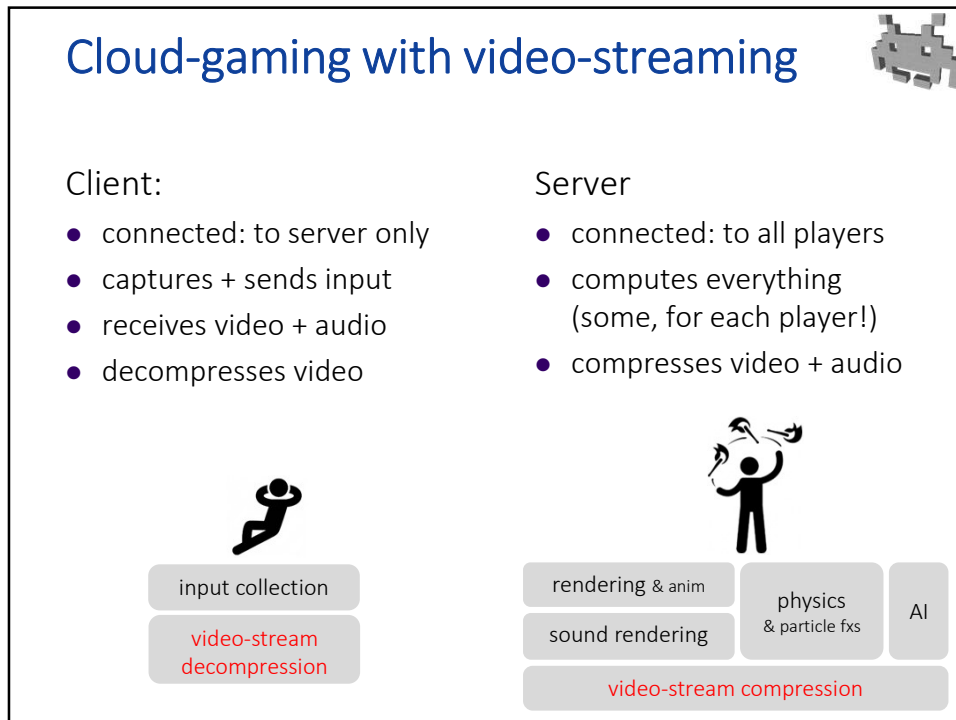
- 😊 No **latency**: immediately react to local input
 - client proceeds right away with next frame
 - *when prediction is correct*: seamless illusion
 - *otherwise*: (minor?) glitches
- 😊 **Determinism**: not assumed
- 😊 **Cheating**: not easy (server is **authoritative**)
 - But still possible for unlawful clients to...
 - ... give their players more informations than lecit (e.g.: transparent walls in FPS, no fog-of-wars in RTS)
 - ... assist their players with automatism (e.g.: autoaim bots)

75

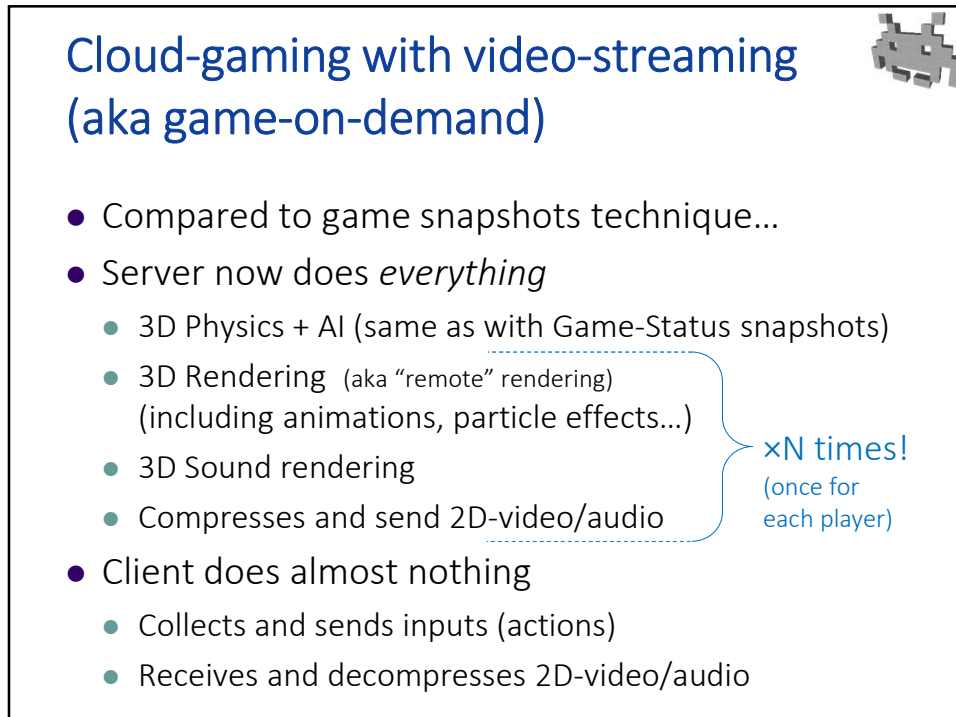
Cloud-gaming with video-streaming (aka game-on-demand)



