MULTIPLAYER PLATFORM

Most games follow the Entity-Component-System (ECS) pattern to some degree. Every “thing” in the game world is represented by entities. These can be characters, trees, parts of cars or entire spaceships, depending on your game.

Entities are defined by their components, which group the properties that describe their state. Components can be shared between different entities, which leads to a modular design. For example, both characters and trees can share a Physical component that defines their position and rotation, whereas an Inventory component will be used only by characters.

An entity only consists of an ID and a container of components. The idea is to have no game methods embedded in the entity. The container doesn't have to be located physically together with the entity, but should be easy to find and access. It is a common practice to use a unique ID for each entity.

These entities and components are brought to life by systems, typically threads that deal with the different aspects of the game. The physics system updates the Physical component, the AI system will observe the world and make decisions, the game logic system will make the game world behave in the way its designers intended, and so on.

The limitation of this approach comes from the fact that all these systems run on a single server with limited capacity; each system is competing for cycles with every other system.

it is cheaper, simpler and infinitely more scalable and reliable to develop applications that run in clusters of commodity hardware instead of running on monolithic mainframes.

Building distributed systems is hard, insanely hard. You need to debug race conditions, write netcode, think about concurrency and consistency, try to understand Paxos, give up and implement Raft, build fault tolerance into your design, have a strategy to deal with network latency and packet loss, put metrics and monitoring systems in place, and a myriad of other issues. None of which are remotely related to the fun and joy of making games!

Fortunately, it is possible to formulate this problem in an elegant way, which preserves the simplicity of the ECS pattern but makes it possible for a platform to abstract away all of the unpleasantness of building distributed systems.

The key innovation is introducing the concept of workers. We take each system in ECS and replace it with a distributed system; each distributed system is built out of many workers, each of them being a program that can simulate some components of a subset of the entities in the world.

The most immediate examples of what workers can be, are general-purpose game logic workers, and physics engines. Perhaps surprisingly, a game client is also a worker that runs on the player's machine, but that isn't treated in any special way otherwise.

It's also possible to use workers for specialized simulation that doesn't fit the game logic workers. These could include anything from accurate weather simulation to flocking behaviors.

What are the complexities of each type of worker?

Which workers need to communicate often, and with whom (Coupling)?

How can we trust the solution of a worker?

What if a particular location of map is heavily occupied?

How do you store a large simulation?

How do you share map information with fast and efficient compression?

How possible is it to have no leader and yet can agreement?

CPU vs GPU

GPUs - SIMD (in lock step, VLIW)

CPUs - MIMD, how manage close (local) parallelism?

How they use far parallelism with cores / multithreads.

How complex the CPU pipeline is, what a CPU pipeline stall is: structural, data, control.

GPUs work with small chunks of daat at once (from GPU RAM) called bins, and one command processor commands all the units to work.

GPUs may also like CPU ALU, they want to utilize their adders and multipliers amd dividers + special functions (Math) in parallel with help of VLIW instructions.

Since GPUs don't have branching, VLIW is a nice way to run the cores, because are SIDM.

Lock-step means no need of too many synchronization wires, and too many command cores.

Why is GPU not the prevalent CPU architecture? CPU + embedded processors + GPU? (unpredictable) +_ (predictable, provable, nocache, no branch prediction) + (SIMD)?

What is the benefit of GPU sharing the same instruction set with CPUs (RISC-V)?

Modern CPUs have deep pipeline (like automobile factory).

Pipeline stall / structural hazard (slow). Over sequential dependencies. Unpredictable branch heavy code.

Virtual function -> entirely unpredictable. Gone case.

Branch prediction when fail, rollback => serious.

Intel core, Intel Pentium, AMD bulldozer: pipeline stages.

How binning work?

What is loose coupling and what is tight coupling.

It depends upon the code that uses it.

Entity – all variable are used closely in update(), so definitely its tightly coupled.

But if we are processing the data separately, then no more it is any tightly coupled.

In essence the component-system architecture respects both data cache and code cache.

Isnt this like GPU already? They do the vertex shader first and then fragment shader. (other shaders) - basically lock-step (ex.) and they love purity.

A shared common interface among very implementations.

How to check if side effects a = sort(a).

How matrix is structured / dense or sparse?

Should we assume priority?

GCC how structure padded/aligned attribute?

This can make designing OS beginner friendly again (through decoupling)? (RISC-V simpler to implement, in some cases faster.) Libraries for making an OS. Kind of like RISC-V. An open source software interface for application (the internal OS design can be customized).