76



77



78

Cloud-gaming with video-streaming

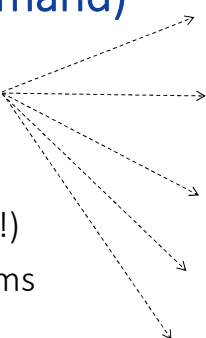
Note: not only for multiplayer games!
Applies to single-player games too
(which wouldn't normally require networking).

- Advantages:
 - client is thin: does almost nothing
 - needed client capabilities are very limited (pads, mobiles, ok)
 - DRM protection: absolute (no asset will be ripped – by sniffing)
 - cheating: basically impossible – including dishonest assists (e.g. “aimbots”) or dishonest reveals
 - game piracy: impossible
- Challenges:
 - Demanding in terms of **bandwidth** (high-res video + audio)
 - Very demanding in terms of server workload
 - **Latency!** Impossible to reduce or hide (by prediction), aggravated by compression (server) & decompress (client) times
 - Video resolution & frame-rates: very costly

79

Cloud-gaming (aka gaming-on-demand)

- A heavily invested-on, fast-growing approach to game networking (single-player ones, too!)
- High latency: ~ 80-120 ms
 - Is this acceptable?
- High bandwidth: ~ 5-50 mbits/s
- Good fit for Games as a Service (“GaaS”)
- Will it become an established model for online 3D games?



- STADIA (Google)
- GEFORCE NOW (nVidia)
- X CLOUD (Microsoft)
- PlayStation Now (Sony)
- luna (Amazon)

80

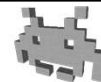
Summary: classes of solutions



- Who computes game evolution? (including physics)
 - **deterministic-lockstep** : clients
 - there may be no server at all: peer-to-peer possible
 - independent computations, same result
 - **game-status snapshots** : server
 - clients are just visualizers
 - maybe with interpolation / extrapolation
 - **(distributed physics** : both clients and server)
 - clients in charge for own agent(s)
 - server in charge for env. / NPCs
 - **client-side predictions** : both clients and server
 - clients “predict” (just for local visualization purposes)
 - server “corrects” (it has the last word!)
 - **cloud gaming** : server
 - clients just collect input, show video stream
 - server does everything

81

Summary : rules of thumb



- Comparing network paradigms
 - **Deterministic Lockstep**, when
 - determinism can be assumed
 - few players (up to 4-5)
 - fast + reliable connection (e.g., LAN)

← Good fit for, e.g., RTS!

} or, slow paced game
 - **Game-status Snapshots**, when
 - game status not overly complex
 - a little latency can be tolerated

Good fit for, e.g., FPS !
 - **Client-side prediction + server correction**, when
 - game status not overly complex
 - **Cloud gaming**, when
 - Bandwidth not a problem, latency tolerable
 - Clients capabilities not assumed

82