No more android, Windows, only the SIA will matter.

How can this SIA be optimizes for high concurrency, high data throughputs.

Can an interface's throughput/latency on a given architecture be predicted, given a use case.

Then, can given a group of similar use cases, the SIA be auto-designed (through some kind of space search)?

Can then an appropriate implementation be packed together by compiler?

Can an OS present multiple SIA based on application usage / request? Think of every application requesting an SIA kind of like OpenGL / Vulkan / DirectX.

Can the same thing be done for processors?

Interface – Implementation

Is more indirections always solve problem? When is it too much?

Is functional decoupling better?

Localized side effects always better, all in one place – overwatch.

Also about versioning. Interface more important than versions. As long as version remains same, its ok.

How can transition systems be used for this (maybe this could be a better way to define algorithms).

Each level of transition systems is an interface (think actions). There is a hierarchy of transition systems.

Tree of transition systems. Using it to model concurrent systems.

Bitonic sort / random sort.

As you can see, each sort has the same interface, but a different implementation.

What one architecture needs is not necessary by others: disk defragmentation, array sorting, precomputed buffer.

Each architecture has its own needs. How can a compiler insert array sort before loop, without changing code behaviour (but improving speed).

Can we get program overview (insight) (without reading it) at different levels. Some kind of p login / website / startup?

Just insight like Healthline insight on food, combined from research papers.

GitHub is already doing it with language detection, we can count ifs, loops, variable in functions, dependencies, dependents, (security). (performance for a given arch.)

Can a game / app. Dynamically pick up an implementation based upon the architecture of machine it is currently running on / the software already available on the machine.

In 10 packages implement X interface either can we dynamically choose any one of them based on what is currently available on the system. May solve the yet another plugin problem.

How can compiler optimize, should you tell it? (like in Unity NativeArray<Temp, TempJob>

Struct<as is>, struct<A>, <a>, <?>.

How to specify which implementation (A, A, A, B)?

Serialized data access is performant.

How can data structures be designed for serial access. If you need some field next, why not let the field be the next in the structure. (depends upon the various algorithms using it, which is why profiling is important)

Amdahl's law based optimization by the compiler itself.

How should various fundamental data structures be designed such that they are performant for CPUs / GPUs?

Is it possible to separate out the optimization logic / synthesis from the programmer code? Can the compiler do automatic profiling (OM) for the appropriate implementation?

How high-level algorithms can be written.

Is there a way to quickly choose appropriate implementation based on constraint (LP* -> IP)?

Maybe using a suggestion from programmer?

Moving towards functional design – helping in separating out logic from data (plain old data).

How can system be made more concurrent , more decoupled?

Testing performance of data structures for ECS friendly. How performance with perf-stat.

Why should OS be written in C, if we want OS to execute various algorithms. Why do we want to use high level languages to implement OS.

Which is the higher layer and can do without optimization? This keeps on changing as we try to solve larger and larger problems.

Promises. Promise.all() requires sync. Basically like "barrier" (job) in burst compiler. But its always better to minimize synchronization.

How long can we delay synchronization.

If I am text processing 200 different texts, why do I need to wait till all of the 200 texts are available? But many JS libs do that (I do that too)?

It would be cool if promises could be handled transparently by for-each higher-level logic. Automatic handling of coroutines just like Node.js event loop.

When to do lazy processing and when to do it now. Haskell approach. Lazy execution model vs eager execution model.

Does this have something to do with CPU cache size?

Similarly, when is it better to have linked lists, instead of contiguous arrays?

Yes, in some cases linked lists are more performant. How can this idea be extended to trees, graphs.

(side quest: can program synthesis / program proof be done for cryptocurrency?) (soln.) space search is critical for cryptocurrency.

Mutation based AI synthesis, could it be a candidate for space search.

Randomize NP algorithms suitable for cryptocurrency workloads.

So they are also suitable for concurrency because – low data high space search (complex).

NP hard, what is it? Can it be used for concurrency?

Such problems in game design?

Data intensive – or space search intensive?

Then how do concurrent programs synchronize?

(SpatialOS) Improbable is using the concurrency of ECS architecture to simulate large worlds?

How user input is given to server, which gives back authoritative state of player and how the manage packet loss (rate goes up) and high latency (prediction).

How can we minimize latency at every stage of processing. Processing physics at various LODs.

How to distribute loads for better concurrency (remember MIMD systems). How to minimize communication overhead. How to share low LOD information with low latency. How to minimize latency to screen. This is very critical for VR/AR.

Net-code: Why is it hard? They expect loss. It involves unpredictability. How can we exploit client-CPU power if available?

ECS:

- Burst compiler
- - C# job system
- Optimized native code using LLVM.
- Stand on the shoulders of giants.
- No GC (or custom). Custom mem-alloc.
- Can components be made separately and shared among people (decoupling)?

1990s actor model.

200s component model (game object?)

Manage complexity on a quickly growing codebase. Visitor pattern? Functional types pattern?

World -> Systems, Entity -> Components

Systems: no fields, only behaviours, no game state (interfaces).

Components: (interfaces) store game state, no behaviours, no functions.

Systems only care about what components they operate on.

Systems can be added with ease.

Pure functions, reducing side effects, preferred in angle area. Behaviours single call site. Singletons, deferment.

Prediction, misprediction of action.

Authoritative server input.

Your input -> server -> you.

Check if matches, if not re-predict the current state after applying inputs.

What if input packet lost. Feedback loop because of it.

Input sliding window kept (for loss).

How to handle ping delay Decoupling. More constraints.

System using components singletons, tuples, archetype. More constraints => better usually? Hole of success) (depends on each?)

Data oriented technology stack (DOTS) ECS.

Entity – id.

Component – physical, speed-comp., health., inventory.

Systems – Logic 1, Logic 2, Physics, AI, game (inventory).

What is level of detail (LOD)?

How can it be suitable for physics (systems)?

Bar graph.

How can super large physics effects (effects that need a large no. Of components to come together work).

How can we share the information of a space (map area) state in a compressed way and then stream in more accurate information (kind of like JPEG) (image sharpening).

How can a map location information be maintained?

Distributed data storage – DS class.

Coding scheme – useful for large maps (ex persistent).

(Satellite looking at night sky)

(LOD – compute in local blocks)

Could be useful for protein simulation in cells. Body subsystems games. Life to cells, operation of organelles. Games at different scale. A bug's life. How like perspective change with scale.

You will get 0 this semester if you don't make a game, need pressure. But I also need to be interested in it.

Also like the concept of edge of tomorrow, that movie where the brain in end, and games. Resume exactly at a point and try again (in multiplayer games). How can it be done? Dota, Apex legends. Checkpoints in multiplayer games. Using AI? Respawn? -> maintain as player? Godlike?

Replaying full match as with AI checkpoints? Think experiment = fun?

Game interface, separate implementation? Why pay for developers?

How to do away with authoritative server and make a fully distributed networked game?

Double / triple buffering for computing physics objects instead of copy can help avoid data hazards (partially modified buffers).

Ralf brown interrupt list. (Interface description of registers). This changed.

Interface description on music (rock) (soulful) (like Youtube) (more insights – better?) (boy / girl singing) (interfaces on music instruments?) (implementations))

Ex- of some common interfaces different implementation. SDS drill bit, Screw bit, clamp bit. USB.

Iceberg based interface, bottom can change because you don't see it. So for ship. Person carrying jhard or bomb? The look of buildings? Staircases, lifts, escalators? Outside interface same, different from inside.

When is homogeneity bad? When should interface change based on use case?

When to use Einstein's relativistic physics vs when to use Newtonian physics.

MotoGP: when to use skeleton renderer, when to use rough renderer, when fully textured renderer.

: Game: scrap mechanic, creativerse. Lots of actions, how they interact.

Collision detection – physics – explosion simulation.

Cloth simulation, LOD. When more from far, but accurate from near, more users => more realistic simulation. Tree falls in jungle (some one to see). How to do persistent maps.

Space trash, someone keeps simulating it disaster => necessary in space games => more players.

Function tags, just like package tags, repo tags, data structure tags, interface tags, implementation tags.

Is uniform interface always better. SSD (flash) vs intel optane. (DSA) - everything will not autoboost.

Question remains, how to parallelize inherently serial operations. What is truly inherently serial? Can we get some numbers? (min max range).

How to do predictable execution. (when necessary?) How can it be implemented? When is it an implementation, and when an interface (flag)?

(Choosing a particular interface)

Component vs inheritance.

Command pattern (closure).

Why copy physics component, when you can do buffer swap? (double buffering, triple buffering)

In OOP not only data cache is trashed, but also the code cache.

AI + physics + game logic in every update(). Bouncing all over the place. Perf problems.

Memory slow, better to recalculate on the fly.

Critical areas: physics / graphics.

Graphic programming languages. Coupling strength.

What is sequential code? Synchronization / dependencies.

When mother cuts vegetable I can still fry onions and spices. Maze runner interesting.